DPKM: A Key Management Scheme Facilitating WireGuard-based Data Plane Encryption in SDN

Luke Hengstenberg
878876

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Master of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

December 14, 2020

SpLD Applies

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....14/12/2020.....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....14/12/2020.....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....14/12/2020.....

Abstract

Industry focus has shifted from static, hardware-centric networking architectures to dynamic, software-centric solutions, driving widespread investment in Software Defined Networks (SDN). SDN streamlines network management by moving forwarding logic from data plane entities to a controller, enabling programmatic configuration of nodes and real-time adjustments to the topology. Encrypted node-to-node communication enhances security but must be manually established, inhibiting the dynamicity of SDN and creating management challenges. Evidence exists that automated data plane encryption could be achieved through integration of a key management scheme. However, no proof of concept could be identified to support this hypothesis.

This thesis documents the enhancement of existing SDN software components to support automated data plane encryption through a novel key management scheme called DPKM. Extensions are proposed for OpenFlow-based SDN, using the tunnelling protocol WireGuard to build encrypted connections between nodes. The controller is modified to administer the key management lifecycle and perform eight procedures over an extended version of the OpenFlow protocol. The nodes support a WireGuard interface and adjust the tunnel configuration based on the controller's instructions. Full operation of DPKM functionality is provided by an intuitive user interface (UI) connected to the controller through REST API's.

Development took place across three VM's, using four GitHub repositories to publish extensions to the software of Floodlight, Open vSwitch, OpenFlow, and Floodlight-webui. DPKM was built and tested for a basic SDN with one controller and two nodes. The contributions of this thesis have helped advance the field of study by proving the feasibility of automated key management and dynamic encryption in SDN. The delivered product was critically evaluated, highlighting the need for further work before it can have applicability in industry. Overall, DPKM achieved its objectives and is a stepping-stone for a multitude of future research directions.

Acknowledgements

Developing DPKM was one of the biggest challenges I have faced in my academic career. Adapting to the steep learning curve had its lows – making little progress for months – but once the initial obstacles were overcome it was an enriching experience. The project taught me many lifelong skills and fuelled a passion for a field I had no previous knowledge in. I would like to thank my supervisor, Dr Pardeep Kumar, for his support and encouragement throughout, especially when I felt the desire to switch projects. His faith in my abilities was a driving factor and prevented me from succumbing to the pressure. I would like to thank my partner Georgia for making my life as easy as possible and being her wonderful, supportive self. I would like to express my gratitude to my friends Thomas, Greg, Jamie, and Mary, for assisting with proof reading, providing essential feedback, and spreading positive energy. Finally, I would like to thank all my family and friends for going above and beyond to keep my spirits up throughout.

Contents

List of Tables **vii**

List of Figures **viii**

List of Acronyms **xix**

1 Introduction **1**

1.1 Motivation 2
1.2 Scope and Novelty 3
1.3 Objectives 5
1.4 Methodology 5
1.5 Outline 6

2 Background Research **8**

2.1 Software Defined Networking 8
2.2 Key Management and Cryptography 12
2.2.1 Cryptographic Approaches 12
2.2.2 Key Management Lifecycle 15
2.2.3 Third-Party Systems 19
2.2.4 Key Management Protocol Review 20
2.3 Technology Choices 25
2.3.1 OpenFlow 25
2.3.2 WireGuard 28
2.3.3 Network Node 32
2.3.4 Controller 33
2.3.5 Application Plane 35

2.4 Related Work 35

3 Planning and Specification **38**

3.1 Requirements 38

3.1.1 Cryptographic Requirements 39

3.1.2 Functional Requirements 40

3.2 Use Cases 41

3.3 Software Development Life Cycle 44

3.4 Progress Timeline 47

4 Designing DPKM **50**

4.1 Architectural Design 50

4.1.1 Controller 51

4.1.2 Node 57

4.1.3 Application 58

4.1.4 Interfaces 65

4.2 Procedures 67

4.2.1 Configuring WireGuard 69

4.2.2 Unconfiguring WireGuard 71

4.2.3 Adding Peers 73

4.2.4 Deleting Peers 75

4.2.5 Starting Communication 77

4.2.6 Ending Communication 79

4.2.7 Rekeying 81

4.2.8 Key Revocation 83

4.3 Messages 86

5 Implementing DPKM **88**

5.1 System Setup 89

5.2 Extending OpenFlow 93

5.2.1 OpenFlow Experimenter Headers 93

5.2.2 Defining OpenFlow Extensions 96

5.2.3 Integrating OpenFlow Extensions 101

5.3 Overview of Software Extensions 102

5.3.1 Floodlight . . . 103
5.3.2 Open vSwitch . . . 109
5.3.3 Application UI . . . 111
5.4 Developing Procedures . . . 113
5.4.1 Configuring WireGuard . . . 113
5.4.2 Unconfiguring WireGuard . . . 120
5.4.3 Adding a Peer . . . 125
5.4.4 Deleting a Peer . . . 133
5.4.5 Starting Communication . . . 139
5.4.6 Ending Communication . . . 144
5.4.7 Rekeying . . . 150
5.4.8 Compromise and Revocation . . . 154
5.5 Testing . . . 157
5.5.1 Test Environment . . . 157
5.5.2 Aims and Metrics . . . 157
5.5.3 Methodology . . . 158

6 Evaluation and Discussion 160
6.1 Requirement Analysis . . . 160
6.1.1 Cryptographic Requirements . . . 161
6.1.2 Functional Requirements . . . 162
6.2 Risk Analysis . . . 164
6.3 Results Discussion . . . 167
6.4 Achievements . . . 170
6.4.1 Enhancements to Field of Study . . . 170
6.4.2 Measuring Success . . . 172
6.4.3 Noteworthy Deliverables . . . 172
6.5 Limitations . . . 173
6.6 Future Work . . . 176
6.6.1 Testing and Analysis . . . 177
6.6.2 Continuing Development . . . 178

7 Conclusion 180

Appendices **195**

A Record of Supervision **196**

B Setup instructions for DPKM system **199**

B.1 Prerequisites 199

B.2 DPKM Floodlight Controller/UI Setup 200

B.3 DPKM OVS Node Setup 202

B.4 Capturing Packets in Wireshark 204

C Full Code Description **206**

C.1 Floodlight 206

C.1.1 ConnectionProvider.java 206

C.1.2 Dpkm.java 207

C.1.3 DpkmCompromiseNodeResource.java 209

C.1.4 DpkmConfigureWG.java 210

C.1.5 DpkmConfigureWGResource.java 212

C.1.6 DpkmConfigureWGWebRoutable.java 212

C.1.7 DpkmEndCommunicationResource.java 213

C.1.8 DpkmError.java 213

C.1.9 DpkmErrorHandler.java 214

C.1.10 DpkmFlows.java 216

C.1.11 DpkmManagePeerResource.java 216

C.1.12 DpkmPeers.java 217

C.1.13 DpkmRekeyResource.java 217

C.1.14 DpkmRevocationResource.java 218

C.1.15 DpkmStartCommunicationResource.java 218

C.1.16 DpkmSwitch.java 218

C.1.17 IDpkmConfigureWGService.java 219

C.2 Open vSwitch 219

C.3 Application UI 223

List of Tables

4.1 The OpenFlow experimenter message extensions needed for DPKM to operate. . . 87
4.2 The required OpenFlow error message extensions reporting issues during DPKM operations. 87

5.1 OpenFlow object types with the corresponding experimenter header and purpose. . 94
5.2 Typical fields in an OpenFlow Experimenter header. 94
5.3 All classes implemented in the net.floodlightcontroller.dpkmconfigurewg Floodlight package. 103
5.4 Possible states of a peer connection. 131

6.1 Project risks, likelihood of occurrence, severity of impact, and mitigation strategy, taken from the preliminary specification document. 164
6.2 Results from the eight tests performed with one controller and two nodes, measuring the average time to complete various DPKM procedures in milliseconds. 167

A.1 Meeting minutes part A taken as record of supervision. 197
A.2 Meeting minutes part B taken as record of supervision. 198

List of Figures

2.1 Topological model of a typical SDN architecture. 10
2.2 Two networking nodes structured in traditional hardware-centric architecture. . . . 11
2.3 Two networking nodes structured as an SDN by separating control and data planes. 11
2.4 Example of a public-key or asymmetric cryptographic approach to encryption, using the public key for encryption and private key for decryption. 13
2.5 Example of a symmetric cryptographic approach to encryption using the symmetric key for both encryption and decryption. 14
2.6 Flow diagram depicting the transitions between the four phases and inner functions of the key management lifecycle. 18
2.7 Sequence diagram illustrating the message exchanges of the key management protocol IKEv2. 23
2.8 Diagram illustrating the WireGuard protocol setting up a network tunnel between two interfaces for encrypted data transport. 31

3.1 Use Case Diagram depicting the DPKM functions for setting up the WireGuard interface and network tunnels. 42
3.2 Use Case Diagram illustrating the DPKM functions for tearing down the WireGuard interface and network tunnels. 43
3.3 Use Case Diagram illustrating the DPKM functions for identifying and handling compromised network nodes. 44
3.4 An example diagram showing the phases of the Waterfall SDLC model. 45
3.5 An example diagram showing the phases of the V-Shaped SDLC model. 46
3.6 An example diagram showing the phases of the Iterative and Incremental SDLC model. 47
3.7 The original project completion timeline created with Microsoft Office Timeline [Tim20] 48

3.8 The updated project timeline created after extension of the deadline from August to December. 49

4.1 The full architectural design of the DPKM system to be developed. 51
4.2 UML class diagram structuring the required classes for realising the DPKM controller. 54
4.3 Focused class diagram structuring the classes comprising the REST system in the DPKM controller. 55
4.4 Model of the DPKM controller proposing the schema of three database tables: ConfiguredPeers, CommunicatingPeers, and ErrorLog. 56
4.5 Mock of the application UI Home Page. 60
4.6 Mock of the application UI "Manage Switches" page for performing DPKM functionality. 60
4.7 Add Connection modal popup form for setting up a new peer connection between nodes. 61
4.8 Status window indicating successful setup of a peer connection (left) or the error that was encountered (right). 62
4.9 "Manage Switches" page after the "Modify" button has been clicked. 62
4.10 Modify Connection modal popup for ending encrypted communication only (blue button) or removing the connection entirely (red button). 63
4.11 Status window indicating successful termination of an encrypted peer connection (left) or the error that was encountered (right). 63
4.12 Mock of the application UI Home Page with "Action Needed" from the administrator. 64
4.13 Action Needed modal popup for handling a compromised node by reconfiguring and rekeying (blue button) or full termination (red button). 64
4.14 UML component diagram mapping the relationship between DPKM components with four main interfaces - northbound (1), southbound (2), inner node (3), and node-to-node (4). 65
4.15 State transition diagram for a node in the DPKM system, plotting possible states and transitions between them. 68
4.16 Sequence diagram designing the procedure for configuring WireGuard. Steps 1-9 plot the expected sequence of functionality, 6A-9A consider an error case. 70

4.17 Sequence diagram planning the procedure for unconfiguring WireGuard. Steps 1-9 plot the expected sequence of functionality, 6A-9A consider an error case. "End Communication" procedure called if the node has any connections. 72
4.18 Sequence diagram describing the procedure for adding a peer. Steps 1-9 plot the expected sequence of functionality, 3A-7A remove an existing peer, 6B-9B consider an error case. 74
4.19 Sequence diagram illustrating the procedure for removing a peer. Steps 1-9 plot the expected sequence of functionality, 6A-9A consider an error case. 76
4.20 Sequence diagram displaying the procedure for starting communication. Steps 1-5 plot the functionality for setting up the flow, 6-11 consider how a packet is handled. 78
4.21 Sequence diagram describing two variations of the procedure for ending communication. Type A (orange) reconstructs the flow for unencrypted communication, Type B (purple) terminates all communication. In both variations the peer connection is removed. 80
4.22 Sequence diagram planning the procedure for rekeying the WireGuard interface after expiry of the set cryptoperiod. 82
4.23 Sequence diagram depicting two variations of the revocation procedure after detection of a compromise by security software. All communication is ended with the compromised node and then the node is either reconfigured (type A) or fully terminated (type B). 84

5.1 Diagram of the full DPKM network topology, consisting of three VMs structured as a bridged network with unique IP addresses for simulating an SDN. 89
5.2 DPKM User Interface labelled with the functions called and the procedures initiated. Multiple table fields have been removed for visibility. 112
5.3 Screenshot captured from the application UI, displaying the modal popup form for initiating the Configure WireGuard procedure. 114
5.4 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_set_key message. 115
5.5 Screenshot of the node console showing the transition from a blank WireGuard interface to a configured interface with a public key. 117
5.6 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating successful configuration of the node. 118

5.7 Compilation of two screenshots: one showing the status response to a set key message on the application UI, and one showing the output in the controller console. 119
5.8 Screenshot of the configured switches table displayed in the application UI with a delete button (red circle) for unconfiguring a node. 120
5.9 Popup notification prompting the administrator to confirm initiation of the Unconfigure WireGuard procedure. 120
5.10 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_delete_key message. 122
5.11 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating that the node has been unconfigured. 124
5.12 Compilation of two screenshots: one showing the status response to a delete key message on the application UI, and one showing the output in the controller console. 125
5.13 Screenshot captured from the application UI, displaying the modal popup form for creating a new peer connection by initiating the Add Peer procedure. 126
5.14 Popup error notification informing the administrator that the connection already exists. 128
5.15 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_add_peer message. 129
5.16 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating that a peer has been successfully added. 130
5.17 Compilation of two screenshots: one showing the status response to an add peer message on the application UI, and one showing the output in the controller console. 132
5.18 Side-by-side comparison of the node A and node B consoles after the add peer procedure, displaying the WireGuard interfaces connected as peers. 133
5.19 Screenshot of the connected peers table displayed in the application UI with a delete button (red circle) for removing a peer connection. 133
5.20 Popup notification prompting the administrator to confirm initiation of the Delete Peer procedure. 133

5.21 Screenshot of Wireshark, displaying a captured of_dpkm_delete_peer message. . . 135
5.22 Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating that a peer has been successfully removed. . . 136
5.23 Compilation of two screenshots: one showing the status response to a delete peer message on the application UI, and one showing the output in the controller console. 138
5.24 Side-by-side comparison of the node A and node B consoles after the delete peer procedure, displaying two unconnected but configured WireGuard interfaces. . . . 138
5.25 Two screenshots of the connected peers table, taken from the application UI, showing the change in the table after clicking start (green circle) to begin communication. 139
5.26 Screenshot of the Wireshark packet analyzer, displaying the full content of a captured flow_add message for adding a new flow entry to node A. 143
5.27 Screenshot of the node A console confirming setup of the flow for encrypting and routing packets through interface wg0. 144
5.28 Screenshot of the node A console after running the ofproto/trace command to simulate arrival of a packet matching the flow entry conditions. 144
5.29 Screenshot captured from the application UI, displaying the modal popup form for ending communication toggled by clicking the delete button in the connected peers table. 145
5.30 Screenshot of the Wireshark packet analyzer, displaying the full content of a captured flow_mod message for modifying the flow table of node A. 148
5.31 Screenshot of the node A console after running the ofproto/trace command to simulate arrival of a packet matching the unencrypted communication flow conditions. 149
5.32 Screenshot of the Wireshark packet analyzer, displaying the full content of a captured flow_mod message for modifying the flow table of node B. 149
5.33 Screenshot of the node A console after running the ofproto/trace command to simulate arrival of a packet matching the terminate communication flow conditions. 150
5.34 Screenshot of the application UI showing expiry of the node A cryptoperiod and automated initiating of the rekey procedure, indicated by the popup notification. . 151

5.35 Screenshot of the Wireshark packet analyzer, displaying the full message exchanges of the rekeying procedure. 153
5.36 Controller log reporting on state changes and successful completion of DPKM functions during the rekeying procedure. 153
5.37 Screenshot captured from the application UI, with the modal popup form for simulating a compromise. 155
5.38 Popup notification indicating commencement of the key revocation simulator . . . 155
5.39 Screenshot captured from the application UI, displaying a fixed modal popup form that can only be closed by initiating the revocation procedure. 156
5.40 Screenshot captured from the excel spreadsheet, showcasing three of the excel formulas used to organise the data and calculate the time difference. 159

6.1 Bar Chart plotting the individual average completion time (ms) for the four single request/response tests with nodes A and B. 168
6.2 Bar Chart plotting the combined average completion time (ms) for the four multipart request/response tests with nodes A and B. 169

B.1 Example of Bridged Adapter setting in VirtualBox. 199
B.2 Example of Wireshark setup, with OpenFlow filter and lua script for capturing DPKM packets. 205

Listings

5.1 MySQL database credentials providing access to Floodlight. 91
5.2 ConnectionProvider class for creating and returning a connection object to the DPKM database. 91
5.3 Bash script performing a series of commands for setting up a network node. . . 92
5.4 DPKM OpenFlow Experimenter headers defined in the Loxigen class dpkm.c. . 95
5.5 DPKM_SET_KEY OpenFlow Experimenter message for configuring WireGuard defined in the Loxigen class dpkm_set_key.c. 97
5.6 DPKM_DELETE_KEY OpenFlow Experimenter message for unconfiguring WireGuard defined in the Loxigen class dpkm_delete_key.c. 97
5.7 DPKM_GET_STATUS OpenFlow Experimenter message for extracting the node status defined in the Loxigen class dpkm_get_status.c. 97
5.8 DPKM_ADD_PEER OpenFlow Experimenter message for adding a new peer defined in the Loxigen class dpkm_add_peer.c. 98
5.9 DPKM_DELETE_PEER OpenFlow Experimenter message for deleting a peer defined in the Loxigen class dpkm_delete_peer.c. 98
5.10 DPKM_STATUS OpenFlow Experimenter message for returning the state of the node defined in the Loxigen class dpkm_status.c. 99
5.11 Enumerator Status Flags for representing the state of the node defined in the Loxigen class dpkm_status_flag.c. 99
5.12 OXM extension for a new packet matching condition in the Loxigen class oxm-dpkm.c. 100
5.13 One of fifteen OpenFlow error extensions in the Loxigen class dpkm_error.c. . 100
5.14 Builder for the of_dpkm_add_peer message generated by Loxigen. 101

5.15 Condensed version of the receive method implementing the DPKM message listener capturing OpenFlow experimenter messages, taken from the DpkmConfigureWG.java class. 104
5.16 processStatusMessage method determining the controller-to-switch message that triggered the status response and passing the information as an object for further processing, taken from the DpkmConfigureWG.java class. 105
5.17 Receive method for the DPKM error message listener capturing error messages for logging issues and experimenter messages for resolving errors, taken from the DpkmErrorHandler.java class. 106
5.18 Condensed version of the processErrorMessage method showing 5 out of 15 cases for checking error type and calling the respective handler, taken from the DpkmErrorHandler.java class. 106
5.19 processResponseMessage method marking unresolved errors as resolved if the node is now returning a status message as expected, taken from the DpkmErrorHandler.java class. 107
5.20 DpkmConfigureWGWebRoutable class registering the REST resource classes to URLs so external calls can be made from the UI. 108
5.21 Snippet from extract-ofp-msgs for extracting OpenFlow message headers and mapping to DPKM. 109
5.22 Snippet from ofp-msg.h adding the raw (ofpraw) and semantic (ofptype) identifiers for DPKM messages. 110
5.23 Snippet from ofproto.c that checks the OpenFlow message type and calls the respective handler function. 111
5.24 POST method for configuring a WireGuard interface, taken from the DpkmConfigureWGResource.java class. 114
5.25 sendSetKeyMessage method constructing and sending an of_dpkm_set_key message from controller to switch, taken from the DpkmConfigureWG.java class. 115
5.26 OVS function for executing the underlying WireGuard configuration script, taken from the ofproto.c class. 116
5.27 Configuration script generating an asymmetric key pair and binding the private key to the WireGuard interface. 116

5.28 Extract of the OVS handle_dpkm_set_key function for creating and sending a status response to the controller, taken from the ofproto.c class. 117
5.29 Condensed version of the processConfigured method for processing the status response to a set key message and updating the database accordingly. Taken from DpkmConfigureWG.java. 119
5.30 DELETE method for unconfiguring a WireGuard interface, taken from the DpkmConfigureWGResource.java class. 121
5.31 sendDeleteKeyMessage method constructing and sending an of_dpkm_delete_key message from controller to switch, taken from the DpkmConfigureWG.java class. 121
5.32 OVS function for executing the underlying unconfigure WireGuard script, taken from the ofproto.c class. 122
5.33 Unconfigure WireGuard script shutting down OVS, removing the interface and keys, creating a new unconfigured interface, and restarting OVS. 123
5.34 The method for processing the status response to a delete key message, removing an remaining connections, and updating the database accordingly. Taken from DpkmConfigureWG.java. 125
5.35 POST method for adding a new peer, taken from DpkmManagePeerResource.java. 127
5.36 sendAddPeerMessage method retrieving peer information from the database and passing it to the sendAddPeerMessageInternal method, taken from the DpkmConfigureWG.java class. 127
5.37 sendAddPeerMessageInternal method constructing and sending an of_dpkm_add_peer message from controller to switch, taken from the Dpkm.java class. 128
5.38 OVS function decoding an of_dpkm_add_peer message and mapping to an ofputil_dpkm_add_peer struct, taken from the ofp-util.c class. 129
5.39 Command for adding a new peer to the WireGuard interface. 130
5.40 Example SQL statement used by the updatePeerInfo method to modify the connection status in the database to statusChange. Taken from the Dpkm.java class. 131

5.41 Example SQL statement used by the checkConnected method to retrieve a count of the connections matching the connection status (statusType). Taken from the Dpkm.java class. 131
5.42 sendDeletePeerMessage method initialising a switch, retrieving peer information from the database, and passing it to the sendDeletePeerMessageInternal method. Taken from the DpkmConfigureWG.java class. 134
5.43 Method constructing and sending an of_dpkm_delete_peer message from controller to switch, taken from the Dpkm.java class. 135
5.44 OVS function for executing the delete peer command, removing a peer matching the key from WireGuard interface wg0, taken from the ofproto.c class. . . . 136
5.45 Method for processing the status response to a delete peer message, sending subsequent messages based on the connection status. Taken from DpkmConfigureWG.java. 137
5.46 startCommunication method for constructing and sending a flow_add message to each peer, taken from the DpkmConfigureWG.java class. 140
5.47 Code extract from the constructFlowAdd method building a match object with four flow matching conditions, taken from the DpkmFlows.java class. 140
5.48 Code extract from the constructFlowAdd method defining three actions for OVS to perform on packets, taken from the DpkmFlows.java class. 141
5.49 Code extract from the constructFlowAdd method creating an instruction to apply all actions, taken from the DpkmFlows.java class. 142
5.50 Code extract from the constructFlowAdd method building and returning the flow_add object, taken from the DpkmFlows.java class. 142
5.51 endCommunication method for sending a flow_mod message to each peer, instructing OVS to either route packets via the controller or drop packets. Taken from DpkmConfigureWG.java. 145
5.52 Method for creating a flow_mod message that modifies the existing flow to route packets via the controller. Taken from the DpkmFlows.java class. 147
5.53 constructFlowDrop method for creating a flow_mod message modifying the existing flow with a new instruction that drops any matching packets. Taken from the DpkmFlows.java class. 147

5.54 handleFlowRekeying method for creating a flow_mod message to route packets via the controller before rekeying and restore the encrypted flow after rekeying. Taken from the DpkmFlows.java class. 152
B.1 Dependencies needed to build and use Open vSwitch. 203

List of Acronyms

AH Authentication Header

API Application Programming Interface

CA Certificate Authority

CPU Central Processing Unit

DB Database

DH Diffie-Hellman

DoS Denial of Service

DpId Datapath Id

DPKM Data Plane Key Management

ECDH Elliptic-Curve Diffie-Hellman

ESP Encapsulating Security Payload

GUI Graphical User Interface

HTTP(S) Hypertext Transfer Protocol (Secure)

ICMP Internet Control Message Protocol

IDS Intrusion Detection System

IKE Internet Key Exchange

IoT Internet of Things

IP Internet Protocol

IPsec Internet Protocol Security

KB Kilobyte

KDC Key Distribution Center

KINK Kerberized Internet Negotiation of Keys

KTC Key Translation Center

MAC Media Access Control

MiTM Man-in-The-Middle

ONF Open Networking Foundation

ONOS Open Network Operating System

OXM OpenFlow Extensible Match

OVS Open vSwitch

PKA Public Key Authority

PKI Public Key Infrastructure

(P)RBG (Pseudo) Random Bit Generator

QoS Quality of Service

RA Registration Authority

REST Representational State Transfer

RT Round Trip

SA Security Association

SDLC Software Development Life Cycle

SDN Software Defined Network

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

TTL Time To Live

TTP Trusted Third Party

UDP User Datagram Protocol

UI User Interface

VM Virtual Machine

VPN Virtual Private Network

WSN Wireless Sensor Network

Chapter 1

Introduction

Over the past decade the demand placed on modern computer networks has drastically increased and become a major technological obstacle. Major structural transformations have been necessary to cater to the growing diversity and requirements of multimedia content, heighten the availability of resources for cloud-computing services, and handle traffic from heterogenous devices, with an explosive surge in mobile usage. Optimising performance for frequently changing network conditions posed a significant challenge to the IT industry. New architectures were required that support adaptive modification of packet forwarding and traffic handling logic in data plane nodes. Traditional hardware-centric networking approaches lack the dynamicity to effectively meet fluctuating demand, requiring individual configuration of nodes for new policies and routing logic. These factors shifted industry attention to software-centric solutions, leading to the emergence of Software Defined Networking (SDN).

The SDN architecture streamlines network management and enhances performance by utilising software properties like virtualization to programmatically configure nodes and make real-time adjustments to the network topology. SDN moves intelligence and control logic from the packet forwarding entities to a centralized controller that can integrate changes based on the requirements of the running applications or network operator. Applications make requests to the controller through northbound API's which are translated into control logic and applied to the nodes through the southbound API. OpenFlow is recognised as the global standard for southbound communication and can be directly attributed to the emergence of SDN [McK+08]. Node-to-node communication typically involves sending packets via the controller or through pre-configured network tunnels.

As SDN adoption in industry nears the ten year mark the architecture is no longer in its infancy and is rapidly becoming a standard networking paradigm. Wider adoption naturally leads to an increase in malicious groups targeting vulnerabilities in SDN security. One method to heighten security is encrypting communication channels to encode information transported between SDN entities. OpenFlow can implement Transport Layer Security (TLS) to encrypt and authenticate southbound communication. Encrypting node-to-node communication is typically achieved through manual configuration of network tunnels between virtualized switches. Tunnels must be established before deployment or whilst nodes are shutdown. Manual configuration of the data plane goes against the core philosophy of SDN, inhibiting dynamicity, and creating security and management challenges. There is a clear need for an automated tunnel configuration process that supports SDNs flexibility and can be facilitated from the application or controller.

This thesis addresses the problem of *how to deliver a key management scheme that facilitates data plane encryption through automated configuration of network tunnels with the ease of other SDN functions*. Encryption relies on cryptographic keys, meaning a key management scheme must be developed, integrated within SDN software components, and operate over the OpenFlow protocol. Such a scheme was developed and given the name *Data Plane Key Management* (DPKM), extending the software of the controller, node, application, and connecting interfaces to support key management operations.

1.1 Motivation

Wider SDN adoption in industry is evident by the rapidly increasing market value, rising from 0.41 billion USD (2013) to 13.7 billion USD (2020), with a forecasted increase to 32.7 billion USD by 2025 [Hol20b; RM20]. The driving factors behind SDN growth are the increasing demand for cloud services, enterprise mobility, and data center deployments. Major technology companies like Cisco, Huawei, and Oracle have invested a substantial amount into the market and harness SDN to reduce operational costs [RM20]. Increasing usage in data centers is apparent, with a forecasted rise in SDN traffic from 0.8 zettabytes in 2015 to 7.4 zettabytes by 2021 [Hol20a]. The booming industry has created a multitude of opportunities within the SDN research field and was one of the motivating factors behind this thesis. Constructive improvements to the SDN architecture would not only advance the field of study but also have significant real-world applicability.

Vodafone (2019) produced a whitepaper titled *"SDN Research Report 2019"* in which 1,891 business decision makers were interviewed on the benefits and challenges of SDN [Vod20]. The report found the biggest challenge to be data security (73%), with 88% of businesses investing in security improvements [Vod20]. Network security is of paramount importance and transmission of unencrypted traffic has the potential to leave sensitive information exposed. Manual configuration of network tunnels increases the difficulty of encrypting all communication channels, creating potential weak points in the network. Integrating DPKM in SDN would permit full control of encrypted connections and reduce human error by enhancing the management tools available to the network operator. Moreover, encryption can enable flow isolation and division of larger networks into smaller segments, allowing traffic to be confined to specific sectors to reduce the risk of data leaks.

The concept of facilitating data plane encryption through key management was theorised by Braadland (2017) in their master's thesis but never implemented due to the complexity and novelty [Bra17]. Extending and improving on this research was the primary purpose behind this project since the successful development of DPKM and automated data plane encryption would progress the subject area. As SDN encompasses new research fields such as Internet of Things (IoT), Wireless sensor networks (WSN) and 5G, there is an increasing volume of wireless transmission between distributed nodes over exposed channels [Mut+19; NHA17; You+17]. Better encryption mechanisms would also provide security benefits to these fields if the SDN architecture is used.

1.2 Scope and Novelty

The focus of this thesis is on designing and implementing SDN mechanisms for configuring WireGuard-based network tunnels between data plane nodes, with a corresponding key management scheme called DPKM. Novelty lies in the development of a functional system, with unique key management extensions to the OpenFlow protocol and SDN software components. The state of the art, explored in section 2.4, conceptualised key management for data plane encryption but never produced a proof of concept. Moreover, a thorough investigation of the subject area showed that WireGuard has never been used within SDN and failed to identify relevant key management designs for controller, node, and application software extensions. This thesis initially intended to implement the theory of Braadland's thesis (2017) but found a multitude of design flaws and misconceptions in the proposed OpenFlow extensions that led

to an entirely new project, with the only remaining similarities being the technology choices and raw motives [Bra17]. The scope of this thesis will include a combination of the following parameters to produce a novel piece of research:

- Building a functional SDN with one centralized controller and two nodes.
- Developing a basic application user interface (UI) that initiates key management operations and manages the network topology.
- Creating northbound interfaces for passing requests and data between the UI and controller.
- Extending the controller software for facilitating key management via OpenFlow.
- Adding a database for storing key and node information.
- Extending the OpenFlow protocol to support new DPKM messages and pass requests between the controller and nodes.
- Modifying switch software to process and execute key management functions.
- Implementing WireGuard interfaces in the nodes that can be connected and used for encrypting IP packets.
- Testing functionality and performance.

The following aspects are beyond the scope of this thesis but may be optionally included if time permits:

- Building a functional SDN with more than two nodes.
- Routing all types of packets by encapsulating as IP. (WireGuard is layer three only – 2.3.2).
- Setting up HTTPS/TLS/SSL for the OpenFlow channel and northbound API.
- Implementing a compromise detection mechanism such as an Intrusion Detection System (IDS) for initiating key revocation.
- Performance testing with alternatives such as IPsec tunnels instead of WireGuard.
- Developing an advanced UI with sophisticated styling and login mechanisms.
- Distributed SDN with multiple controllers.
- Industry standard testing and deployment.

1.3 Objectives

- Explore the field of research and review the state of data plane security and SDN key management.
- Build an understanding of existing protocols and available technologies for executing the project.
- Formulate and adhere to a set of cryptographic and functional requirements.
- Choose a Software Development Life Cycle (SDLC) model and plan the development timeline.
- Identify use cases and design the procedural functionality of the system.
- Enhance the state of the art and produce a proof of concept with extensions to OpenFlow, UI, controller, and switch software that facilitate node-to-node encryption with WireGuard.
- Administer the main phases of the cryptographic key management lifecycle (2.2.2).
- Demonstrate a working SDN system and satisfy the full list of outputs in the project scope.
- Critically evaluate the final product and design decisions to identify achievements and limitations.
- Recommend improvements for extending the findings of this thesis.

1.4 Methodology

The methodology behind this thesis involved a vast amount of research, careful planning and design of software extensions, and a complex, challenging implementation process. The preliminary work consisted of investigating the general state of security and key management in SDN to discover areas lacking in research. The author set out to find limitations in control/data plane security and learn if key management has any relevance to SDN. The original title *Secure Key Management in Software Defined Networks* was chosen to allow the general concept of improving SDN security through key management to be explored. A master's thesis by Braadland (2017) was found that conceptualised a promising and inventive approach to enhancing SDN security but never developed a proof of concept (2.4) [Bra17]. This led to the goal of building on the research by producing a working solution to data plane key management called DPKM.

Various tools and technologies were analysed for developing DPKM, deciding on the technologies presented in section 2.3, with a Floodlight UI/controller, Open vSwitch switch, and WireGuard tunnelling protocol. Three virtual machines were set up (5.1) – one controller and two nodes – determining if a basic working SDN could be simulated and if nodes could be manually connected via WireGuard. Testing the technologies showed that the project was feasible, and development could begin.

Requirements and use cases were formulated by studying key management literature and existing protocols. The project timeline was planned, breaking development into phases of an iterative and incremental SDLC. Designs were drawn up for the architectural components and connecting interfaces based on the chosen technologies, project specification document, and background research. Eight procedures were proposed to deliver the key management functionality, with careful consideration of system state and communication between SDN components. Blueprints of classes, methods, and OpenFlow extensions were created to guide the implementation.

Implementation began with adding a new module to the controller software and generating the DPKM OpenFlow extensions. Simultaneous modifications to the UI, controller, switch, and connecting interfaces were made to incorporate each procedure successively, starting with the functionality for configuring the WireGuard interface in a node. Four GitHub repositories were used, adding approximately 7,500 lines of code to deliver DPKM. Testing was performed throughout iterations of the SDLC and improvements were made based on discovered issues.

1.5 Outline

Chapter 2 – *Background Research* – builds a base understanding of the field of study and research undertaken to formulate this thesis. Begins with the emergence and benefits of SDN (2.1) and the relationship between cryptography and key management (2.2). Follows with an analysis of the available tools and technologies for developing DPKM, including OpenFlow, WireGuard, Open vSwitch, Floodlight, and UI (2.3). Ends with a review of the related work and its influence on this thesis (2.4).

Chapter 3 – *Planning and Specification* – outlines the project specification and planning that preceded the design and implementation. Begins with a complete description of the requirements (3.1) and formulation of several use cases (3.2). Follows with an investigation of SDLCs and how to structure phases of development (3.3). Ends with a comparison between planned project timelines and the actual timeline (3.4).

Chapter 4 – *Designing DPKM* – presents the full architectural and functional designs conceptualising the DPKM system from the previous chapters and guiding implementation in the subsequent chapter. Begins with the structural extensions and expected responsibilities of the architectural components, including the controller, node, application, and interfaces (4.1). Follows with a proposal of the underlying sequences of functionality within DPKM procedures (4.2). Ends with a description of OpenFlow experimenter message extensions (4.3).

Chapter 5 – *Implementing DPKM* – documents the process of transforming the designs conceptualising DPKM into a functional product. Begins with a condensed explanation of the system setup and development environment (5.1). Follows with an overview of the extensions to the OpenFlow protocol (5.2) and software components (5.3). Ends with a detailed discussion of the code implementing and operating DPKM procedures (5.4) and a summary of the testing methodology (5.5).

Chapter 6 – *Evaluation and Discussion* – evaluates the implemented system and discusses the project in its entirety. Begins with an analysis of DPKM in satisfying the requirements (6.1) and discussion of the risks encountered during development (6.2). Follows with an examination of the test results (6.3), achievements (6.4), and limitations of the delivered product (6.5). Ends with a proposal of possible future developments on DPKM (6.6).

Chapter 7 – *Conclusion* – summarises the full content of the thesis and offers some final points for the reader to take away.

Chapter 2

Background Research

This chapter provides insight into the theoretical background of the subject area and explores the wide range of literary sources that contributed to the formulation of this project. Extensive research into SDN technologies, cryptography, key management, and the state-of-the-art, highlighted opportunities for contributing to the field of study and progressing the knowledge domain. This chapter begins with section 2.1, building a base understanding of the emergence, structure, and benefits of Software Defined Networking (SDN). Following this, section 2.2 investigates the relationship between cryptography and key management, contrasting different approaches, architectures, and protocols, to envision the requirements of a new key management scheme. Thereafter, section 2.3 analyses a range of tools and technologies available for developing a software solution to data plane encryption. Finally, section 2.4 surveys the state-of-the-art and studies related work to discover areas of unfulfilled potential.

2.1 Software Defined Networking

The Software Defined Networking (SDN) architecture presents a software-based approach to network management that offers more flexibility than traditional hardware centric architectures. SDN enables centralized network control and programmability through its separation of the control plane and data plane. SDN transforms data plane network devices such as switches and routers into simple packet forwarding devices, allowing the controller to dictate the control logic and network behaviour [KF13]. Network activities are segregated, meaning data plane devices only have the forwarding responsibilities in their integrated hardware tables, giving centralized control to the controller which has knowledge of the full network topology and can

therefore optimize performance [Gor14]. Furthermore, the controller utilises software to program hardware on the data plane devices, allowing changes to the forwarding logic to be made flexibly and efficiently. Applications running on the controller make changes to the network by submitting requirements through the northbound API. These requirements are translated by the controller using its control logic and applied by programming the data plane devices through the southbound API.

The northbound API provides an abstraction of the entire network topology to the applications and enables real-time network management without the need for an in depth understanding of the internal logic. There is no universal northbound API due to the varying forms of SDN and differences in the design of the applications [Jar+14]. Popular approaches to the northbound API include REST, SDN programming languages such as Frenetic or Procera, or APIs designed for a specific controller such as SDMN API (MobileFlow) and PANE API (PANE) [TV16]. The southbound API is the key component for externalising the control logic of the data plane. It provides programmability to support new policies, abstraction of and shared access to physical resources, and isolated flows of traffic within the network [SNK12]. Unlike the northbound API, OpenFlow is still seen as a universal standard for southbound APIs due to its pinnacle role in the birth of SDN. However, use of traditional OpenFlow as a southbound API is less commonplace due to the benefits of having more intelligent data plane devices forming the basis of customised southbound APIs. Furthermore, a move toward distributed or hybrid SDN architectures has created the need for expansions of or alternatives to OpenFlow.

The topology of a typical SDN architecture can be seen in figure 2.1 which depicts the connections between the discussed network components. In this example each application makes a change to the flow through the northbound API, which is translated and applied by the controller through the OpenFlow southbound API, dictating the flow between the network nodes.

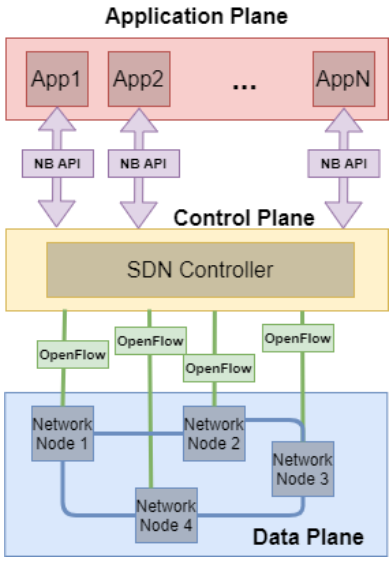

Figure 2.1: Topological model of a typical SDN architecture.

The benefits of SDN over traditional approaches to network management are illustrated in figures 2.2 and 2.3. A traditional network is composed of many nodes each equipped with ports to receive and forward packets, and a forwarding table assembled by the control plane establishing rules for packet flow [Gos17]. Figure 2.2 depicts two of such nodes, wherein each respective control plane has been individually configured to adhere to a forwarding policy implemented by the administrator or applications. The static nature of this approach means if changes to the policy are required then the administrator would have to manually adjust the configuration of each node. For larger networks this would be a time-consuming process that is restrictive to network scalability in response to changes in demand, reducing the efficiency of the network and presenting difficulty in making minor changes [Sez+13].

Figure 2.3 depicts the same two nodes but structured as an SDN, wherein each node has separated the control plane from the data plane, sharing a central controller. The dynamic nature of this approach means if the same changes to the policy were required then the administrator could simply programmatically configure both nodes from the same location through the southbound APIs. Furthermore, the controller now has shared access to both resources and a view of the complete network topology, so can adjust policies based on required demand for optimised efficiency.

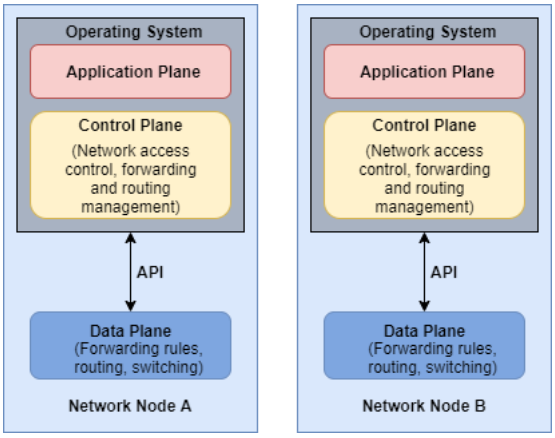

Figure 2.2: Two networking nodes structured in traditional hardware-centric architecture.

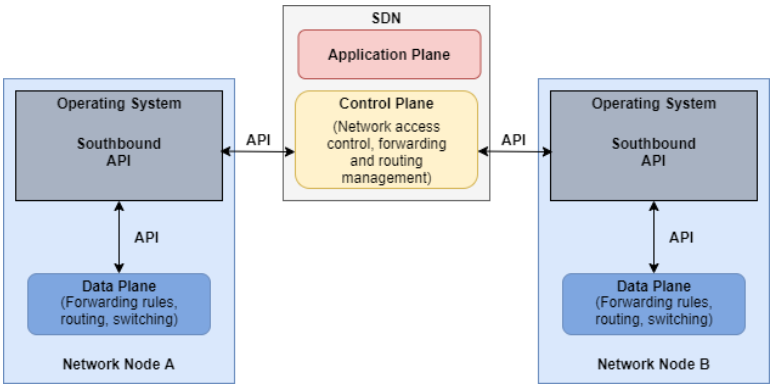

Figure 2.3: Two networking nodes structured as an SDN by separating control and data planes.

The origins of the SDN architecture can be traced to the development of the OpenFlow protocol at Stanford University in 2008. McKeown et al. (2008) were innovational in their proposal of an open protocol that enables an administrator to program the flow tables in network nodes from a controller with the ability to modify virtualized OpenFlow switches [McK+08]. Early work such as the *"Procera Controller"* played a part in SDNs rise to popularity by reporting significant reductions in the complexity of network management when in use on a campus network and several home networks [KF13]. SDN was also shown to be extendable to serve different purposes such as in Internet of Things [Qin+14], traffic anomaly detection [MKK11], improving vehicular adhoc networks [TLG15], and wireless sensor networks [OCF15]. This research was a key supporting factor in the viability of this project as SDN has been shown to be extensible for a multitude of purposes whilst providing its core benefits.

SDN-based architectures are popular in industry with organisations such as Google using SDN to structure their data centres in 2013 [Jai+13], and Facebook, Google, Microsoft, Yahoo, Deutsche Telekom, and Verizon funding the Open Networking Foundation (ONF) [ONF20a] in 2011 with the goal of developing open standards to encourage SDN adoption [Kre+15]. The industrial benefits of SDN are still apparent with Tins and Rabelo Oliveira (2017) and Ma et al. (2017) utilising SDN for energy and manufacturing efficiency in what is commonly referred to as industry 4.0 - the latest phase of the industrial revolution [LR17; MCC17]. This reinforces the motivation that enhancements to SDN have relevance and real-world applicability.

2.2 Key Management and Cryptography

Cryptography is the age-old term for the protection of information from access by an unauthorized third-party [DH76]. Humans are engaged in the endless pursuit for unbreakable cryptosystems from handwritten ciphers before the digital age to modern day developments in quantum cryptography. Cryptography and key management are natural companions, as information is protected through encryption by combining a message with a secret key using an algorithm protected by computational complexity and randomness [Zbi+98]. Cryptographic keys are required to have a lifecycle to reinforce a systems security and come in different forms based on the cryptosystem and algorithms. Key management protocols are responsible for carrying out the phases of the key's lifecycle, transitioning between several states from generation to destruction [Bar+13].

This section begins with subsection 2.2.1, exploring the classic cryptographic approaches to encryption. Following this, subsection 2.2.2 illustrates the phases involved in administering the key management life cycle and subsection 2.2.3 investigates key management in systems with the presence of a third-party. Finally, subsection 2.2.4 reviews existing key management protocols and relates them to DPKM.

2.2.1 Cryptographic Approaches

Prior to the design of a key management protocol, it is essential to consider the cryptographic approach to encryption that will be taken and properties of the keys the system will use. Barker et al. (2009) categorize cryptographic keys into twenty types based on their properties [Bar+09]. Key type is categorized in three forms: 1) public, 2) private, and 3) symmetric. Public and private keys come as a pair and are used in asymmetric or public-key cryptogra-

phy. Whereas, symmetric keys are used individually in symmetric key cryptography. These approaches facilitate encryption in the following ways:

Asymmetric Approach

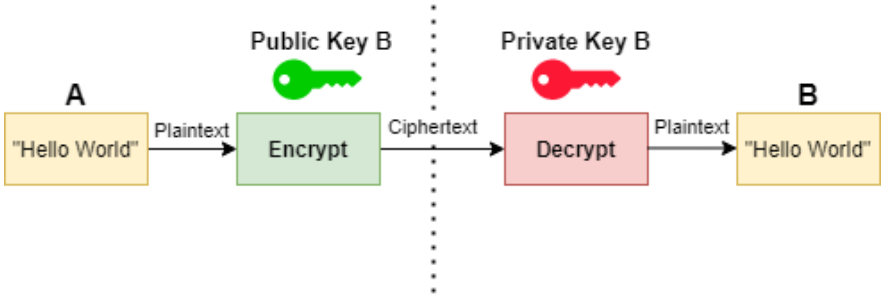

Figure 2.4: Example of a public-key or asymmetric cryptographic approach to encryption, using the public key for encryption and private key for decryption.

The asymmetric or public-key cryptographic approach uses public keys for encryption and private keys for decryption. In an asymmetric cryptosystem each party has its own public and private key. Public keys are readily available and distributed to all other parties in the system by the key management entity whereas private keys are kept secret, known only to a single party. Using figure 2.4 as an example: A has a message to send to B of which the content should be kept secret. A uses B's public key and some encryption algorithm to generate a ciphertext, sending the encrypted message to B. B then runs a decryption algorithm with the private key to get the plaintext message. Taking an asymmetric approach presents the benefits of easier key distribution and greater security since private keys are only known to one party. However, asymmetric cryptography comes at a performance cost when compared to symmetric approaches due to the computational complexity of the encryption algorithms.

Symmetric Approach

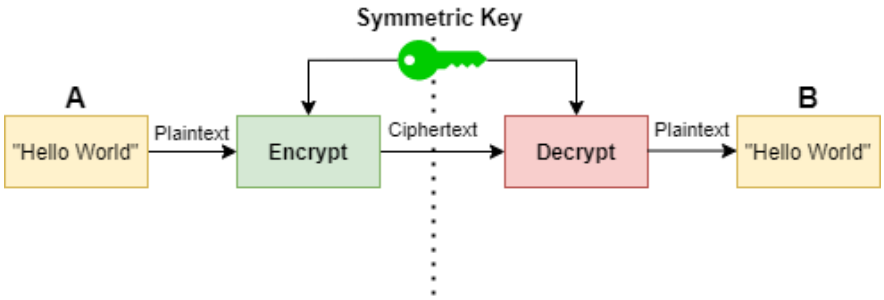

Figure 2.5: Example of a symmetric cryptographic approach to encryption using the symmetric key for both encryption and decryption.

The symmetric key cryptographic approach uses the same key for both encryption and decryption. In a symmetric cryptosystem each party has a copy of the same secret key which has been shared through some secure channel prior to any communication. Figure 2.5 illustrates this concept where A encrypts a message and sends it to B who then decrypts the message with the same key. Taking a symmetric approach to encryption presents the benefits of greater performance and strong security against interception if the key is kept secret. However, key distribution is difficult and a compromised key causes greater damage since data belonging to all parties can be decrypted.

Lifespan

The lifespan of cryptographic keys can be split into two subcategories: 1) static or long-term keys and 2) ephemeral or short-term keys. Static keys are created to be replaced infrequently and have a lifespan of years before they are replaced by the key management entity. All three key types can be static keys, but generally symmetric keys have a shorter recommended lifespan [Gir20]. Ephemeral keys are created specifically for a single session or instance of a key establishment transaction, after which they expire and cannot be used [Bar+13]. Ephemeral keys such as session keys are symmetric since the same key is used to authenticate the session and are used in conjunction with both symmetric and asymmetric long-term keys.

2.2.2 Key Management Lifecycle

The lifecycle of a cryptographic key can be divided into phases involving transitions between states from creation to destruction. Although the number of states and transitions can vary, all key management protocols share commonalities due to the universal responsibility of administering the lifecycle of a systems cryptographic keys. Barker (2020) divides the key management lifecycle into four main phases: Pre-operational, Operational, Post-operational, and Destroyed, in their most recent publication *"Recommendation for Key Management: Part 1 – General"* [Bar20]. Within these phases the key management entity performs certain functions to change the state of the key. This subsection structures key management by these four phases, investigating the functions and state transitions between them.

Pre-operational Phase

The pre-operational phase is the beginning of a cryptographic key's lifecycle. Prior to this phase a key has not been generated or activated and is not available for use in the cryptographic operations of the system. In this phase the functions of key generation, distribution, and exchange are performed.

Key Generation is the process in which a key is generated by a random or pseudorandom bit generator (RBG/PRBG). Randomness is crucial to the security of the cryptosystem as it must be unfeasible to deduce the key from analysis of the generation algorithm or an exhaustive search for the key [Mau92]. Cryptosystems based on symmetric cryptography generate keys directly from the RBG meaning the generation process is fast with low computational complexity. Cryptosystems based on asymmetric cryptography have a complex generation process that requires use of both an RBG and a generation algorithm that produces key pairs specifically designed for the asymmetric algorithm [BR12].

Key Distribution is the delivery of keys and associated key material to the network entities. The distribution process is easy for asymmetric cryptosystems as the public key can be delivered to all entities with the public key protected by the secrecy of the private key. Unlike public key distribution, the difficulty of symmetric key distribution is renowned and commonly referred to as the *"Key Distribution Problem"* due to its security relying on the secrecy of the key itself [MI88]. Approaches to symmetric key distribution include manual delivery of the key, use of a trusted third party (2.2.3), delivery through a secure channel, and pre-distribution hashing/encryption of the key.

Key Exchange is the establishment of a shared key or pair of keys between two or more communicating entities to authenticate the communication session. There are two main techniques for performing key exchange: 1) Key agreement and 2) Key transport. Key agreement is the use of an agreement protocol such as Diffie-Hellman key agreement by the communicating entities to generate a key value from an equal contribution of both [Jus05]. Key transport is the creation and distribution of a shared key by a single entity.

Operational Phase

The operational phase is the period of the management lifecycle where the cryptographic keys are in use within the system. Differing functions are performed based on the architecture of the cryptosystem. For brevity, the functions of key activation, key storage, and rekeying have been chosen as they apply to most lifecycles.

Key Activation is the initialization of the keys for use in the cryptosystem by the network entities. Leading up to this stage the key management protocol has generated the keys and any required properties, ensuring all material is in possession of the correct network entities to begin encrypted communication. Activation is performed during the transition from the pre-operational to the operational phase.

Key Storage is the temporary or permanent storage of key properties for immediate or later use by the key management entity. Storage should have high availability and strong tamper resistance to prevent compromise of the systems performance and security. Barker (2020) describes storage as being essential in assisting in normal operations and recoverability of lost or corrupted key material [Bar20].

Rekeying is the replacement of an expired or compromised key with a newly generated key. Rekeying plays a critical role in maintaining the security of a system and ensuring an attacker has insufficient time to break an existing key. Li et al. (2016) describe rekeying as providing the security benefits of key protection, through replacement after expiry or compromise, and access revocation through replacement of keys that are still in use [Li+16]. The recommended lifespan of a key (discussed in 2.2.1) is used by the administrator to set the rekeying period.

Post-operational Phase

The post-operational phase is the period of the management lifecycle where the cryptographic keys are not operational within the system. In this phase key material may be accessed but only

to recover, revoke or destroy a key.

Key Recovery is the retrieval of an expired key from an archive or reconstruction from archived information. To allow for recovery the key management entity must archive keys during the rekeying process. The purpose of this function is to recover information that has been encrypted by an expired key and is only used in unique instances where access to encrypted information is necessary, such as assisting law enforcement [Gen+99].

Key Revocation is the blacklisting or removal of a key in the system. Key revocation occurs as a result of a change in an entities authorization, a direct compromise of the key, or compromise of a connected entity in the system [ZR00]. Revoked keys are either added to a blacklist or deleted from the system. All entities that shared the compromised key (symmetric) or used a public key paired with a compromised private key (asymmetric) are notified that the key is no longer valid.

Key Destruction is the deletion of all copies of a key to prevent future use within the cryptosystem. Fully erasing cryptographic keys is challenging due to remaining residual traces in an entity's physical memory. However, erasable memory techniques such as those detailed in the paper *"How To Forget a Secret"* can be employed to transform the non-erasable memory containing symmetric and private keys into erasable memory [Di +99].

Destroyed Phase

The destroyed phase is the final phase of the key management lifecycle where records of the key's existence are destroyed completely or contain metadata to illustrate the history of the system.

Phase Transitions

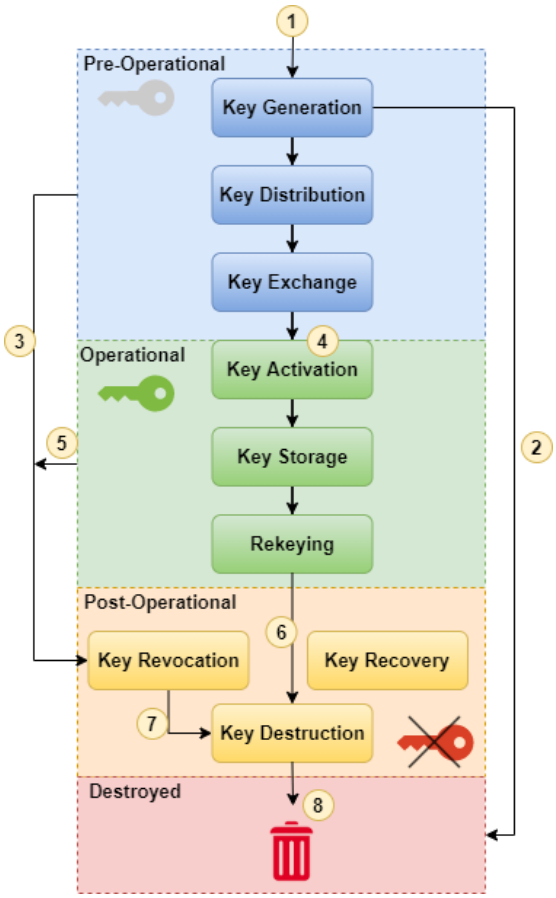

Figure 2.6: Flow diagram depicting the transitions between the four phases and inner functions of the key management lifecycle.

Figure 2.6 is a flow diagram depicting the transitions between the four phases and inner functions of the key management lifecycle. The eight labelled transitions are as follows:

1. A cryptographic key is generated entering the pre-operational phase.
2. The generated key is not used in the pre-operation phase so gets destroyed by transitioning to the destroyed phase.
3. The generated key transitions to the post-operation phase to perform revocation because it has been compromised during key distribution, key exchange, or another pre-operational function.

4. All the required key material has been established meaning the key can now be activated, transitioning from the pre-operational phase to the operational phase for use by entities in the cryptosystem.
5. The active key transitions to the post-operation phase to perform revocation because it has been compromised during operation or an operational function.
6. The lifespan of the key has come to an end meaning the key is no longer valid. The key transitions from the operational phase to the post-operational phase to perform the destruction of the key and any records.
7. Records of a revoked key must be destroyed transitioning from the key revocation function to the key destruction function within the post-operational phase (and subsequently to the destroyed phase).
8. Records of post-operational key are destroyed, transitioning from the post-operational phase to the destroyed phase.

2.2.3 Third-Party Systems

Efficient administration of the key management lifecycle is often achieved through incorporation of a Trusted Third Party (TTP) into the cryptosystem's architecture. Performing management functions, particularly key distribution/exchange, becomes far less complex and computationally intensive if there is a mediator that can safely exchange information with the systems entities. Involvement of a TTP relates to the design and implementation of a key management protocol for SDN since the controller is essentially a TTP between network nodes. This subsection investigates TTP components in symmetric and asymmetric cryptosystems to gain an understanding of any additional roles expected of the DPKM controller.

Symmetric TTP

As discussed in the previous section, key distribution poses a challenge in symmetric key cryptosystems. One solution to this is using a third-party component called a Key Distribution Center (KDC). A KDC is a TTP that generates and distributes encrypted secret keys to all members of a group through a secure channel if the requesting entity can be proven to legitimately belong to the group [DAr01]. The Kerberos protocol is one of the most famous examples of integrating a KDC for symmetric key distribution, using a ticket validation system to authenticate client requests for secret keys [Mil+88]. Much like KDC, a Key Translation Center (KTC) is a TTP that attempts to solve the key distribution problem with a different

approach. Rather than generating and distributing a key to all entities of a group the KTC expects the entities to generate their own keys and acts as a relay between them. The KTC receives an encrypted key from an entity on one channel and proceeds to decrypt it. Following this, a shared key between the KTC and destination entity is used to encrypt the key again and forward it on another channel to the destination entity [Fum98].

Asymmetric TTP

In asymmetric or public-key cryptosystems, a Public Key Authority (PKA) is a TTP that assists in the distribution and maintenance of public keys. Instead of openly broadcasting public keys, entities in the system register keys with the PKA, adding them to a shared directory which is used by the PKA to return public keys upon request by registered entities [AS10]. Using a PKA and public key directory enhances system security since only registered entities can request public keys, reducing the risk of forged public keys being circulated by unregistered entities. However, for larger systems with many keys, PKAs can reduce efficiency and cause synchronicity issues due to the demand placed on the PKA.

To better handle key registration and distribution in asymmetric systems; digital certificates can be used to bind key ownership and authenticity to the identity of an entity. A Public Key Infrastructure (PKI) is a set of policies, services, and protocols that combine asymmetric cryptography with key and digital certificate management to support the authentication and secure exchange of credentials between network entities [Wei01]. PKIs use a Certification Authority (CA) to authenticate key ownership by validating certificates against identity and adding the CAs own signature as a universally recognised seal of approval [TJ01]. By validating identity with the CA, network entities can maintain a mutual level of trust as their digital certificates have been generated and signed in accordance to the PKIs policies or global standards. In conjunction with the CA, PKIs use a registration authority (RA) for verifying and forwarding certificate requests from the entities to the CA, a certificate repository for storage, and a management system for issuing the digital certificate lifecycle and policies [AL99].

2.2.4 Key Management Protocol Review

To form a better understanding of what the design and implementation of this project entails it was important to review existing key management protocols before development. Inspiration can be drawn from studying the architecture and components of key management protocols used in industry. This subsection selects several key management protocols from a wide range

of literary sources to build perspective and influence the design of DPKM (chapter 4).

SSH, SSL and TLS

Secure Shell (SSH), Secure Sockets Layer (SSL), and Transport Layer Security (TLS) are network protocols that perform asymmetric encryption based on PKI and digital certificates. SSH is designed for client/server architectures wherein clients make requests to execute commands or transfer files and servers accept or reject the connections based on authenticity of the client and integrity of the data [YL+06]. In an SSH architecture each client and server have an asymmetric key pair, using the public key for proof of identity, and a symmetric session key for encrypting communication between them. The server has two types of public keys – host keys and server keys. The host key is static and used to identify the server whereas the server key is ephemeral and used to encrypt the session key with a lifetime of approximately one hour [Bar+01]. The session key is a symmetric key that is shared by two parties and destroyed after the session.

Key management for SSH can become a complex process for larger networks where many entities need fast access and authorization. Arkhipkin et al. (2008) attempted to address this problem through their proposal of an SSH key management system called *"STAR"* [Ark+08]. This paper presented several design concepts for key management systems such as a web-based user interface that can track cryptographic keys, users, and system status. A more recent publication by Harchol et al. (2018) provided further inspiration for DPKM with their proposal of a logically centralized SSH key manager called *"ESKM"* [HAP18]. Much like SDN, ESKM separates the control plane and data plane, with the control plane managing and storing private keys as well as logging and performing other key management functions. Both these papers have contributed valuable insight into designing the user interface, database, and functionality of DPKM.

Despite being a client/server architecture and using a PKI/digital certificates, SSL and its successor TLS have a different use case to SSH. SSH is primarily for command execution whereas SSL/TLS is designed for secure data transfer. Furthermore, SSL/TLS only authenticates the server unless specifically instructed otherwise, meaning the client can communicate without authentication [FKK11]. SSL was upgraded to TLS due to security flaws such as MITM attacks and performance issues causing slow connections between clients and servers. In TLS, identity of clients and servers are authenticated using asymmetric cryptography and

connections are protected through symmetric cryptography. The process of starting communication is called a handshake and is initiated by the client. The client transmits a hello message containing all the symmetric key algorithms it supports, followed by a response from the server containing the preferred algorithm to use. The digital certificate bound to the server is then signed by a CA and distributed to the client, authenticating the server [Cho02; Opp16]. The client and server can now use asymmetric cryptography to establish session keys and an encrypted communication channel. TLS/SSL is used universally in HTTPS to authenticate webpages, highlighting the success of using digital certificates and PKI key management.

Although the SDN architecture differs from the client/server model, digital certificates can enhance security in a similar fashion. Toseef et al. (2014) proposed *"C-BAS"* as an authorization mechanism for SDN experimentation facilities based on digital certificates [Tos+14]. C-BAS enhances security at application level, policing users of shared resources through permissions and credentials linked to users' digital certificates. Chowdhary et al. (2019) also utilise digital certificates in SDN with a novel framework called *"TRUFL"* using a root CA in the controller and multiple distributed CAs in the network nodes [Cho+19]. Whilst the use of digital certificates is beyond the scope of this project, the literature illustrates an interesting approach to enhancing security and shows promise for future work on DPKM (6.6).

IPsec and IKE/IKEv2

Davis (2001) defines IP security (IPsec) as a *"suite of protocols that seamlessly integrate security into IP"* with the purpose of providing *"end-to-end"* encryption over the IP layer [Dav01]. IPsec can be utilised to protect protocols such as TCP, UDP, and ICMP and build Virtual Private Networks (VPNs) to connect network entities through an encrypted tunnel. IPsec is composed of the two security protocols: Authentication Headers (AH) and Encapsulating Security Payloads (ESP); components called Security Associations (SA), and the key management protocol Internet Key Exchange (IKE). IPsec and IKE are one of the main inspirations for both the development of this project and Braadland's thesis [Bra17] which proposed the idea this project is built upon (see 2.4). For this reason, IPSec and IKE will be covered in depth and referenced on multiple occasions in this thesis.

In IPsec the AH provides data integrity, authentication of entities and protection against replay attacks [Elk+02]. The ESP provides the same features as AH in addition to confidentiality through encryption. SAs are data structures describing IPsec connections that offer security

services to the network traffic and contain shared information such as cryptographic keys and algorithms [Opp98; BIK02]. When handling a packet, the SA will either supply IPSec security services, drop the packet, or allow IPsec protocols to be bypassed. IKE handles key management in IPsec and is responsible for both key distribution and establishment of SAs from network information. The original version of IKE suffered from many issues such as its unnecessarily high complexity impacting performance, and vulnerability to a range of attacks including impersonation attacks, authentication bypasses, dictionary attacks against pre-shared keys, and other methods to break the encryption [PK00; Fel+18; Mea99]. IKE was therefore upgraded to IKEv2 in an effort to simplify key management and improve security. Communication with IKEv2 follows a request/response pattern meaning all messages come in pairs.

Figure 2.7 is a sequence diagram depicting the message exchanges carried out during the operation of IKEv2. All `IKE_SA_INIT` exchanges must be completed first, followed by all `IKE_AUTH` exchanges, after which `CREATE_CHILD_SA` and `INFORMATIONAL` exchanges can occur in no order [Kau+05].

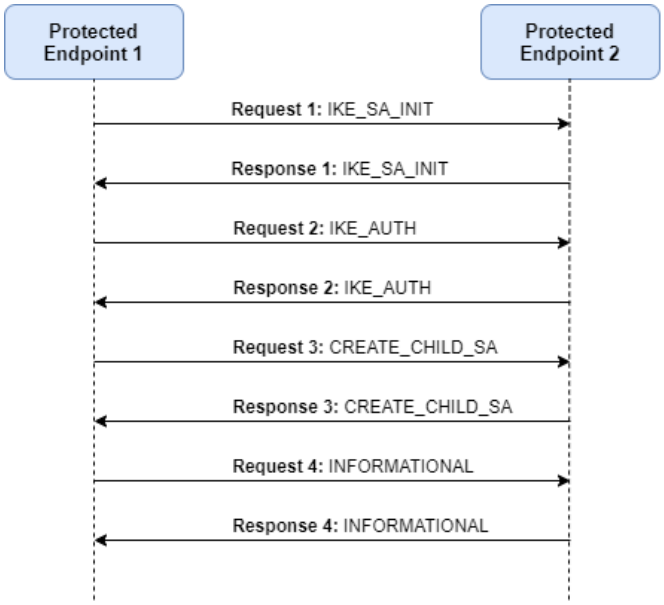

Figure 2.7: Sequence diagram illustrating the message exchanges of the key management protocol IKEv2.

1. **IKE_SA_INIT:** The conditions for a new SA are established for the connection between the endpoints by negotiating security parameters such as cryptographic algorithms, trans-

ferring nonce values, and performing an exchange of Diffie-Hellman (DH) handshake values.

2. **IKE_AUTH:** The SA created by the `IKE_SA_INIT` message exchange is authenticated by exchanging the identity information and digital certificates bound to the endpoints. Shared secret material between the endpoints is verified against their identity and DH key exchange is completed with the `IKE_SA_INIT` payload values. Finally, `IKE_AUTH` establishes the first `CHILD_SA` which is either type AH or ESP. IKEv2 authentication is symmetric in its use of pre-shared keys, asymmetric in its encryption, and reinforced by digital signatures [Kau+05; Cre11].
3. **CREATE_CHILD_SA:** Additional `CHILD_SAs` are created for the original SA between the endpoints using this message exchange or the `CHILD_SA` created by the `IKE_AUTH` exchange is rekeyed.
4. **INFORMATIONAL:** Control messages containing payloads with functions or information such as errors, instructions to remove a SA, and notifications regarding events.

KINK

Although IKE is the most popular protocol for setting up IPsec SAs the *Kerberized Internet Negotiation of Keys* (KINK) protocol can be used as an alternative approach to IPsec key management. KINK provides centralized key management with a TTP (KDC) and authentication based on the Kerberos protocol (2.2.3) [Mil+88]. KINK operates in a command/response pattern that creates, deletes or modifies IPsec SAs. Unlike IKE, KINK is stateless meaning storage of a hard state is not necessary after every command and response [Sak+06]. Furthermore, use of the Kerberos protocol means encryption is primarily delivered through symmetric cryptography using shared secret keys between the endpoints and KDC. This reduces complexity and computational cost as less memory is required than stateful protocols and KINKs symmetric algorithms outperform IKEs asymmetric algorithms. However, KINK is vulnerable to attacks targeting the KDC server since it is a single point of failure that if compromised would enable an attacker to impersonate any device on the network. This means managing a KINK system requires greater cost and maintenance as security of the KDC must be impenetrable. Overall, IKE is the primary key management choice for IPsec due to its flexibility, limited susceptibility to compromise, and range of authentication mechanisms. IKE was therefore one of the primary reference points in the design of this project.

2.3 Technology Choices

One of the most important aspects of background research involves investigating the tools and technologies available to develop the project. Various technologies were analysed to assess suitability and contrast strengths/weaknesses. Although technology choices traditionally come under the planning and specification stages of a project, the author felt the section is better fitting in the background research chapter because of the depth of study. This section begins with subsection 2.3.1, exploring the characteristics and inner workings of the OpenFlow protocol. Following this, subsection 2.3.2 analyses the tunnelling protocol WireGuard and compares it with alternative VPN solutions. Finally, the technologies involved in constructing an SDN architecture are dissected layer by layer covering the data plane nodes (2.3.3), controller (2.3.4), and application plane (2.3.5).

2.3.1 OpenFlow

OpenFlow is an open communications protocol that precedes the creation of SDN and enables network control to be moved from data plane switches to a logically centralized controller [Fou12]. The protocols widely documented popularity and extensibility instantly made it the main contender for use as the southbound API in this project. OpenFlow is essentially comprised of three components: a virtualized switch compatible with OpenFlow, a controller, and the secure channel connecting them. OpenFlow defines the standards and message format for communication between the controller and switch, acts as a specification for the functionality expected of an OpenFlow switch, and can even be interpreted as a network architecture itself as a subset of SDN [LKR14].

Communication between the controller and switch involves the exchange of a set of messages defined by the OpenFlow specification. OpenFlow is built upon the Transmission Control Protocol (TCP) to enforce reliable message delivery, with the controller listening on TCP port 6653, and typically uses TLS to asymmetrically encrypt the channel [Azo13]. In most circumstances, the OpenFlow channel is created as a result of the switch initiating a TCP or TLS connection to the controllers IP address and transport port [Spe13]. However, the switch can also be configured to accept incoming connections by setting up a transport port, enabling the controller to be the initiator. A single OpenFlow message can store up to 64KB of data and OpenFlow provides multipart message functionality where data larger than 64KB can be spread across multiple messages [Zur+17]. OpenFlow messages can be divided into three cat-

egories: controller-to-switch, asynchronous, and symmetric, each with multiple subcategories. Controller-to-switch messages are transmitted by the controller to manage or extract the state of a switch and do not always require a response. Asynchronous messages are transmitted by the switch to inform the controller of events such as a packet arriving or the state of the switch changing. Symmetric messages can be sent in both directions for setting up, reporting errors or additional functionality.

Before planning the design of DPKM it was important to understand which subcategories of OpenFlow messages the project could utilise. From the OpenFlow Switch Specification version 1.4 [ONF13] it was deduced that the following messages were particularly relevant:

1. **Modify-State:** Controller-to-Switch messages that manage the flow tables and other stateful properties of the switch. FlowMod modify-state messages were pertinent because they set the switches flow table rules for packet matching. This means they can enable routing of packets in DPKM through a port with a network tunnel to another switch. FlowMod messages contain a match dictating the rules to accept/reject packets, a cookie, a port number to output the packet on, list of instructions with actions to perform on a received packet, and a command to add, modify, or delete the flow. FlowMod messages will be explored in more detail in the design and implementation chapters of this thesis.

2. **Packet-out:** Controller-to-Switch messages that contain a full packet or reference to a packet, a list of actions for the switch to apply, and a port in the switch to output the packet from. Packet-out messages are also used for the purpose of forwarding a Packet-in message received from a switch. Packet-out messages were considered as a potential option for sending messages in DPKM through a specific port with a network tunnel or playing a role in key management using custom actions.

3. **Packet-in:** Asynchronous messages that send a packet captured by the switch to the controller for processing. Packet-in messages are sent if a packet does not match the flow table rules, an action specifies that a packet should be sent to the controller, or there is a TTL error. Packet-in messages were investigated because the DPKM controller would have to process them in a different way if packet-out messages are used as part of the scheme.

4. **Experimenter:** Symmetric messages that can be extended to incorporate additional functionality into OpenFlow with new properties and fields. Experimenter messages showed promise for implementing the extensions needed for DPKM and were the main argument for using OpenFlow over other southbound APIs. Experimenter messages ultimately form the backbone to this project and are explored in depth in section 4.3 and throughout the implementation chapter of this thesis.

An OpenFlow switch is composed of one or multiple flows tables, a group table, the secure OpenFlow channel to the controller, packet processing pipelines between flow tables, and ports for sending and receiving packets [Gor14]. Flow tables contain flow entries with match fields to compare packets against, a priority value for deciding precedence of flow entries, counters for statistics, instructions to dictate packet processing, timeouts to determine the lifespan of the flow, and a cookie set by the controller [ONF13]. The packet processing pipeline handles incoming packets by attempting to match the first flow table on the highest priority entry and moving on to subsequent tables based on the instruction set. If the packet does not match the flow entries of a table then the packet is handled in accordance with the tables "table miss" logic. A missed packet is either dropped, sent to the controller as a packet-in message or passed to the next flow table. Ports can be physical ports representing a hardware device such as an ethernet interface or logical ports mapping to a virtual interface or some abstraction. For DPKM to operate with OpenFlow, custom flow entries need to be designed (section 4.2.6) and a dedicated logical port is required to support a network interface connecting to another node (section 4.1.4).

An OpenFlow controller can be configured to operate with a reactive or proactive flow control model [Azo13]. In a reactive control model, the switch sends a packet-in message to the controller whenever a decision regarding a missed packet needs to be made. The controller uses its view of the network topology to decide the route the packet can take to reach its destination and adds or modifies a flow entry in the switches flow table to update the forwarding logic. In a proactive control model, all flows are programmed by the controller before communication begins and any packets that do not match a flow entry are dropped or handled in accordance to the flows "table miss" logic.

2.3.2 WireGuard

One of the primary objectives for DPKM is to facilitate encrypted node-to-node communication in the data plane by implementing a network tunnel between the nodes. Before development it was important to choose a tunnelling protocol that offers strong security, high performance, and an implementation complexity suited to the developer's skill level. IPsec was explored in section 2.2.4 and has the potential to achieve node-to-node encryption in SDN. The set of protocols can be utilised to build a VPN of network nodes with IKE establishing the required key management functionality. IPsec is considered a cryptographic standard for IP communication and has well-documented usage in industry making it an appealing tool for implementing the network tunnel. IPsec is renowned for the strength of its security due to the large number of encryption algorithms IKE implements and there are generally no major vulnerabilities [Dav01]. However, performance analysis of IPsec shows usage can cause a substantial increase in CPU overhead due to additional processing and increased packet size [SM14; PM18]. Furthermore, IPsec has been described as *"too complex to be secure"* [FS99] due to the vast range of configuration options available and could be challenging to implement correctly by an inexperienced developer. It was therefore felt that a less complex alternative to IPSec should be chosen for developing DPKM.

OpenVPN is an open-source VPN proposed by James Yonan in 2001 [Fei06] that implements a network tunnel based on SSL/TLS encryption and key exchange. OpenVPN is designed to be an alternative to IPsec that is easier to configure and deploy whilst providing the same strength of security and a multitude of authentication mechanisms [CK15]. The comparative performance of IPsec and OpenVPN is well documented with studies showing IPsec outperforming OpenVPN. Pohl and Schotten (2017) report higher maximum throughput using IPsec and more efficient key exchange using IKEv2 [PS17]. Lackovic and Tomic (2017) and Kotuliak et al. (2011) report similar results reinforcing IPsec's superior performance due to higher average throughput for all tested encryption ciphers [LT17; KRT11]. On the other hand, all three studies reference IPsec's complexity and feel that OpenVPN's simple configuration makes it a more attractive network tunnelling solution despite having weaker performance. However, as with IPsec, progressions in technology mean newer and more lightweight solutions have become available.

Why WireGuard

WireGuard is a layer 3 network tunnel proposed by Jason Donenfeld in 2017 [Don17] with the

mission of replacing IPsec and OpenVPN with a simpler but high-speed approach, requiring under 4000 lines of code in comparison to 70,000+ (OpenVPN) [Pri20]. WireGuard sets up network tunnels between endpoints using asymmetric cryptography in a similar fashion to SSH (section 2.2.4). Each endpoint device has an interface that is pre-configured with a public/private key pair and connects to another endpoint device by adding the targets public key and IP address as a peer. WireGuard secures communication sessions with symmetric cryptography by generating a shared key from the public keys and using it to encrypt the session. Furthermore, WireGuard can be configured to use pre-shared symmetric keys between two endpoints for post-quantum resistance [DP18]. Donenfeld (2017) describes WireGuard as using *"state of the art cryptography"* with single round trip (RT) key exchange based on the Noise protocol and Curve25519 elliptic-curve Diffie-Hellman (ECDH) key agreement; ChaCha20 and Poly1305 for symmetric encryption and authentication; and BLAKE2s for hashing [Don17]. By preventing algorithm negotiation and offering no cipher agility, WireGuard is secured against the downgrade attacks possible in OpenVPN as only one set of cryptographically secure ciphers are used [DP18]. If a vulnerability is found in a version of a protocol it will need to be replaced for WireGuard to securely operate, meaning unlike IPsec or OpenVPN, WireGuard cannot be accidentally misconfigured by the administrator to use a depreciated cipher.

In terms of performance, Donenfeld (2017) benchmarked WireGuard alongside IPsec and OpenVPN and Mackey et al. (2020) compared WireGuard to OpenVPN [Don17; Mac+20]. Both studies reported WireGuard outperforming OpenVPN (and IPsec in [Don17]), with higher maximum throughput, more efficient key management with a lower overhead using the chosen ciphers, and stronger multithreading resulting in efficient CPU usage and better scalability [Don17; Mac+20]. Moreover, WireGuard's small code base not only improves performance but heightens security by minimising the attack surface and makes usage ideal for an individual or smaller development team. Overall research into WireGuard presented a multitude of advantages over IPsec and OpenVPN in performance, security, and complexity, making it the ideal choice for implementing the tunnel between network nodes in DPKM. The DPKM development team is comprised of a single developer meaning WireGuard's simplicity is the primary attraction for using the tunnelling protocol. To further inform the developer's decision to use WireGuard this subsection investigates the inner workings of the protocol, detailing key management and data transport, and ending with an analysis of WireGuard's limitations and applicability within the SDN architecture.

Key Management

The operation of WireGuard can be split into two phases: a key exchange/handshake phase and data transport phase [DP18]. To achieve optimal performance the key management functionality provided by WireGuard focuses on delivering the key exchange phase of the cryptographic key lifecycle, expecting the CPU intensive static key generation and distribution functions to be carried out by an external entity. Bernstein (2006) proposed Curve25519 as a highly efficient and secure ECDH function for cryptographic applications such as key agreement [Ber06]. The efficiency of Curve25519 meant adoption by WireGuard in its key exchange functionality for agreement of session keys from 32-byte static keys. Before operating WireGuard an external entity creates the static key pair bound to the interface by generating a random Curve25519 32-byte private key and using the Curve25519 function to compute a 32-byte public key from the private key. Following this, the external entity distributes the public keys and IP addresses to the appropriate endpoints to establish a connection between the WireGuard interfaces. A 32-byte shared symmetric key can then be calculated by using Curve25519 with one interfaces private key and another interfaces public key. WireGuard sends the Curve25519 symmetric keys in its 1-RT key exchange handshake which is then used to derive a pair of symmetric session keys using BLAKE2s hashing, producing one key for sending messages and one key for receiving messages [Don17].

Figure 2.8 illustrates WireGuard's full operation including the handshake which is based on the Noise protocol [Per16]. Node A begins by preparing a handshake initiation message, generating a Curve25519 symmetric key from private key A and public key B. Node A initiates the handshake by sending an initiation message containing public key A, the Curve25519 symmetric key, a timestamp, and several other parameters. Node B responds with the other half of the parameters (public key B, other Curve25519 symmetric key) completing the handshake. The pair of sending and receiving session keys are then derived by each node hashing the Curve25519 symmetric keys exchanged during the handshake with BLAKE2s [Aum+13]. Finally, encrypted data transport between the nodes can begin.

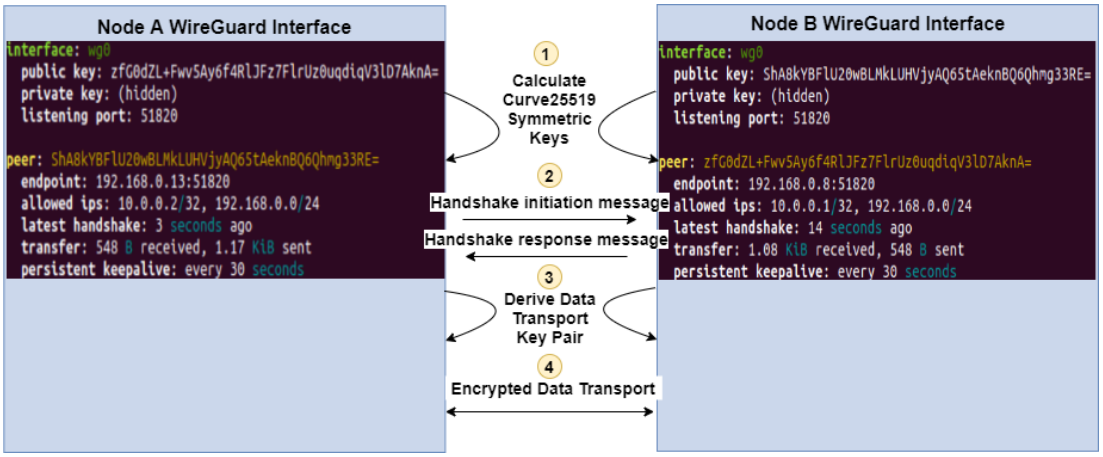

Figure 2.8: Diagram illustrating the WireGuard protocol setting up a network tunnel between two interfaces for encrypted data transport.

Data Transport

Once the key exchange handshake process has been completed packets can be encrypted/decrypted using ChaCha20Poly1305 [Lan+16] and sent over UDP between the endpoints. Data transport between WireGuard interfaces A and B can be summarised as follows [Don17; DP18]:

1. Plaintext packet arrives on interface A with a destination IP address matching peer interface B.
2. Interface A encrypts the packet with ChaCha20Poly1305, the symmetric sending key associated with peer B and a nonce counter.
3. The encrypted packet and a prepended header are encapsulated as a UDP packet and sent to peer B's endpoint IP address and UDP port.
4. The UDP/IP packet is received on the UDP port of B.
5. Interface B inspects the header to check if A is a registered peer, has a valid nonce, and tries to authenticate and decrypt the packet with the symmetric receiving key associated with peer A.
6. If the packet cannot be authenticated/decrypted or is discovered to be a non-IP packet once decrypted it is dropped. Otherwise interface B updates the endpoint of peer A and adds the plaintext packet to a receiving queue.

Limitations of WireGuard

To provide a fair assessment of WireGuard in the context of DPKM and SDN it is important to identify potential limitations of the protocol. Although WireGuard's rigid set of algorithms is seen as a cryptographic advantage by preventing downgrade attacks it presents a potential issue. The nodes in the DPKM architecture can only operate if WireGuard's ciphers are supported meaning there is no flexibility or support for nodes that cannot use one of WireGuard's ciphers. Furthermore, WireGuard is rendered insecure if a vulnerability is discovered in one of the ciphers. However, this limitation is unlikely to cause any issues for DPKM based on the technical assessment of WireGuard use in SDN [Bra17], explored in the related work section (2.4), and the extensive literature supporting the cryptographic strength of the chosen ciphers.

Another limitation is WireGuard has been designed as a purely layer 3 protocol meaning only IP packets can be routed through the network tunnel. This is problematic for SDN as the network nodes will be prevented from forwarding non-IP L2-L5 traffic. To overcome this issue there are three possible solutions. One solution is designing or modifying software in the network node to encapsulate non-IP packets as IP packets before passing them to the WireGuard interface. A second solution is routing all non-IP packets via the controller and exclusively sending IP packets through the network tunnel. A third and less ideal solution is dropping all non-IP packets and focusing on a purely L3 SDN architecture. A final limitation is little to no research exists for the practical use of WireGuard in SDN due to its relatively short existence (since 2017). However, although purely theoretical, Braadland's thesis (2017) gives the impression that WireGuard has strong potential use in SDN and supports the feasibility of DPKM (2.4) [Bra17].

2.3.3 Network Node

In an SDN architecture network nodes consist of virtualized switches for forwarding traffic that are connected to a controller through the southbound API. As discussed in 2.3.1, OpenFlow has been chosen as the southbound API due to its extensibility and potential to support the additional functionality expected of DPKM. It was therefore crucial to choose an OpenFlow compliant switch that can be modified to work with the extended OpenFlow protocol. Open vSwitch (OVS) is an open-source virtualized software switch that can be programmatically extended and controlled using OpenFlow [PD13]. OVS typically operates in both user space and kernel space but can be configured to function in user space only with the sandboxed

version. The main components of OVS are an ovs-vswitchd daemon for implementing and managing OVS switches and communicating with OpenFlow [Pfa+15], a lightweight database server called ovsdb-server, a set of tools for configuring, querying or updating kernel modules and OVS daemons, and a set of tools for testing and running commands against switches or controllers [OVS20].

Modifications to OVS are well documented with Chaignon et al. (2018) adding stateful packet filtering and monitoring functionality [Cha+18], Jackson et al. (2016) incorporating middlebox services [Jac+16], and Tu et al. (2017) proposing an extensible OVS kernel datapath [TSP17]. The multitude of research detailing OVS extensions with the OpenFlow experimenter structure makes OVS the primary choice for implementing the data plane nodes in DPKM. OVS was also used in the related work that formed the basis of this project (section 2.4) meaning use of OVS catalyses development since inspiration for DPKM can be taken from previous designs. Moreover, a wide range of documentation and tutorials are available on the official OVS website [OVS20] with an active community to assist in troubleshooting.

2.3.4 Controller

At the core of the SDN architecture is the controller providing management and configuration of the network nodes. There is no shortage of controller options with a multitude of software solutions for both OpenFlow based SDN and alternative architectures. Since the data plane technologies have been decided the main factors for choosing the controller are OpenFlow compatibility, support for OpenFlow experimenter extensions, an open source code base, and accessible documentation to assist with software modification.

Open Network Operating System (ONOS) is the most popular open source SDN controller and is endorsed by the ONF [ONF20b]. ONOS is designed for high performance, scalability for distributed SDN architectures and simplified network resource management [Ber+14]. Another popular open source controller is OpenDaylight which rivals ONOS as a solution to transform the SDN architecture into a distributed solution, providing several southbound API options including NETCONF/YANG, LISP and OVSDB, in addition to OpenFlow [Med+14]. Mamushiane et al. (2018) conducted a comparative performance and feature evaluation of the four controllers ONOS, OpenDaylight, Ryu, and Floodlight [MLD18]. OpenDaylight was discovered to be the most feature rich whereas ONOS reported the best overall throughput and performance. Furthermore, both ONOS and OpenDaylight have the most well-established use

in industry with a multitude of supporting partner companies. Although ONOS and OpenDaylight present significant advantages over other SDN controllers it is important to consider their suitability in relation to the scope of this project and scale of development. ONOS and OpenDaylight have been designed for larger scale deployments and distributed SDN architectures. Since the concept of DPKM is small-scale and designed for a traditional centralized SDN architecture, use of either controller could cause implementation to become unnecessarily complex and difficult for a single developer. Moreover, the lack of supporting documentation for use with the OpenFlow experimenter structure would lengthen the development timeline. Therefore, a centralized controller should be considered that is suited to smaller scale development.

Ryu is an open source controller built in the programming language python for traditional centralized small scale SDN deployments [Tom13]. Much like OpenDaylight, Ryu supports several southbound APIs including OpenFlow, NETCONF, OFCONFIG and OVSDB. Another open source controller with a similar use case to Ryu is Floodlight. Floodlight is a centralized java-based controller that is designed purely for SDN with the OpenFlow protocol [Flo20]. In the performance analysis by Mamushiane et al. (2018) Ryu displayed lower latency than Floodlight but worse throughput, with similar overall performance [MLD18]. Both Ryu and Floodlight presented themselves as worthy candidates for developing DPKM because of their suitability for small scale deployments. Although Ryu is more feature rich Floodlight was chosen for this project because of its superior documentation and supporting research proving that Floodlight can be extended more easily than alternative SDN solutions [Por+15; MMR15]. Unlike Ryu, Floodlight offers tutorials for adding OpenFlow experimenter extensions which can assist a new developer in implementing complex systems like DPKM. Moreover, Floodlight supports use of the tool Loxigen [Lox20] which allows OpenFlow protocol libraries to be generated and simplifies adding new messages and other OpenFlow objects needed for DPKM.

Overall, it would be naïve to disregard the fact that the Floodlight controller has numerous limitations in the full scope of SDN solutions. As the industry moves toward distributed and cloud based SDN architectures; centralized controllers such as Floodlight are rarely maintained. However, for small scale projects like DPKM it can be argued that Floodlight is the perfect starting point because of the supporting documentation and former popularity. Once the functionality of DPKM can be established and proven for centralized SDN an expansion to distributed architectures can be considered in future work (section 6.6).

2.3.5 Application Plane

The final technology choice to consider is the application plane component for interacting with and managing the entire system. Structuring the application plane is entirely dependent on the northbound interfaces and programming languages supported by the controller. The chosen controller Floodlight uses RESTful APIs for the northbound interfaces and provides a web-based GUI called Floodlight Web GUI built with HTML, CSS/Bootstrap and jQuery [AY17]. The Floodlight GUI is basic but can perform all expected SDN functionality through its REST API connections to the controller, allowing the administrator to interact with the network topology and view statistical information. The GUI is accessed using the IP address of the machine running the controller and port 8080 [Iza16]. Floodlight and the other network components must be extended substantially meaning a GUI is needed that can perform DPKM related functions. The two options are extending the current GUI or creating a new GUI specifically for DPKM. Since the primary focus of the project is on the data/control plane functionality it was decided that the current Floodlight GUI would be extended as creating a new GUI would increase the development timeline beyond the project deadline.

2.4 Related Work

Turning raw ideas or objectives into novel projects requires the researcher to gauge the state of the art of a topic area and discover potential gaps of unfulfilled potential. Prior to any specification or design the researcher surveyed research and literature related to security and key management in the SDN data plane. It was discovered that improvements to the security of the controller, southbound API, and data plane nodes has been widely considered. A multitude of research exists to provide solutions for securing southbound communication [Hu+15; Pan+17], or analysis and formulation of data plane security protocols [KKS13; Dar+17; Sha+20]. However, little research can be found for facilitating key management and encrypted communication over the connection between data plane nodes.

A single master's thesis on data plane encryption and key management was identified that addressed a similar problem envisioned for this project. Braadland (2017) proposed a key management scheme named *"DPKMS"* in their master's thesis titled *"Key Management for Data Plane Encryption in SDN Using WireGuard"* [Bra17]. The thesis detailed potential extensions to OpenFlow that would enable implementation of a new SDN key management scheme for initialising encrypted network tunnels between nodes using WireGuard as the tunnelling protocol.

DPKMS is highly novel as extending OpenFlow for key management had not been theorized and was the first to consider WireGuard use in SDN. Initial discovery of Braadland's thesis gave the impression that the chosen problem definition had been satisfied. However, on further inspection it was clear that the thesis only theorized DPKMS and did not produce a working implementation. In future work Braadland explains that a proof of concept is necessary to determine if DPKMS has any practical potential [Bra17]. Moreover, studying DPKMS made it apparent that it is far from being implementation ready, presenting many issues including:

- Poor design decisions creating security issues such as sending the private key from the controller to the switch.
- Misconceptions in how OpenFlow can be extended and no study of modifying the Floodlight and OVS code bases.
- Unnecessary or incorrect requests/responses in theoretical message sequences.
- Lack of understanding in how packet flow is setup and modified.
- Mistakes in most if not all blueprints for extensions.
- No design theory for application plane and REST APIs.

The inherent issues with the theory of DPKMS and lack of a proof of concept formed the motivation for this project. DPKM was created with the goal of extending, improving, and implementing a key management scheme using Braadland's thesis as a starting point. Although similar in theory, comparing the design decisions (chapter 4) and final implementation (chapter 5) of DPKM shows significant novelty and more differences than similarities to DPKMS. The functional DPKM system is more akin to a unique development than the system theorized in Braadland's thesis.

Birkeland (2016) produced a thesis titled *"Software Defined Data Flow Isolation by Virtualization and Cryptographic Key Distribution"* which is directly referenced by Braadland (2017) as the basis of their thesis [Bir16; Bra17]. Birkeland investigated data plane encryption and flow isolation in SDN by studying and testing various software solutions to determine the best feasible approach. Birkeland discovered that data plane encryption was possible using Open vSwitch and an IPsec tunnel, but a dynamic and easily configurable approach was not realised. To transform setup from a manual to dynamic process there was a clear need for further extension of controller, OpenFlow, and switch functionality. Comparison of the two

theses shows that Birkeland directly influenced Braadland's technology choices and certain design decisions of the key management scheme [Bir16; Bra17]. Birkeland's thesis supported the intended use of OpenFlow and OVS in this project and further expressed the need for an implementation of dynamic data plane key management. Overall, both Braadland and Birkelands theses can be interpreted as predecessors to this thesis, with each extending and improving on the previous.

Hauser et al. (2020) published the paper *"P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN"* which to the researcher's knowledge is the latest study with relevance to this thesis [Hau+20]. The paper proposes a protocol for SDN called P4-IPsec that enables applications to trigger the automatic creation and teardown of VPN network tunnels when access to specific data plane resources is required. P4-IPsec is an extension of IPsec that is designed for programmable P4 switches and replaces IKE with simpler functions for configuring network tunnels. Similarities in one use case of DPKM and P4-IPsec can be identified, as both enable automated configuration of data plane network tunnels in SDN. However, the approach taken is vastly different using alternative technology choices and architecture. Detailed comparisons between DPKM and P4-IPsec are challenging as the complex nature of IPsec makes P4-IPsec convoluted. Nevertheless, the paper will be referenced when evaluating DPKM in chapter 6 since development ran concurrently.

Similar utility of IPsec within SDN was also considered by Lopez et al. (2017) in the IPsec flow protection scheme presented in the internet draft *"Software-Defined Networking (SDN)-Based IPsec Flow Protection"* [LLP17]. Unlike Hauser et al. (2020), the draft model's extensions to IKE for IPsec tunnel establishment without implementation in SDN or use of P4 [Hau+20; LLP17]. Similarities to this project can be seen in the proposed extensions to controller functionality and the dynamicity of key management delivered by the extended IKE protocol. This research piece is the closest to Braadland's thesis (2017) theorising a similar project but with IPsec and extended IKE instead of WireGuard and DPKMS [Bra17]. Since no proof of concept has been delivered for either publication this project could have chosen to expand on the work of Lopez et al. instead [LLP17]. However, as explored in section 2.3.2 there are many benefits to using WireGuard over IPsec and the technology choices in Braadland's thesis are more suited to small scale development.

Chapter 3

Planning and Specification

This chapter presents the requirement specification and planning behind the project development timeline. Before designing DPKM it was important to determine the requirements of the system, formulate standard use case scenarios, choose a Software Development Life Cycle (SDLC), and decide the development timeline. This chapter begins with section 3.1, dividing the expectations of DPKM into cryptographic or security-oriented requirements and functional requirements. Following this, section 3.2 explores several use case scenarios for DPKM, and section 3.3 investigates how SDLCs can be used to structure development in phases. Finally, section 3.4 compares planned iterations of the development timeline to the actual development timeline.

3.1 Requirements

Determining project requirements is a crucial part of the development process that should precede planning and design. Requirements assist in ensuring the completeness and correctness of the implementation, providing a basis to evaluate the produced work for missing or incorrect elements. Since this project builds on the work of Braadland (2017) [Bra17], many of the requirements were used as inspiration for the cryptographic and functional requirements in this project in combination with guidance from NIST [Bar+13; Bar+09]. Cryptographic requirements (section 3.1.1) describe the security related properties that DPKM is expected to deliver. Functional requirements (section 3.1.2) outline behavioural aspects that form the standard operation and functionality.

3.1.1 Cryptographic Requirements

- **Authenticity of Entities:** All entities in the system need to be authenticated and any entities that cannot be authenticated must be prevented from participating. Entities should prove their identity before communication and there should be mutual authentication between the application, controller and network nodes.
- **Authenticity of Keys:** All keys need to be authenticated to prove that they belong to the right entity. Unauthentic keys should be revoked.
- **Integrity:** Keys in DPKM should be tamper-resistant and protected from alteration at all phases of the key management life cycle. There needs to be a mechanism for checking the integrity of the keys and connections should not be formed if a key is untrustworthy.
- **Confidentiality:** Communication between entities should be confidential and protected by encrypted communication channels.
- **Synchronicity:** All processes should be synchronized by only performing tasks when the associated nodes are in a ready state. The controller should use its view of the network topology to assist in maintaining a synchronized state between the nodes. This prevents data leakage, packet loss and inconsistencies in the key management procedures.
- **Freshness:** Each communication session needs to be confined to a time period after which the session key expires and must be rekeyed. Each key should only be used to encrypt a small amount of data to limit the damage of a compromised key.
- **Context Bound Keys:** When a key is generated it should be bound to its required context and the identity of an entity.
- **Access Control:** Measures should be put in place to prevent physical or remote access to entities, source code and databases by anyone that is not the administrator.
- **Strength of Key:** Keys should be generated by algorithms proven to be cryptographically secure and provide strong resistance to attack.
- **Compromise causes minimal damage:** Compromise of one key should not impact the security of the other keys.

3.1.2 Functional Requirements

- **32-byte key size:** DPKM should be built to generate and distribute 32-byte Curve25519 keys for WireGuard [Lip18].
- **Maintain secrecy of private key:** Private keys should be kept secret to the node, stored securely after generation, and never distributed or exposed. Only public keys should be distributed and stored outside of the entity.
- **Secure agreement of session keys:** Session keys between nodes must be agreed by a cryptographically secure key exchange procedure.
- **Rekey after expiry:** Keys should have a set lifespan (cryptoperiod) initially chosen by the administrator, after which they are replaced with minimal interruption to other processes.
- **Randomness of key:** The key generation process should be randomised and unpredictable.
- **Handle compromise:** Any compromise of an entity should be handled as quickly as possible by alerting the administrator, stopping communication/quarantining the entity, and revoking the keys.
- **Handle errors:** The controller should be aware of any errors in the entities and have methods for handling the errors. The controller needs to display or log errors for the administrator to review. Persistent errors or DoS attempts should result in an entity being quarantined.
- **Handle packet loss/delay:** Synchronization of the system should always be maintained, even in instances where packets are delayed or lost. Lost or delayed packets should be handled without detriment to key management processes.
- **Handle crash/restart:** If a network entity crashes all communication links should be restored once running again. Any communication with the crashed entity should be paused as soon as possible to prevent packet loss.
- **Intuitive UI:** The application plane UI should be intuitive and easy to use. The administrator should be able to understand the system status and perform DPKM functionality with no difficulty.
- **REST support:** All DPKM procedures should be accessible from the UI via REST APIs connected to the controller.

- **OpenFlow support:** All DPKM procedures and messages should be supported by an extended version of the OpenFlow protocol, allowing the controller to interact with the nodes.
- **Efficiency:** Construction/destruction of communication sessions must be fast, and executing DPKM functionality should feel instantaneous with minimal interruptions to normal system processes.
- **Responsiveness and Reliability:** All requests by the controller should trigger a response from the node that indicates successful delivery or the encountered error. Response time must be fast, and lack of a response should trigger retransmission of a request.

3.2 Use Cases

Use Cases are a software engineering technique for illustrating the sequence of events within a system and reducing the ambiguity of the requirements [BS03]. To specify the requirements of DPKM functionality three use case diagrams were constructed depicting the order of events between an administrator actor and many use cases. These use cases form the basis of the more detailed sequence diagrams in the design phase of this project (section 4.2). To understand the diagrams a grasp of the following concepts from [Hey00] is necessary:

- The stick figure is referred to as an actor and represents the entity interacting with the system to trigger events.
- The green ovals are use cases which represent some function or action within DPKM.
- A *"uses arrow"* from use case X to Y indicates that performing a function X means doing the function Y at least once. All subsequent use cases linked by uses arrows must be performed. For example, in figure 3.1 configuring a WireGuard interface means updating the database, setting the interface keys, and calculating new keys. The order is not important at this stage and will be further defined in the design stage.
- An *"extends arrow"* from use case Y to X means performing a function X can optionally result in performing the extended behaviour Y. For example, in figure 3.1 setting up the network tunnel can sometimes result in removing old peer details in circumstances where they are no longer valid.

Scenario one: Setting up the system

Figure 3.1 depicts the use cases related to an administrator setting up the DPKM system. This includes configuring the WireGuard interface, establishing the network tunnel between two interfaces, and starting communication. To start communication the functions for setting up the network tunnel and configuring the interfaces must be performed. Rekeying and removing keys/peers are optional functions and reserved for cases where there is outdated key or peer information that must be replaced.

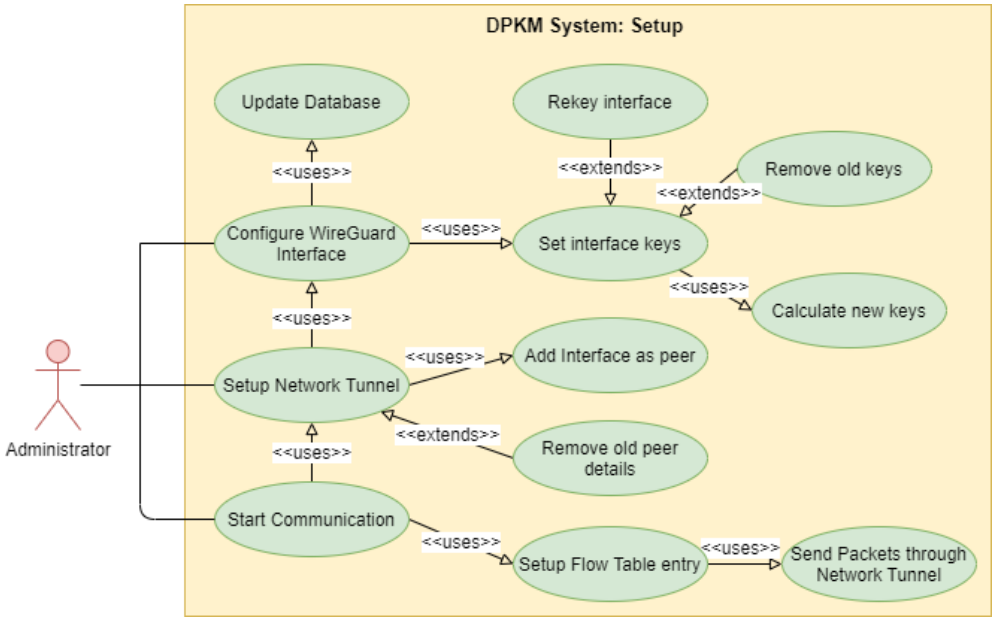

Figure 3.1: Use Case Diagram depicting the DPKM functions for setting up the WireGuard interface and network tunnels.

Scenario two: Tearing down the system

Figure 3.2 illustrates the use cases for tearing down or unconfiguring components in the DPKM system. The administrator can perform the functions of unconfiguring a WireGuard interface, tearing down an existing network tunnel, and ending communication between nodes. When the administrator unconfigures a WireGuard interface the optional function of tearing down the network tunnel is performed if any connections with the interface exist. When a network tunnel is removed the optional function of ending communication is performed if the two endpoints are communicating. When communication is ended there is a function to update the flow table and either remove the flow between the nodes or send packets unencrypted.

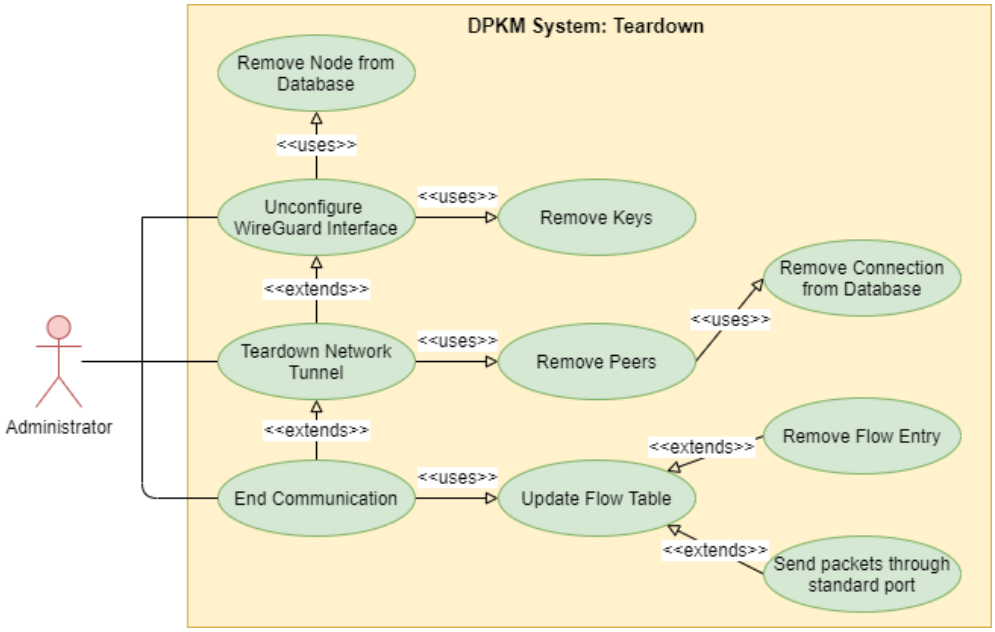

Figure 3.2: Use Case Diagram illustrating the DPKM functions for tearing down the WireGuard interface and network tunnels.

Scenario three: Handling security threats

Figure 3.3 investigates the use cases for identifying and handling compromised network nodes. There are two actors in this scenario: an external security software that detects the compromised node and an administrator that handles the compromise and revokes the key. When the security software detects a compromise, it performs the functions of ending communication with the insecure node and alerting the administrator. The administrator then handles the compromise by revoking the key and either reconfiguring the interface or terminating the node. In theory all three figures could be combined to form one large use case diagram. The reconfigure function in figure 3.3 could be linked to the configure WireGuard function in figure 3.1, and the end communication and unconfigure WireGuard functions in figure 3.2 could replace the end communication and terminate node functions in figure 3.3.

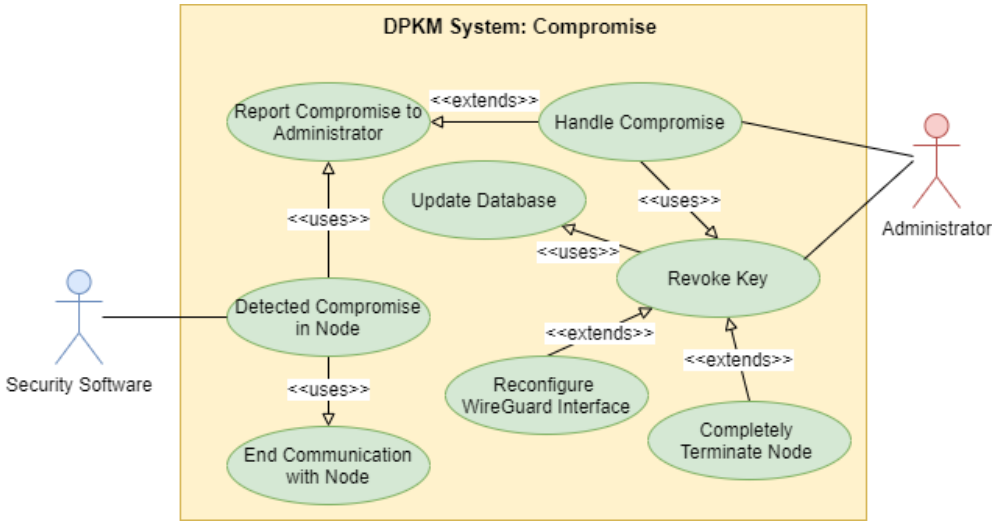

Figure 3.3: Use Case Diagram illustrating the DPKM functions for identifying and handling compromised network nodes.

3.3 Software Development Life Cycle

A Software Development Life Cycle (SDLC) is a conceptual model for structuring software development into phases of a life cycle that can be followed to transform the initial formulation of a product idea into a deployable product [Rup10]. No two software development projects are the same, meaning a multitude of different SDLC models are available and should be chosen on a project by project basis. Choosing the right SDLC means assessing compatibility with project related factors such as the preliminary specification and design requirements, development scale and team size, amount of stakeholder interaction, and timeline constraints. If chosen correctly, the right SDLC model helps reduce cost and project overhead, greatly assists in planning development to meet deadlines, increases product reliability, and prevents important stages from being skipped or overlooked [DBC88]. Analysis of popular SDLC models has been widely considered with several papers offering insight into strengths, weaknesses, and use cases [MD13; BM12; AB15; Ran17]. For brevity only the chosen iterative and incremental model will be discussed in detail, with a short summary of the Waterfall, V-Shaped, and Agile models.

Waterfall Model

The Waterfall model, shown in figure 3.4, is known as the traditional SDLC structuring development into the five phases: requirements, design, implementation, testing, and maintenance,

that must be completed in a fixed order [MD13]. The model provides several strengths including simplicity of implementation, well defined ordering of phases, and low resource overhead [AB15]. The model was not suitable for this project because of its lack of flexibility forcing requirements to be decided at the start of development. Since DPKM is a new product it was important to return to previous stages of development to expand requirements or functionality.

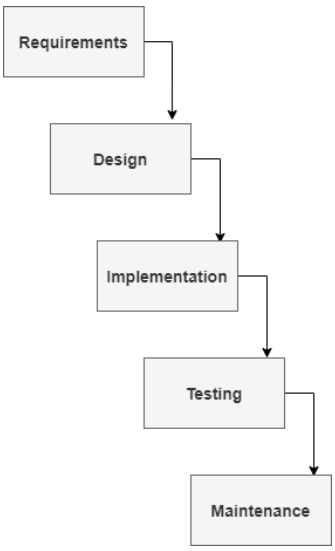

Figure 3.4: An example diagram showing the phases of the Waterfall SDLC model.

V-Shaped Model

The V-Shaped model, shown in figure 3.5, is a variation of the Waterfall model that focuses on testing and verification at each sequential stage of the development process [BM12]. It provides the same strengths as the waterfall model in addition to delivering more results and documentation from the testing processes at each stage. However, the model was still too rigid to be used for this project, and is seen as high risk since functionality is difficult to change and software is delivered at a late stage in development.

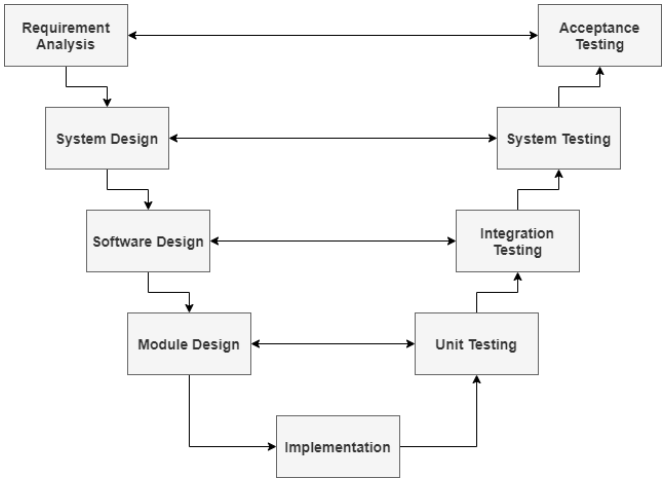

Figure 3.5: An example diagram showing the phases of the V-Shaped SDLC model.

Agile Models

Agile SDLC models are adaptive approaches to software development that offer more flexibility than traditional predictive SDLC models. Unlike the Waterfall model, Agile begins development with minimal planning aside from a set of future objectives [SMG13]. The focus of Agile is on delivering software rapidly and accommodating changing requirements from stakeholders at any stage of the development process. There is extensive team collaboration throughout, structuring development and product releases in iterations [BM12]. Agile would have been an attractive SDLC for this project as the flexibility it offers is suited to building new products with expected changes in functionality. However, Agile is designed for frequent team discussion and customer input whereas DPKM has a solo developer and no stakeholders. In this an Agile model would not provide any advantages over a traditional iterative and incremental model. Nevertheless, Agile would be selected over the iterative and incremental SDLC if future development on DPKM is continued by a team.

Iterative and Incremental Model

After researching several SDLCs it was decided that the iterative and incremental model in figure 3.6 would be used to structure the development of DPKM. This model enables development to be performed in iterations, with each iteration adding new functionality and requirements to the last, producing an updated version of the product until the finished version is delivered [AB15]. The phases of a single iteration are similar to the Waterfall model, but without the expectation of delivering a finished product at the end of the sequence.

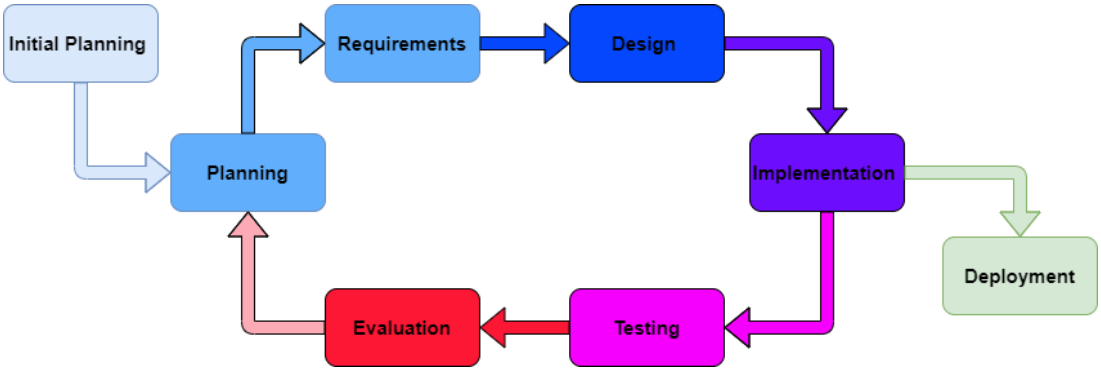

Figure 3.6: An example diagram showing the phases of the Iterative and Incremental SDLC model.

This model seemed the best choice for building DPKM because it was expected that development would require multiple iterations wherein issues can be addressed, and functionality incrementally extended. The novelty of DPKM means using new technology in an almost entirely undocumented way within an area that was unfamiliar to the developer. This suits the use cases for the iterative and incremental model since the model accommodates projects with new technology, in unfamiliar domains, where requirements can be prioritised but are expected to evolve [Tut20]. The main difficulty of using this model is the need to extensively plan the fully functioning system in advance so the product can be successfully divided into increments. Since this project is an extension of Braadland's work (2017) much of the functionality had been considered and could therefore be used to plan development [Bra17].

3.4 Progress Timeline

Once a SDLC model has been chosen, the next aspect of planning involves allocating time periods to the development phases to ensure a functional product is delivered by the final deadline. As discussed in the previous section, DPKM will be built in iterations following the iterative and incremental SDLC. During the original specification of this project the timeline in figure 3.7 was created using Microsoft office timeline [Tim20].

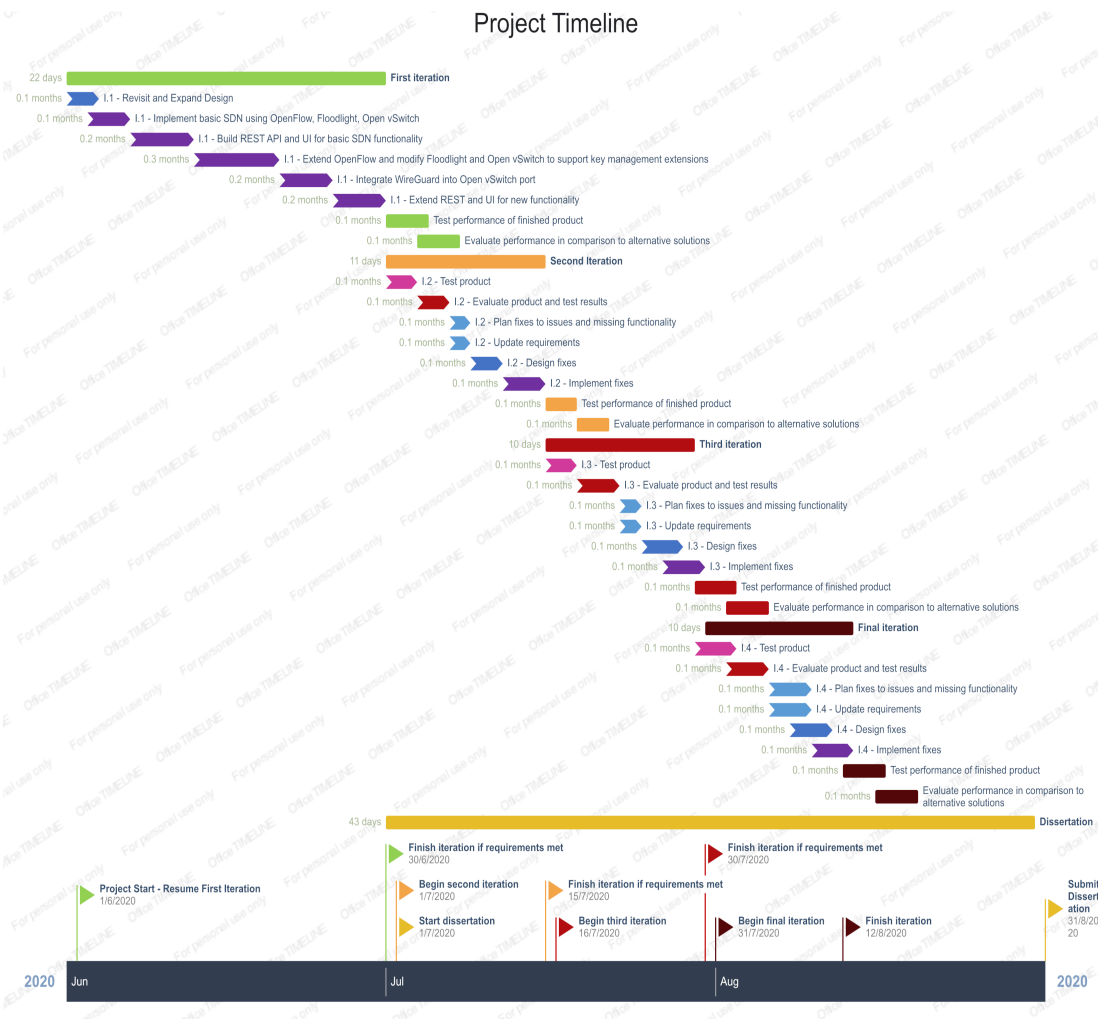

Figure 3.7: The original project completion timeline created with Microsoft Office Timeline [Tim20]

Figure 3.7 planned four iterations with the possibility of completion at the end of each iteration. Before starting development, it was predicted that two iterations would be needed to achieve a satisfactory base product. However, due to circumstances beyond the developers control implementation was delayed during the first iteration causing a shift in the development timeline. The project deadline was pushed from the end to August to December to mitigate any risk of incompletion and made it necessary to create a new timeline during the planning phase of the second iteration. Figure 3.8 shows the updated project timeline which was followed to meet the December deadline and deliver the full functionality expected of DPKM. The factors that delayed development will be discussed in detail in the evaluation chapter of this thesis (6).

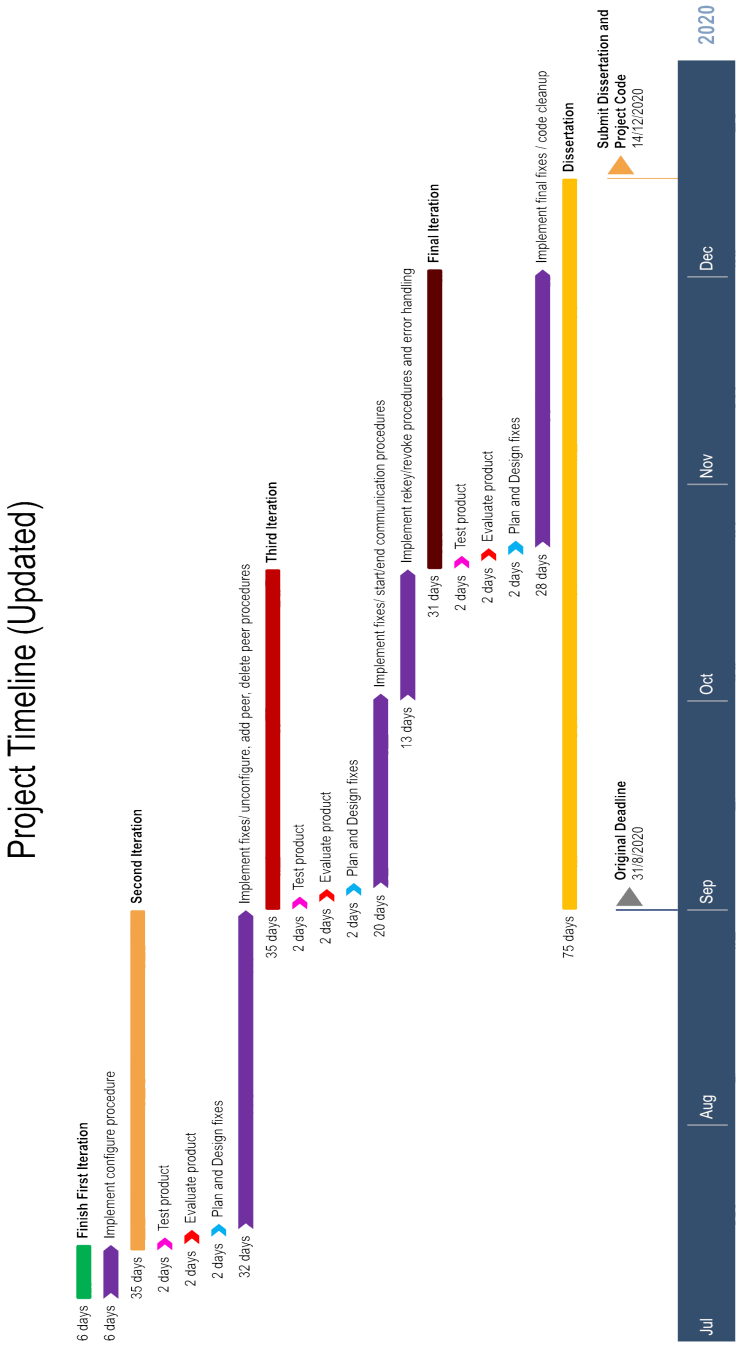

Figure 3.8: The updated project timeline created after extension of the deadline from August to December.

Chapter 4

Designing DPKM

This chapter details the architectural and functional designs conceptualising the Data Plane Key Management (DPKM) system into a software solution based on the requirements presented in chapter 3. The meticulous structural and operational design of DPKM, around the chosen technologies in section 2.3, creates a guide for implementation that can be followed to deliver a well-formed, functional product. This chapter begins with section 4.1 where the architectural designs of SDN components and connecting interfaces are formulated. Following this, section 4.2 builds on the requirements (3.1) and use cases (3.2) to establish the procedures and sequences of functionality within DPKM. Finally, section 4.3 proposes the OpenFlow experimenter message extensions necessary for DPKM to operate and perform key management functions.

4.1 Architectural Design

The architectural design of a system involves breaking down components by their individual structure and characteristics, considering the interfaces and connections between them, and merging them to form the entire system topology. The technologies for developing DPKM were selected in section 2.3 and collectively form the network architecture in figure 4.1. Figure 4.1 designs the full DPKM system for implementation across three virtual machines (VMs) with one controller connected to two nodes through the southbound API. The first VM must implement a controller and a connected application UI. The other two VMs must implement network nodes containing virtualized switches and connected network tunnelling interfaces. With reference to figure 4.1, this section proposes the individual architectural designs and

responsibilities of the controller (4.1.1), node (4.1.2), and application (4.1.3), before planning the interfaces for assembling the full DPKM system (4.1.4).

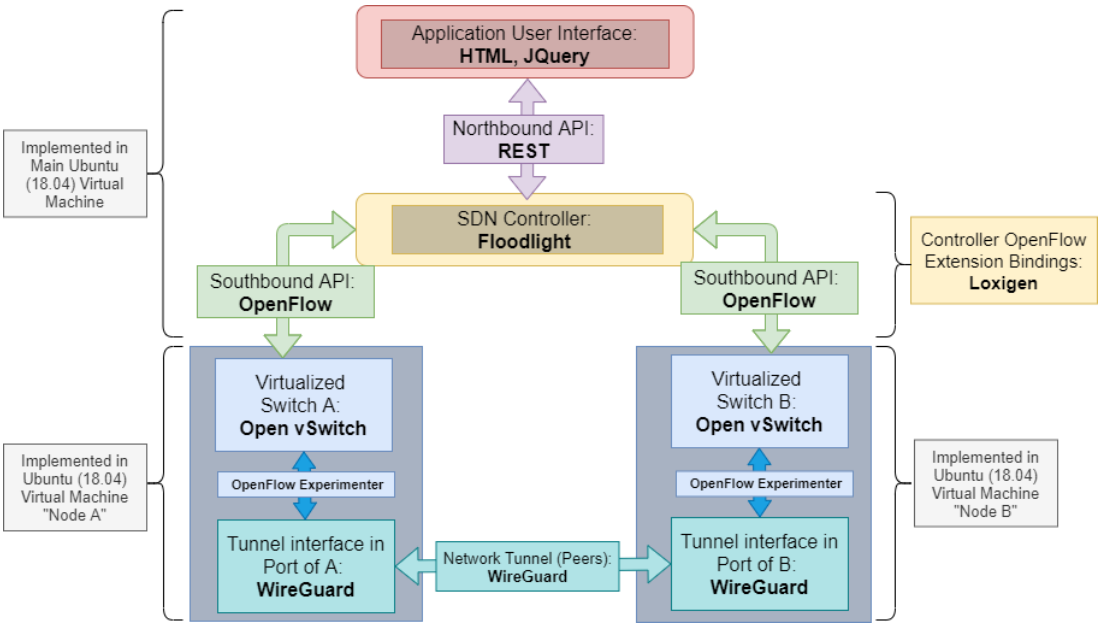

Figure 4.1: The full architectural design of the DPKM system to be developed.

4.1.1 Controller

The controller is the central mediator of all DPKM and general SDN functionality, overlooking the network topology and packet flow, handling communication and errors, and performing key management operations to facilitate the cryptographic key life cycle. In the DPKM architecture the controller will be developed by extending the java-based Floodlight controller (section 2.3.4) to support additional functionality, provide REST connections to the application UI and OpenFlow connections to the nodes, with an integrated database for storage. Before designing classes and database tables it was important to compile a list of the controller's duties based on the background research (chapter 2), Braadland's work [Bra17], and requirement specification (chapter 3). The duties of the controller in the DPKM system can be summarised as follows:

- Configuring/Unconfiguring the WireGuard interfaces.
- Setting up network tunnels by distributing public keys / peer information.
- Tearing down network tunnels and removing peer information.

- Storing DPKM related information about the network nodes including public keys, IP addresses, and status.
- Storing peer connections and keeping track of connections/communication.
- Constructing DPKM messages and sending to nodes via extended OpenFlow.
- Listening for and handling status or error responses.
- Sending subsequent messages based on a particular response. For example, status confirms that one peer has been successfully added so add the other peer.
- Starting encrypted communication by setting up flow table entries.
- Ending encrypted communication by modifying/removing flow table entries.
- Maintaining and querying the database to accurately represent the state of the nodes.
- Providing REST APIs for performing all procedures with the application UI.
- Rekeying after expiry of the set cryptoperiod.
- Reporting errors or compromises.
- Revoking compromised keys and terminating untrustworthy nodes.

With a clearer impression of the expectations of the controller the next aspect of the design process is converting the duties into class and database designs. Since Floodlight [Flo20] has been built in java the implementation will be object-oriented meaning UML class diagrams can be used to describe the class structure. UML class diagrams plan the structure of a system by mapping the classes, their attributes and methods, and the relationship between them [Par20]. Studying the documentation for the standard Floodlight modules and looking ahead at tutorials gave an impression of how classes should be structured and enabled the class diagram in figure 4.2 to be designed. Figure 4.2 structures the DPKM controller system into the following colour coded classes:

- **Blue:** The core DPKM controller system for sending/receiving messages to the nodes, bridging REST and OpenFlow, and interacting with the database. The “Main Listener” must be responsible for capturing any DPKM status messages from the nodes, processing the messages by querying the database and constructing responses, implementing the REST service and all DPKM methods, and adding the REST routes/other dependencies. This class should extend two classes, “Dpkm” and “DpkmFlows”. “Dpkm” will contain

the expected internal methods for querying and updating the database. “DpkmFlows” will contain the expected internal methods for managing the flow tables in the node.

- **Green:** The classes for connecting to the database and mapping node and connection information to objects. The “ConnectionProvider” class is needed to connect to an external database, with “Dpkm” establishing zero or many connections, depending on the required access. The “DpkmSwitch” class should store information for a DPKM node and the “DpkmPeers” class should store information for two connected peers. “Dpkm” will establish zero or many switch/peer objects, depending on the methods used.

- **Yellow:** Two standard Floodlight interfaces that were identified in the documentation as being necessary for DPKM to operate. “IOFSwitch” is an interface representing an OpenFlow switch that can be used for interactions such as sending messages [Doc20b]. The DPKM listener will use this interface to interact with the nodes and can have zero or many switches depending on how many are in the topology. “IOFMessageListener” is the interface used throughout Floodlight to listen for OpenFlow messages and will be implemented by the DPKM listener to capture responses [Doc20a].

- **Purple:** The classes for error handling in the DPKM system. The “Error Listener” must be responsible for capturing any DPKM error messages from the nodes and processing the errors by calling a handler based on the error type. This class should extend the “DpkmError” class which like “Dpkm” will contain internal methods for querying and updating the database to track the state of the nodes. The error listener should be packaged with the main listener, meaning each main listener has one associated error listener.

- **Red:** The REST API system for registering the paths to REST resources and providing an abstract interface for accessing the DPKM functions. “DpkmWebRoutable” should create a Restlet specifying the paths to resources which are registered by the main listener. “DpkmRestService” should define the abstract DPKM functions implemented by the main listener.

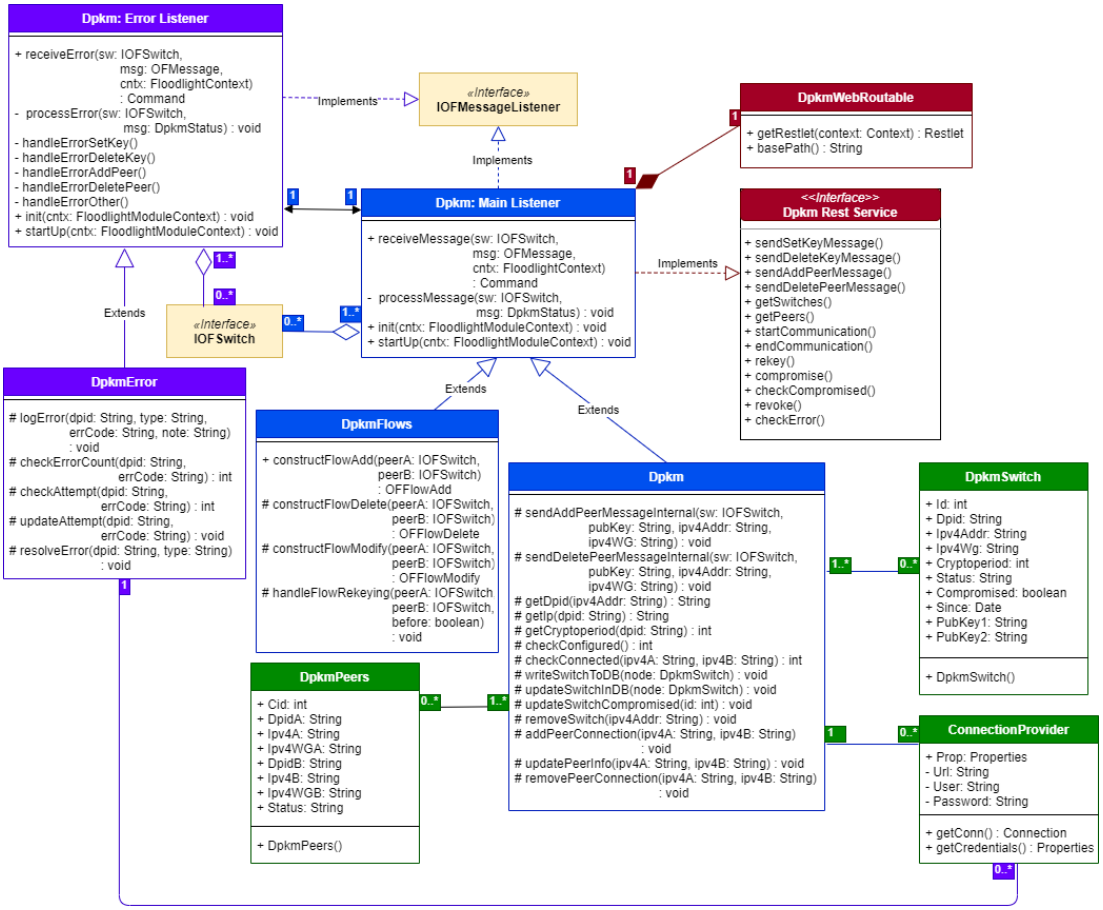

Figure 4.2: UML class diagram structuring the required classes for realising the DPKM controller.

A multitude of REST resources are needed to provide `GET`, `POST`, and `DELETE` methods for all DPKM procedures. An additional class diagram in figure 4.3 was created for this purpose, structuring the REST system into the following colour coded classes:

- **Yellow:** The classes for providing REST functions and resources to DPKM procedures. "ConfigureWGResource" must provide the functionality for retrieving a list of nodes (`GET`), configuring WireGuard (`POST`), unconfiguring WireGuard (`DELETE`), and translating json from the UI into "DpkmSwitch" objects. "ManagePeerResource" must provide the functionality for retrieving a list of peers (`GET`), adding a new peer connection (`POST`), deleting a peer connection (`DELETE`), and translating json into "DpkmPeers" objects. "StartCommunicationResource" and "EndCommunicationResource" must provide the `POST` functionality for starting/ending encrypted communication. "RekeyRe-

source" must provide `POST` functionality for rekeying a node. "CompromiseNodeResource" and "RevocationResource" must provide the `POST` functionality for simulating a compromised node and revoking a compromised key respectively.

- **Blue:** "DpkmWebRoutable" for assigning a path to each resource. There must be a composite relationship meaning none of the resources can exist without the "DpkmWebRoutable" class.
- **Red:** "DpkmRestService" which must be initialised and used by each resource to perform DPKM functionality. There must be an aggregate relationship meaning each resource uses an instance of "DpkmRestService" but "DpkmRestService" can exist outside of the resources (such as in the main listener).
- **Green:** Objects representing nodes and peer connections as explained above.

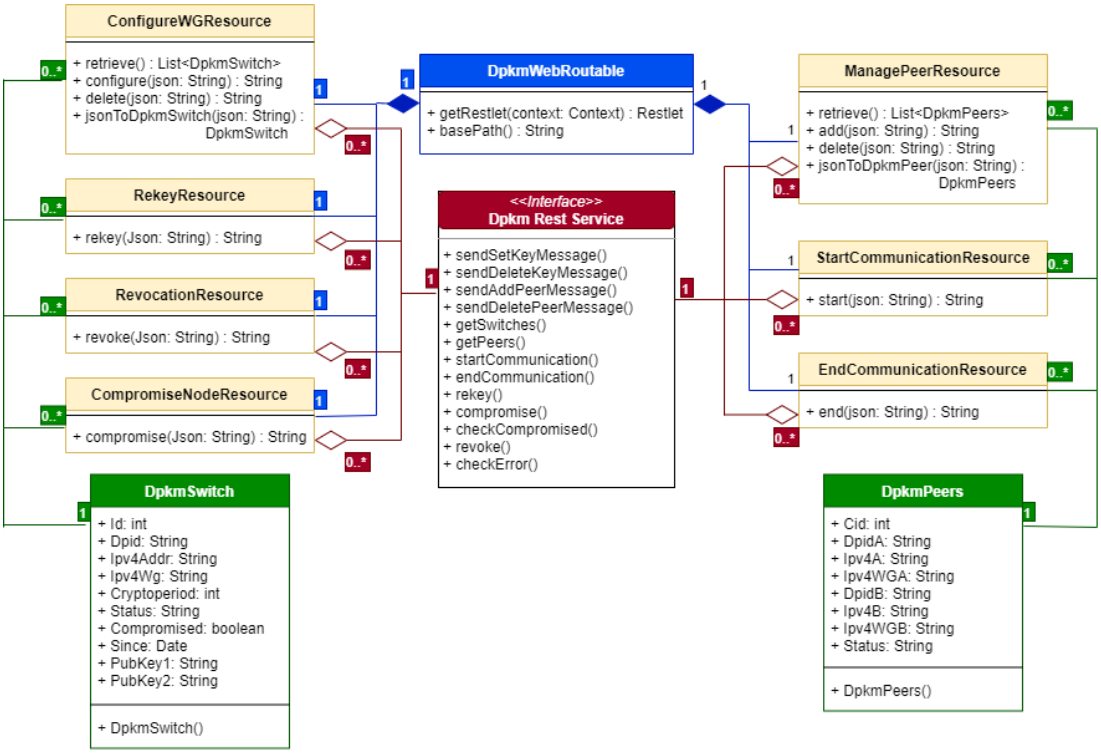

Figure 4.3: Focused class diagram structuring the classes comprising the REST system in the DPKM controller.

For the controller to successfully operate as a key management entity a database is required to assist in tracking the state of the data plane components. From the expected functionality and class diagrams it can be deduced that three database tables are needed: one to store information related to the node, one to store peer connections, and one for logging errors. Figure 4.4 presents a simple model for the database tables required by the controller.

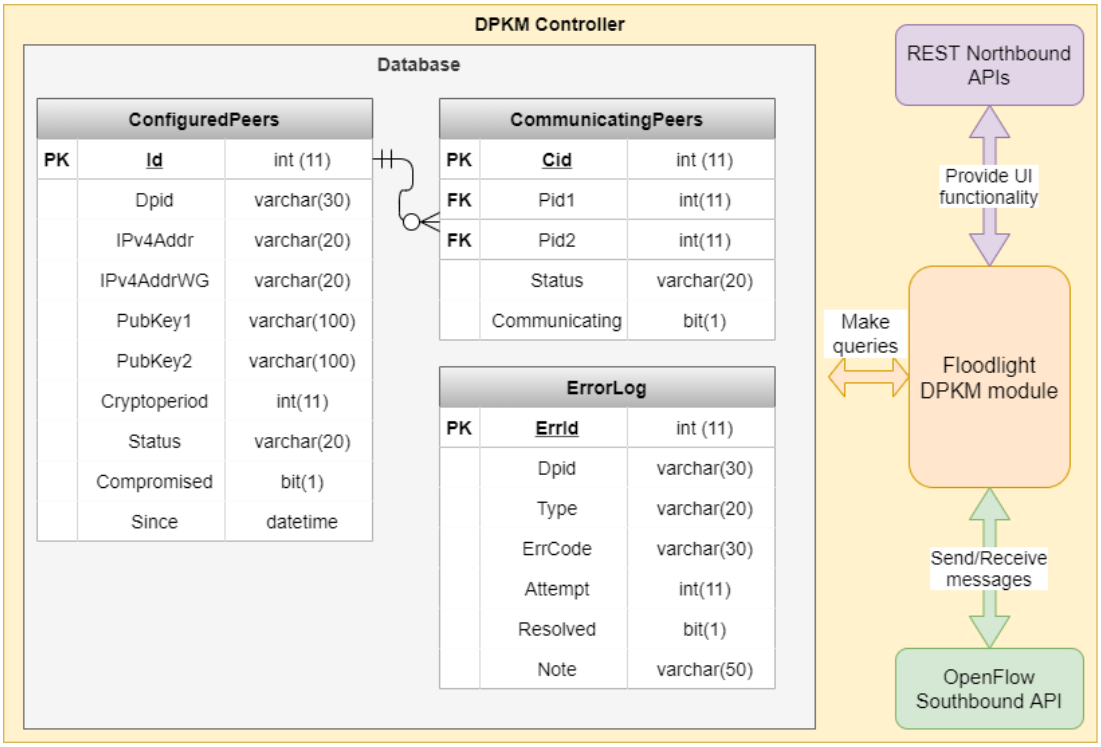

Figure 4.4: Model of the DPKM controller proposing the schema of three database tables: ConfiguredPeers, CommunicatingPeers, and ErrorLog.

The content of the three database tables can be summarised as follows:

1. **ConfiguredPeers** stores information related to a network node in DPKM. Each node is assigned a default record Id, a Dpid or DatapathId which is the unique mac address of the switch, two ipv4 addresses one for the switch and one for the WireGuard interface, two public keys one current and one previous, a cryptoperiod value for rekeying, the status of the node, a Boolean value indicating compromise, and a datetime timestamp of when the node was last configured.

2. **CommunicatingPeers** stores information related to a pair of peer network nodes in DPKM. Each record in communicating peers has a default record Cid, and two foreign keys, Pid1 and Pid2, which are individually connected to a single Id in ConfiguredPeers. The table also requires a status field explaining the state of the connection and a Boolean value indicating if the peers are communicating.
3. **ErrorLog** stores diagnostic information describing an error that was encountered during operation of DPKM. Each error has a default record ErrId, the Dpid of the node that encountered the error, the type of error, the exact error code returned by the switch, the number of retransmission attempts, a Boolean value indicating resolution, and a note for additional information.

4.1.2 Node

As depicted in figure 4.1, a network node in DPKM is composed of a virtualized Open vSwitch (2.3.3) with an integrated WireGuard interface (2.3.2). The node will be setup as a standard OVS switch with a TLS-encrypted OpenFlow connection to the controller. The WireGuard interface will be assigned to an OVS port during setup and serves as the encrypted connection to other nodes. To maintain the decoupled nature of SDN, where switches are simple forwarding entities, the nodes responsibilities in DPKM will be kept minimal and consist of the following duties:

- Performing functionality dictated by the controller.
- Returning status messages on success or error responses on failure.
- Translating messages from the controller to WireGuard commands.
- Forwarding packets through the network tunnel to a peer interface (when required).

Designing specific classes for extending OVS functionality prior to development was challenging for two main reasons. Firstly, the OVS code base is built in C which is a procedural programming language and not object-oriented meaning construction of a class diagram would serve little purpose. Secondly, the functionality of existing classes will need to be extended rather than adding an entirely new module as in Floodlight. Nevertheless, the design of the node is realised through the procedures in section 4.2 and handling of the messages detailed in

section 4.3. Moreover, since node operation is based on the functionality communicated by the controller the following extensions to the OVS code base are necessary:

- An OpenFlow extension class defining DPKM OpenFlow experimenter messages in the same style as the controller to ensure consistency and successful mapping of messages between both entities into structs and objects.
- Handlers decoding all DPKM OpenFlow messages from the controller into usable structs and attributes.
- Handlers encoding status or error responses with specific information relating to the decoded message and respective function.
- Kernel level script or OVS actions that can make changes to the WireGuard interface.
- Support for new flow matching conditions to direct packets intended for DPKM through the WireGuard port.

4.1.3 Application

In DPKM the application is a web user interface (UI) that is deployed when the controller is running and can be accessed from a browser with a URL containing the site address, controller IP, and port 8080. As described in section 2.3.5, the Floodlight controller comes equipped with a Web GUI [Iza16] that enables the administrator to manage the full network topology and perform all standard SDN operations through REST APIs. Since the focus of this thesis is on developing key management in the data/control planes it was decided that the Floodlight Web GUI would be extended to perform DPKM functionality, instead of undergoing the arduous process of building an entirely new UI. Moreover, to enable more time to be dedicated to developing software in the controller and node the extended UI has been designed to be basic, only providing the means to carry out the DPKM procedures with aesthetics left for the end of development or future work (6.6). The duties of the application consist of providing the administrator with the ability to instruct the controller to perform DPKM functions and can be summarised as follows:

- Choosing which nodes to configure and setting the cryptoperiod to decide how often the controller should rekey the interfaces.

- Setting up peer connections to decide encrypted flows between nodes and starting communication.
- Tearing down network tunnels and choosing to end communication completely or continue unencrypted.
- Displaying information from the database about the current state of DPKM including nodes, peer connections, and communication.
- Removing nodes and deleting outdated or expired data.
- Addressing compromise and dictating how revocation should be handled.
- Informing the administrator of errors.

To better visualize the design of the UI the three use cases specified in section 3.2 were chosen as a basis for constructing several prototypes. Three scenarios were mocked to give a feel for DPKM from the administrator's point of view, with the underlying sequence of functionality considered in section 4.2.

- **Scenario One:** Setting up an encrypted channel between two nodes.
- **Scenario Two:** Tearing down/modifying an existing connection.
- **Scenario Three:** Handling a compromised node.

When an administrator connects to the UI, they are presented with the default home page displayed in figure 4.5, which presents status information about the controller and network topology. From here the administrator can begin scenarios one and two from the "Manage Switches" page by clicking the "Encrypted Links" button or using the navigation bar.

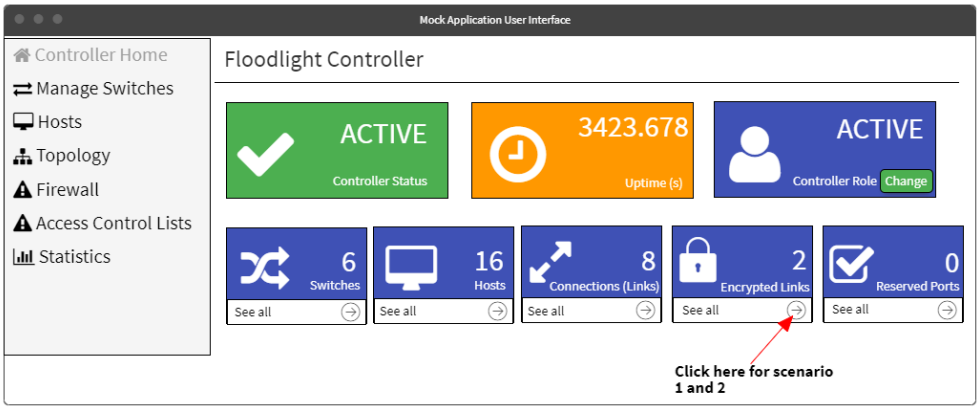

Figure 4.5: Mock of the application UI Home Page.

The “Manage Switches” page, displayed in figure 4.6, is where all DPKM functionality can be performed. Information from the two database tables “ConfiguredPeers” and “CommunicatingPeers” (4.1.1) is displayed in the switches and connections tables through `GET` queries over the REST connection to the controller. For brevity only the basic information required by the scenarios has been mocked, whereas in reality several extra columns will be displayed. To add a new encrypted connection (scenario one) or modify an existing connection (scenario two) the indicated buttons in the connections table are clicked. It is worth noting that the “Add Switch” button can be clicked to configure a new node, but this scenario has not been mocked.

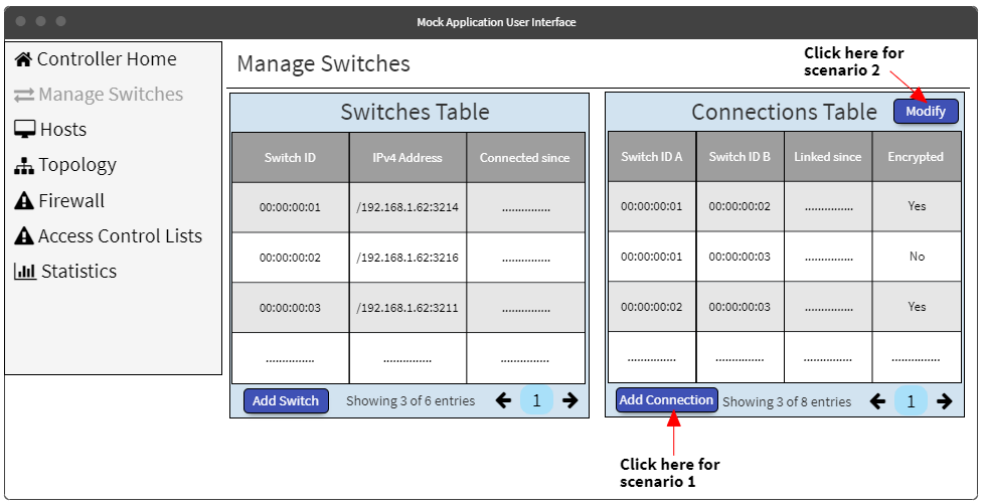

Figure 4.6: Mock of the application UI "Manage Switches" page for performing DPKM functionality.

Scenario One

Clicking the indicated "Add Connection" button triggers the modal form in figure 4.7. The administrator enters the MAC address id of each switch and the ipv4 endpoint to receive packets on. If the encrypted checkbox is checked, the session period box appears for the administrator to set a cryptoperiod for rekeying the connection. Upon clicking the "Save Changes" button, an encrypted WireGuard connection is setup by making a `POST` REST request to the controller and executing the *Add Peer* procedure discussed in section 4.2.3. If the checkbox is unchecked a standard unencrypted SDN link is created without using WireGuard or DPKM procedures.

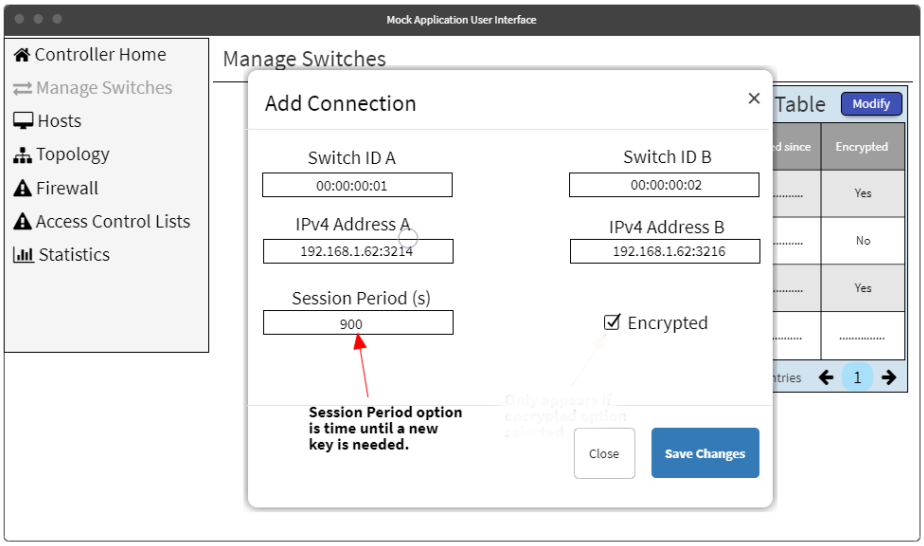

Figure 4.7: Add Connection modal popup form for setting up a new peer connection between nodes.

After performing the underlying functionality for adding the new connection the controller displays the status popup window shown in figure 4.8. This window confirms the success of the procedure or displays the error that was encountered. The connection will now appear in the connections table to represent the new state of the data plane.

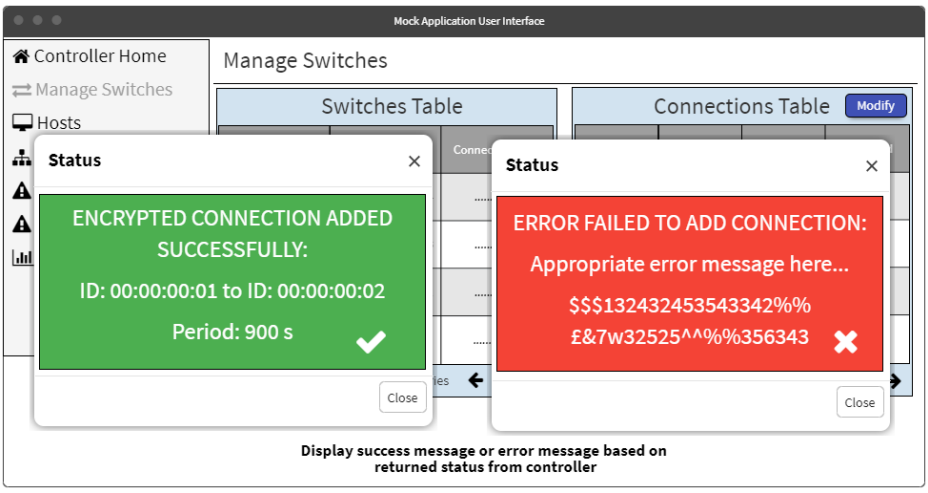

Figure 4.8: Status window indicating successful setup of a peer connection (left) or the error that was encountered (right).

Scenario Two

Clicking the “Modify” button on the “Manage Switches” page (figure 4.6) changes the interface to figure 4.9. Spanner icons appear in a column next to each connection that can be clicked to modify active connections.

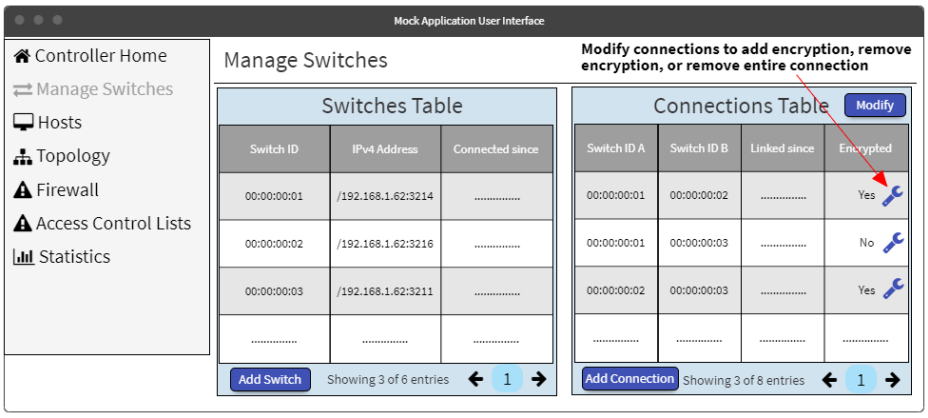

Figure 4.9: "Manage Switches" page after the "Modify" button has been clicked.

If the first connection was clicked the modal window in figure 4.10 would be displayed with the details of the two peer nodes. The administrator is now presented with the decision of removing the entire connection between the nodes or keeping an unencrypted link. Both buttons make

the same `POST` request for triggering the *End Communication* procedure (section 4.2.6) but add a unique type value to ensure the controller performs the correct variation.

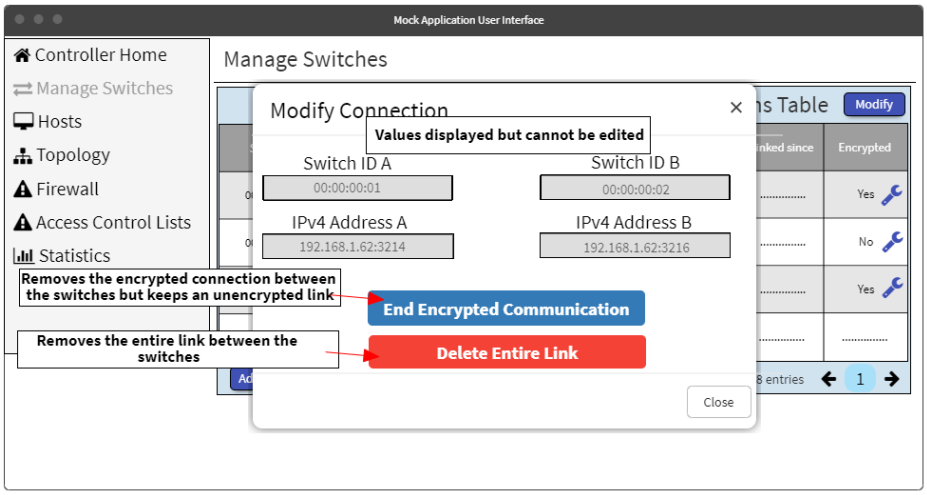

Figure 4.10: Modify Connection modal popup for ending encrypted communication only (blue button) or removing the connection entirely (red button).

After performing the underlying functionality for modifying or removing an existing connection the controller displays a similar status popup (figure 4.11) to scenario one, confirming success or reporting the error that was encountered. Depending on the chosen variation, the connection will be removed from the connections table or set to unencrypted, reflecting the systems state.

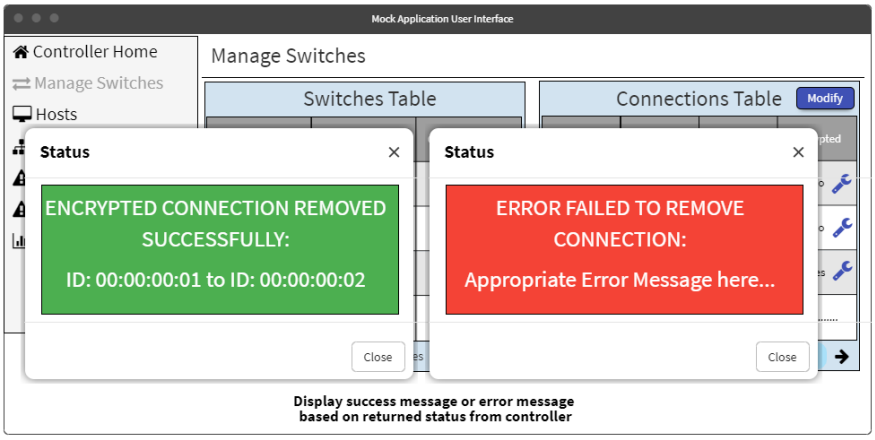

Figure 4.11: Status window indicating successful termination of an encrypted peer connection (left) or the error that was encountered (right).

Scenario Three

To consider how a compromise will be handled a different version of the home page was designed. Figure 4.12 displays the home page but with a change in the status of the controller from “active” to “action needed”, alerting the administrator to a threat or fault in the system.

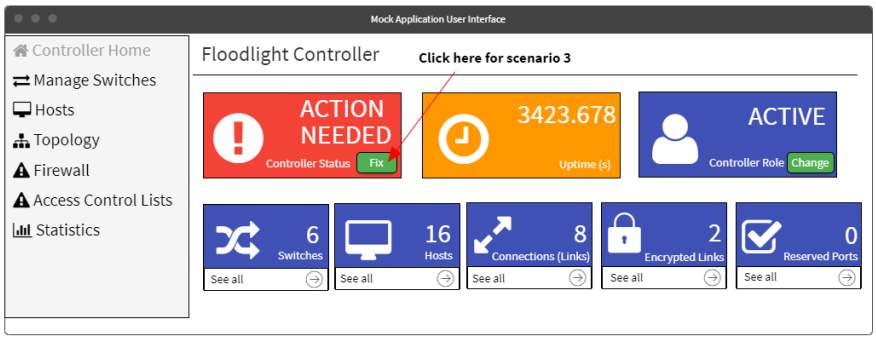

Figure 4.12: Mock of the application UI Home Page with "Action Needed" from the administrator.

To address this issue the administrator clicks the “Fix” button, triggering the modal window in figure 4.13. This window displays diagnostic information about the security threat or error, including details of the affected node and any connections it has, and offers options for handling it. In the case of a compromised node, the administrator can choose to reconfigure the node and its connections with a new set of keys or terminate the node and all connections entirely. Both buttons use the same REST API to make a `POST` request to the controller, triggering a different variation of the revocation procedure discussed in section 4.2.8.

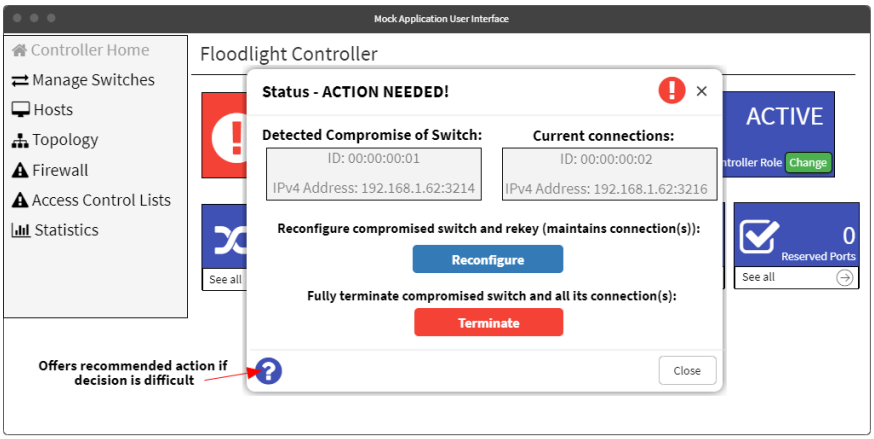

Figure 4.13: Action Needed modal popup for handling a compromised node by reconfiguring and rekeying (blue button) or full termination (red button).

Once the underlying functionality is performed by the controller a similar status or error window to figures 4.8/4.11 is displayed, indicating the successful or unsuccessful completion of scenario three. To ensure the security of the system is maintained before an administrator can act the controller will immediately end all communication with a compromised node. This quarantines the compromise to stop spread to other entities in the system.

4.1.4 Interfaces

Assembling the full DPKM architecture (figure 4.1) involves linking the previously discussed components through several interfaces that serve as bridging communication channels. Since instructions from the administrator must be passed from the application to the node it is crucial that all entities are interconnected and able to maintain or make concurrent changes to a synchronized state. Figure 4.14 is a component diagram that conceptualises the links between components. The four labelled interfaces – northbound (1), southbound (2), inner node (3), and node-to-node (4) – must be developed for DPKM and will be examined in this section.

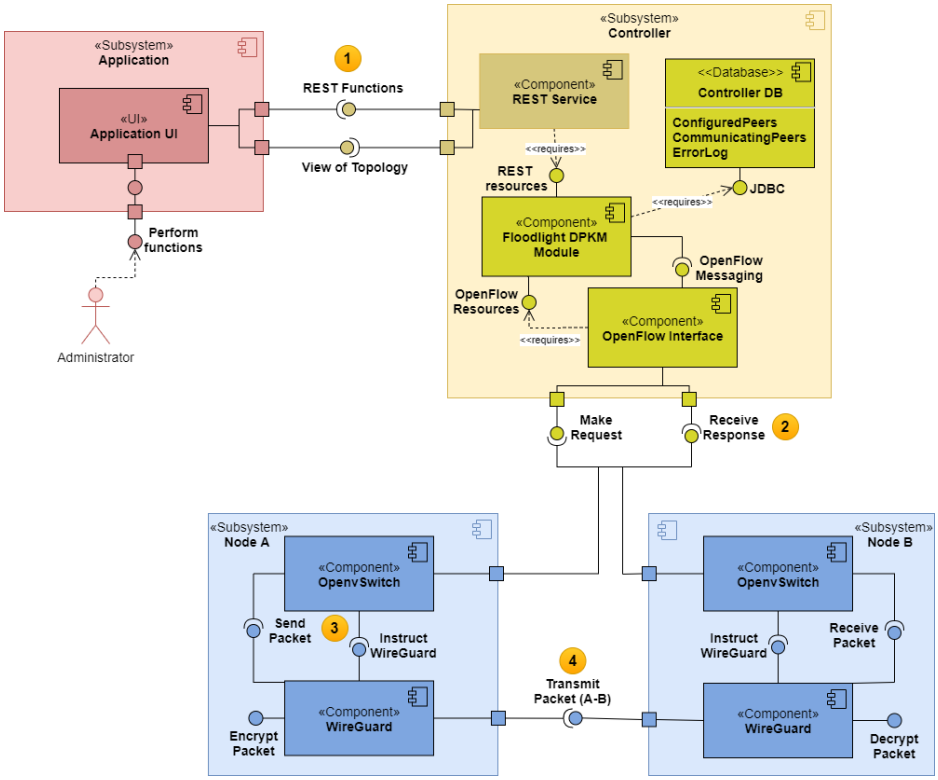

Figure 4.14: UML component diagram mapping the relationship between DPKM components with four main interfaces - northbound (1), southbound (2), inner node (3), and node-to-node (4).

Northbound

The northbound interface is built on the Floodlight controller and connects to the application UI, enabling the administrator to perform all DPKM functionality. Floodlight only supports RESTful interfaces meaning the northbound interface must be realised with REST. As described in 4.1.1, each procedure (4.2) has corresponding REST resources to which the controller assigns URL paths. These URLs can be used in a request/response pattern where calling a URL makes a request to a unique endpoint in the controller, triggering the corresponding function, with the controller returning some data in the response. Three types of HTTP requests are needed in DPKM – `GET`, `POST`, and `DELETE`. `GET` requests retrieve resources such as lists of nodes and peers. `POST` requests create new resources such as configuring nodes, adding peer connections, and managing communication. `DELETE` requests delete existing resources such as unconfiguring nodes and removing peer connections.

Each URL should be mapped to an element of the UI, with `GET` requests populating tables with returned data, `POST` requests passing json values from forms to create new objects, and `DELETE` requests providing ids of resources to remove. Theoretically, the DPKM system could be used without the UI by calling these URLs directly. However, this would defeat the purpose of SDN since network management would be difficult and the state of the system would be unclear.

Southbound

The southbound interface is the channel between the Floodlight controller and OVS switches. Floodlight, like most SDN controllers, exclusively uses the OpenFlow protocol for southbound communication meaning support for OpenFlow is available out of the box. OVS also uses OpenFlow exclusively and only requires the IP address and port of the controller to fully establish the channel. This thesis has already covered standard OpenFlow in depth (2.3.1) and builds on this knowledge to develop extensions for key management functionality. To perform the additional duties expected of the controller (4.1.1) and node (4.1.2) OpenFlow needs to be extended to support transmission of new messages between Floodlight and OVS in a request/response pattern. Since OpenFlow offers support for extensions through the "Experimenter" message type (2.3.1) all new messages can be implemented as subtypes. This means the challenge of extending the OpenFlow protocol lies not in the protocol itself, but rather the construction and processing of new messages by the controller and switch software. OpenFlow message extensions will be designed in section 4.3 and the implemented in section 5.2.

Inner Node

The inner node or OVS-to-WireGuard interface is more of an abstract connection that intertwines OVS and WireGuard to form a complete network node. The design of the node was formulated in 4.1.2 where an OVS switch integrates the WireGuard interface into a dedicated port. When the switch receives a DPKM message from the controller over the extended OpenFlow protocol it must translate the message into executable WireGuard commands through the inner node interface. This can be achieved through OpenFlow experimenter extensions that trigger the execution of hidden script files or kernel modules within the isolated machine hosting the node.

Node-to-Node

The node-to-node interface is the encrypted network tunnel between two WireGuard interfaces. The purpose of DPKM is dynamically setting up and managing this interface from the application UI. To achieve this all previously discussed components and interfaces must be used in synchronicity so both tunnel endpoints have matching configurations and can successfully engage in encrypted communication. WireGuard is a layer three network tunnel meaning only IP packets can be sent over this interface and non-IP packets must be encapsulated as IP. The WireGuard architecture was covered extensively in section 2.3.2 and no designs are needed for the interface itself.

4.2 Procedures

Designing the underlying functionality of the proposed DPKM architecture involves the construction of several procedures to perform key management operations and transition the system between different states. It was important to ensure administration of the complete key management lifecycle described in section 2.2.2, whilst fulfilling the requirements set in section 3.1 and expectations of the entities discussed in section 4.1. Since DPKM is an extension of Braadland's thesis (2017) similar operations need to be developed and several of the designs were influential to the construction of the procedures presented in this section [Bra17]. However, a multitude of alternative design decisions were made for this project meaning the sequences of functionality have changed drastically and must be redesigned. DPKM procedures are formed from the expected transitions between the different states of a network node illustrated by the state transition diagram in figure 4.15.

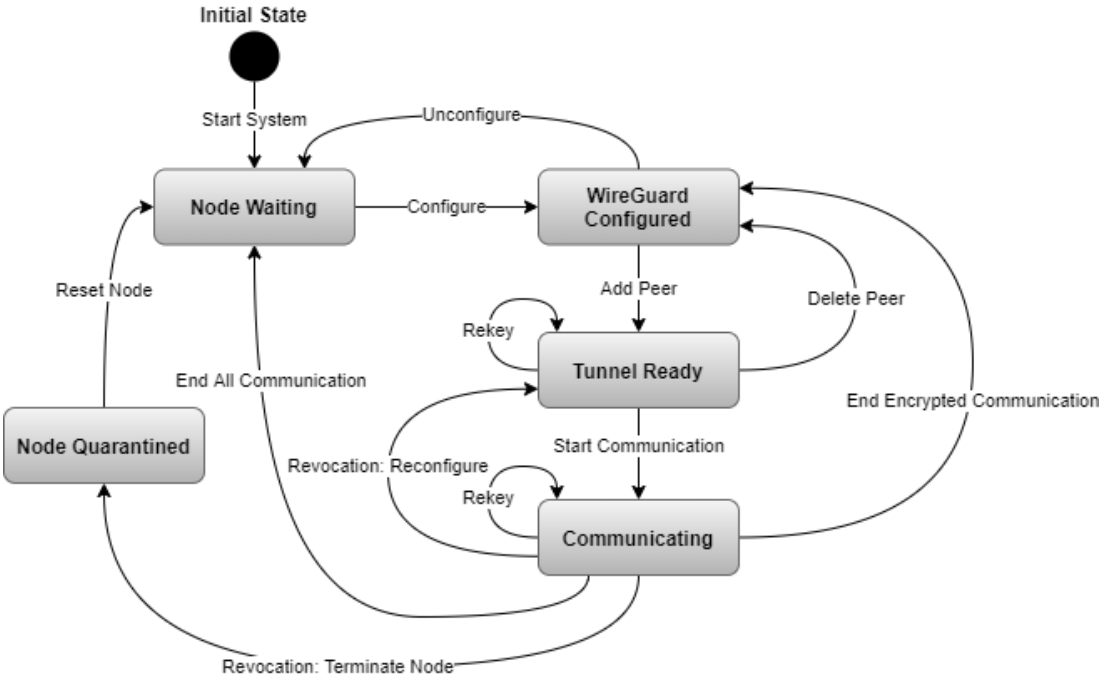

Figure 4.15: State transition diagram for a node in the DPKM system, plotting possible states and transitions between them.

When the system boots up the node begins in the *"Node Waiting"* state. In this state the node is a standard OpenFlow switch with an unconfigured WireGuard interface containing no keys or peer connections. To begin any DPKM functionality the node must transition to the *"WireGuard Configured"* state by running the configure WireGuard procedure (4.2.1) to generate and bind cryptographic keys to the interface. From here a WireGuard tunnel to another configured node can be setup by performing the add peer procedure (4.2.3) to distribute/exchange keys and transition to the *"Tunnel Ready"* state. In the *"Tunnel Ready"* state both nodes have exchanged all the necessary peer information to form an encrypted network tunnel but are yet to start communication. The start communication procedure (4.2.5) transitions the node from the *"Tunnel Ready"* to the *"Communicating"* state by setting up the flow rules for routing packets through WireGuard. A node in the *"Communicating"* state requires no further setup and is now engaged in encrypted communication. When the predetermined cryptoperiod lifespan of a key has been reached the rekeying procedure (4.2.7) replaces keys with a new set. Rekeying occurs in both the *"Tunnel Ready"* and *"Communicating"* states to ensure the peer connection maintains its security by replacing expired keys.

To end the communication session two variations of the end communication procedure (4.2.6) can be performed to either continue unencrypted or terminate all communication, transitioning from the *"Communicating"* state to the *"WireGuard Configured"* or *"Node Waiting"* states respectively. Tunnels can be removed with the delete peer procedure (4.2.4), transitioning from *"Tunnel Ready"* to *"WireGuard Configured"* state. Nodes in the *"WireGuard Configured"* state can also return to the initial *"Node Waiting"* state with the unconfigure WireGuard procedure (4.2.2). If a node is compromised two variations of the key revocation procedure (4.2.8) must be available to either reconfigure the node with new keys or fully terminate the node. Reconfiguring the compromised node transitions from the *"Communicating"* to *"Tunnel Ready"* state since communication is ended and the tunnel is rebuilt. Terminating the compromised node transitions from the *"Communicating"* to *"Quarantined"* state since the node is unconfigured and excluded from the system. To allow re-entry the administrator must manually reset the node, returning it to the initial *"Node Waiting"* state.

This section dissects the aforementioned states and transitional procedures in depth, proposing the expected sequence of functionality and communication within each procedure. The messaging between components has been considered further in section 4.3.

4.2.1 Configuring WireGuard

The procedure for configuring WireGuard requires nine steps and communication between four system components – Application UI, Floodlight controller, Open vSwitch (OVS), and WireGuard (WG) – through the interfaces discussed in section 4.1.4. The sequence diagram in figure 4.16 illustrates the order of functionality from initiation by the UI, to key generation and storage, and finally status or error response.

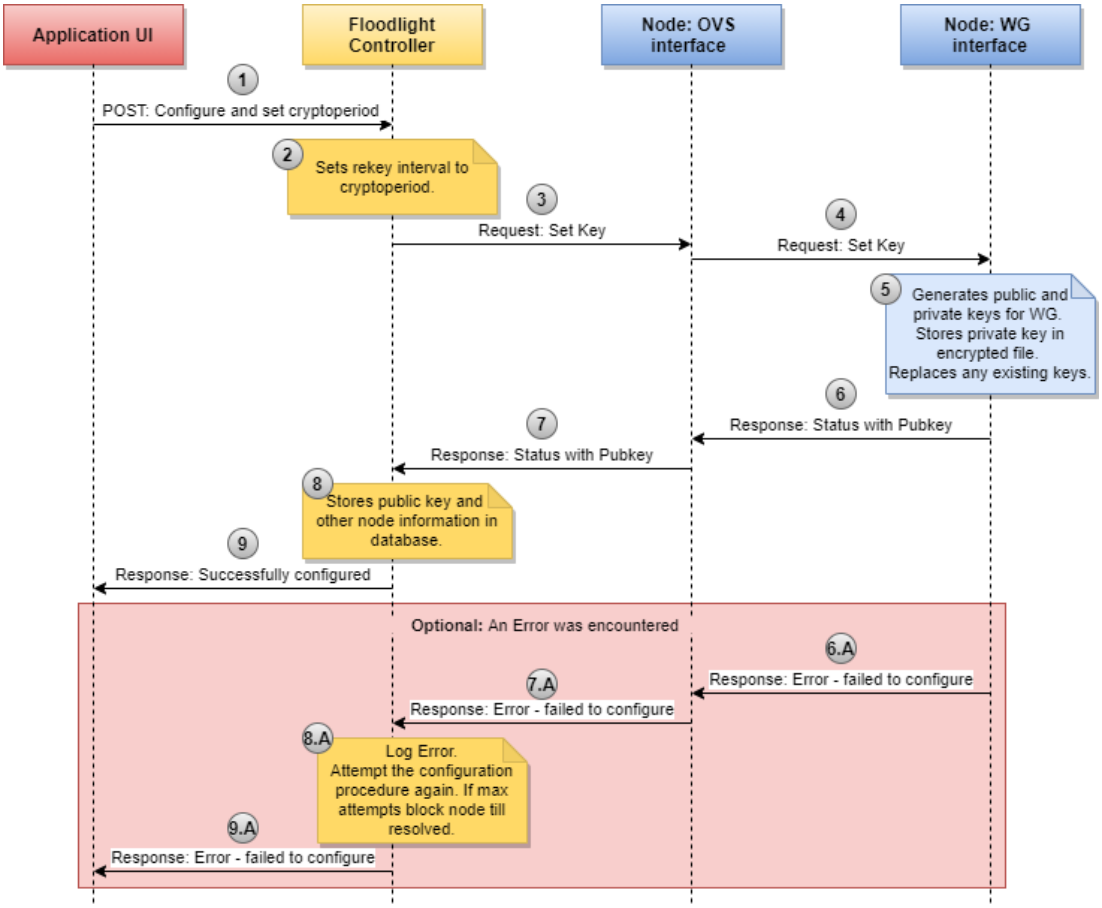

Figure 4.16: Sequence diagram designing the procedure for configuring WireGuard. Steps 1-9 plot the expected sequence of functionality, 6A-9A consider an error case.

Whilst planning the configure procedure it was decided that key generation would take place in the node instead of the controller. This design decision was made deliberately to avoid the security risks associated with transporting private keys. In Braadland's approach (2017) all keys are generated and distributed by the controller, presenting a substantial vulnerability to eavesdropping attacks [Bra17]. Exclusively transporting public keys in DPKM mitigates this risk and improves overall security. The steps for configuring a WireGuard interface can be summarised as follows:

- **Step 1:** Application UI initiates the procedure by making a `POST` request through the REST API connected to the controller. This request contains the identity of the node to configure and specifies a cryptoperiod for rekeying the interface.

- **Step 2:** Controller receives the `POST` request, records the cryptoperiod and sends a set key message to the node's OVS interface over OpenFlow.
- **Steps 3-4:** Set key request is passed from controller to OVS, then translated and passed by OVS to WireGuard as a command for generating and setting the keys.
- **Step 5:** WireGuard generates the 32-byte Curve25519 public and private keys, storing them securely in hidden, encrypted files. Any existing keys are overwritten and destroyed so only one pair of keys is ever stored. The safety of the private key is ensured by the isolation of the node, password protection, and strength of encryption.

  Standard outcome:

- **Steps 6-7:** Status response containing the public key is passed from WireGuard to OVS, and then OVS to the controller over OpenFlow.
- **Step 8:** Status response indicates success of the set key request and WireGuard configuration. Controller adds a new database record for the node or updates an existing record with the new public key.
- **Step 9:** Controller returns response to the application, informing of successful `POST` request and new system state.

  Error outcome:

- **Steps 6A-7A:** Error response passed from WireGuard to OVS to controller, detailing the type of error that was encountered.
- **Step 8A:** Controller logs the error and attempts steps 3-5 again. If the maximum limit of attempts is reached the controller quarantines the node until the administrator addresses the error.
- **Step 9A:** Controller returns error response information application of unsuccessful `POST` request.

4.2.2 Unconfiguring WireGuard

Unconfiguring WireGuard reverses the configure procedure and typically involves nine steps with the same components/interfaces. Figure 4.17 represents the sequence of functionality from the initial `DELETE` request by the UI, to the destruction of key material, and finally status or error response.

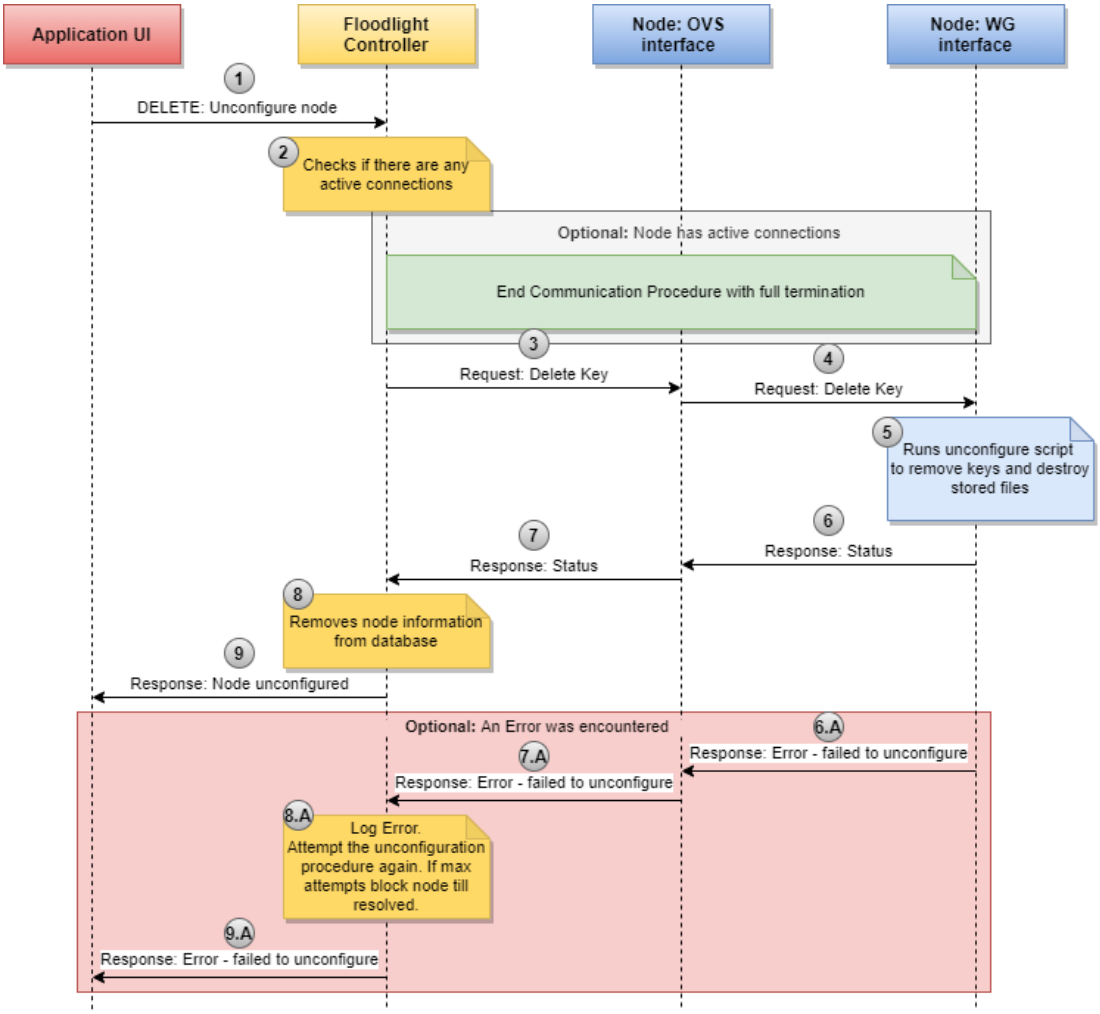

Figure 4.17: Sequence diagram planning the procedure for unconfiguring WireGuard. Steps 1-9 plot the expected sequence of functionality, 6A-9A consider an error case. "End Communication" procedure called if the node has any connections.

As depicted in figure 4.17, the *End Communication* procedure (4.2.6) must be performed during the unconfigure procedure if active peer connections to the node are discovered by the controller at step 2. This design decision prevents packet loss by ensuring no traffic is accidentally routed to an unconfigured node. The steps for unconfiguring a WireGuard interface are as follows, omitting the end communication and error case for brevity:

- **Step 1:** Application UI initiates the procedure by making a `DELETE` request through the REST API connected to the controller with the id of the node to unconfigure.
- **Step 2:** Controller receives the `DELETE` request, checks the database for any peer connections and sends a delete key message to the node's OVS interface over OpenFlow.
- **Steps 3-4:** Delete key request is passed from controller to OVS, then translated and passed by OVS to WireGuard as a command for running the unconfigure script.
- **Step 5:** Unconfigure script instructs the node to remove the keys from the interface and destroys all files containing key material.
- **Steps 6-7:** Status response to delete key request is passed from WireGuard to OVS, and then OVS to the controller over OpenFlow.
- **Step 8:** Status response indicates that WireGuard has been successfully unconfigured. Controller removes all records of the node from the database.
- **Step 9:** Controller returns response to the application, informing of successful `DELETE` request and new system state.

4.2.3 Adding Peers

Establishing an encrypted tunnel between two nodes involves mutually adding the WireGuard interfaces as peers. Much like configuring/unconfiguring WireGuard, the add peer procedure typically requires nine steps and communication between the four system components. The procedure can be initiated with a `POST` request from the application UI containing peer information or internally by the controller. Figure 4.18 presents the sequence of functionality underlying the add peer procedure from an initial request by the application UI, to addition of peer information to the WireGuard interface and controller database, ending with a response indicating success or error.

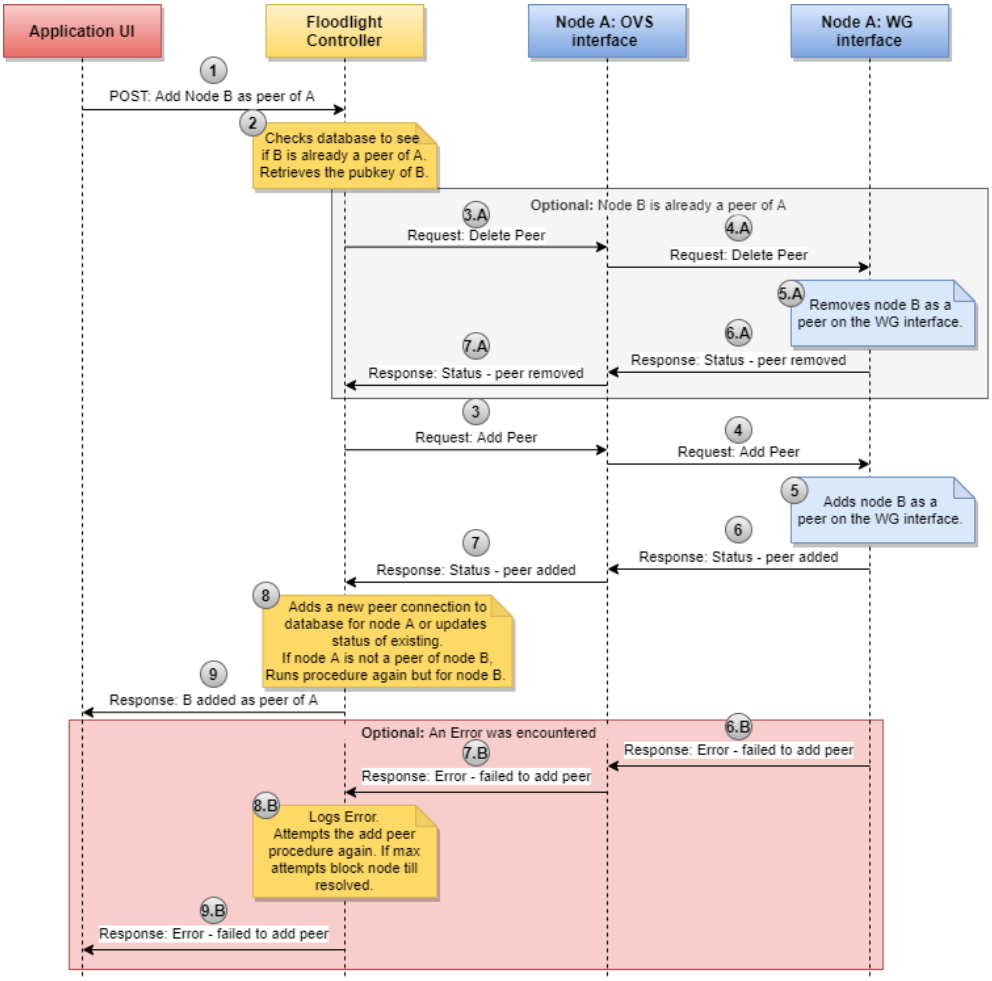

Figure 4.18: Sequence diagram describing the procedure for adding a peer. Steps 1-9 plot the expected sequence of functionality, 3A-7A remove an existing peer, 6B-9B consider an error case.

Maintaining synchronicity of the system state is essential for avoiding concurrency issues and portraying an accurate top-level abstraction of the network topology. With this in mind two design decisions were made regarding the functionality of the add peer procedure. Firstly, the controller must check the database at step 2 for any existing peer connections between the nodes and run a condensed version of the delete peer procedure. This ensures outdated peer information such as an expired public key is removed prior to building the new peer connection. Secondly, the controller must run the add peer procedure again at step 8 with the opposite node if only one interface has been added as a peer. This guarantees all connections are bidirectional and peer information is added to both nodes.

Adding a peer involves the following nine steps, omitting the delete peer case (covered in 4.2.4) and error case (explained in 4.2.1) for brevity:

- **Step 1:** Application makes a `POST` request to the controller over the supported REST API, containing the identity of the node adding the peer (A) and node to be added (B).
- **Step 2:** Controller receives the `POST` request, checks the database for any existing connections and optionally runs the delete peer procedure. Controller constructs and sends an add peer message to node A containing the public key, switch ipv4, and WireGuard ipv4 of target peer B.
- **Steps 3-4:** Add peer request is sent from the controller to OVS over OpenFlow and then translated by OVS into the WireGuard command for adding a new peer with the message data.
- **Step 5:** Node B is added to the peer list of WireGuard interface A, setting the public key for encryption, endpoint destination IP address for sending packets and allowed IP addresses for receiving packets.
- **Steps 6-7:** Status response to add peer request is returned from WireGuard to OVS, and then OVS to controller for processing.
- **Step 8:** If response indicates success, the controller either adds a new peer connection record to the database, marking it as one-way, or updates an existing peer connection to bidirectional. If the connection is one-way, the controller sends another add peer message but to node B with the information of target peer A, subsequently turning the connection into bidirectional.
- **Step 9:** Controller informs application of successful `POST` request and new system state.

4.2.4 Deleting Peers

The delete peer procedure reverses the add peer procedure, tearing down an existing WireGuard tunnel between two peer nodes. As presented in figure 4.19, deleting a peer requires a similar nine step sequence of functionality to the add peer procedure and can be initiated externally by the application or internally by the controller.

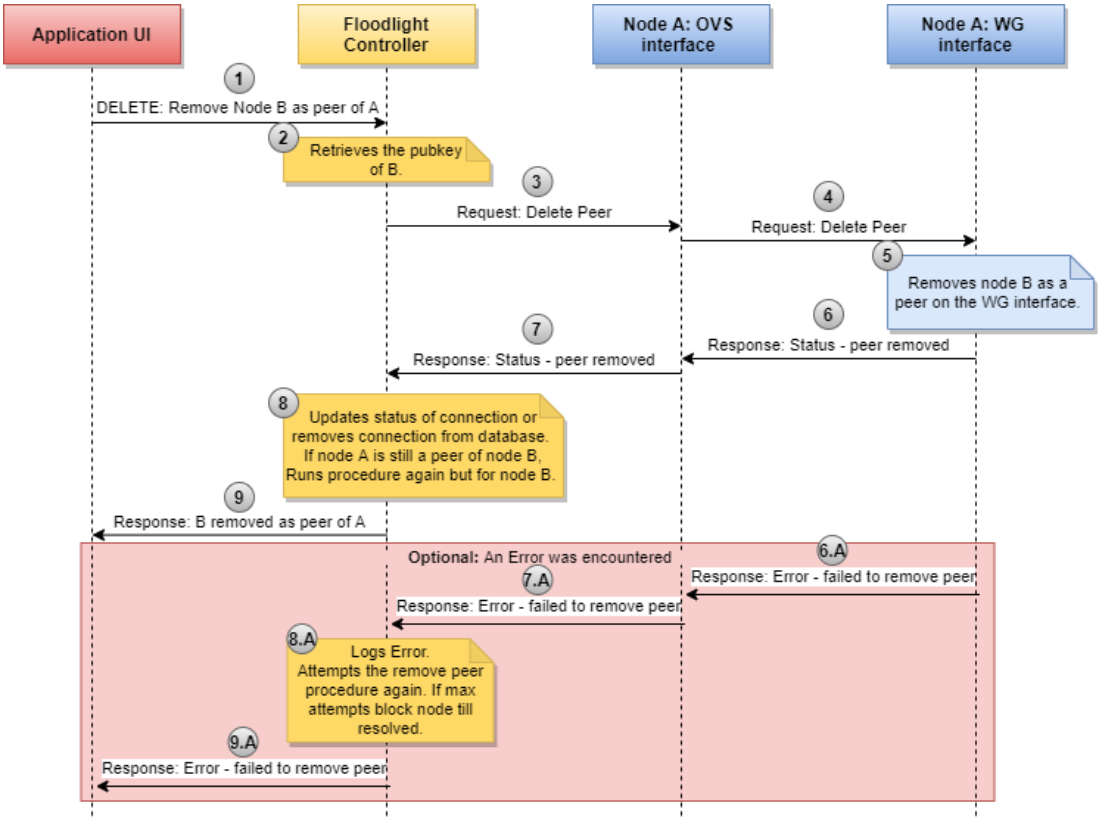

Figure 4.19: Sequence diagram illustrating the procedure for removing a peer. Steps 1-9 plot the expected sequence of functionality, 6A-9A consider an error case.

Synchronicity of the system state is maintained in the delete peer procedure by allowing the controller to perform the procedure twice at step 8 if only one half of the peer connection has been removed. The controller must modify the peer list of both WireGuard interfaces, tracking the state of the connection from bidirectional (both added) to one-way (one peer deleted) and finally removed (both peers deleted). Although not shown in figure 4.19, if two peers are found to be communicating at step 2 the *End Communication* procedure (4.2.6) should be executed by the controller to stop communication before tearing down the tunnel. This design decision prevents packet loss and unexpected interruptions to active communication channels. Deleting a peer requires the following nine steps:

- **Step 1:** Application makes a `DELETE` request to the controller containing the identity of the peer connection to delete.
- **Step 2:** Controller receives `DELETE` request over REST, finds a matching connection between nodes A and B in the database, and constructs a delete peer message for A with the details of B.
- **Steps 3-4:** Delete peer request is sent from the controller to node A (OVS) over OpenFlow and then executed as a WireGuard command, deleting the peer matching the public key in the message data.
- **Step 5:** Node B is removed from the peer list of WireGuard interface A.
- **Steps 6-7:** Status response to delete peer request is returned to controller for processing.
- **Step 8:** Controller either updates the database by marking the connection as one-way, performing the procedure again with the opposite peer, or removes the connection entirely (if already one-way).
- **Step 9:** Controller informs application of successful `DELETE` request and new system state.

4.2.5 Starting Communication

Starting encrypted node-to-node communication in DPKM involves a procedure for setting up switch flow table rules that route traffic through the OVS port containing the WireGuard interface. The procedure requires the tunnel endpoints to be two WireGuard interfaces previously configured by the configuration procedure (4.2.1) and connected by the add peer procedure (4.2.3). The underlying functionality, depicted in figure 4.20, is comprised of eleven steps and message exchanges between two peer nodes with a central controller acting as the mediator.

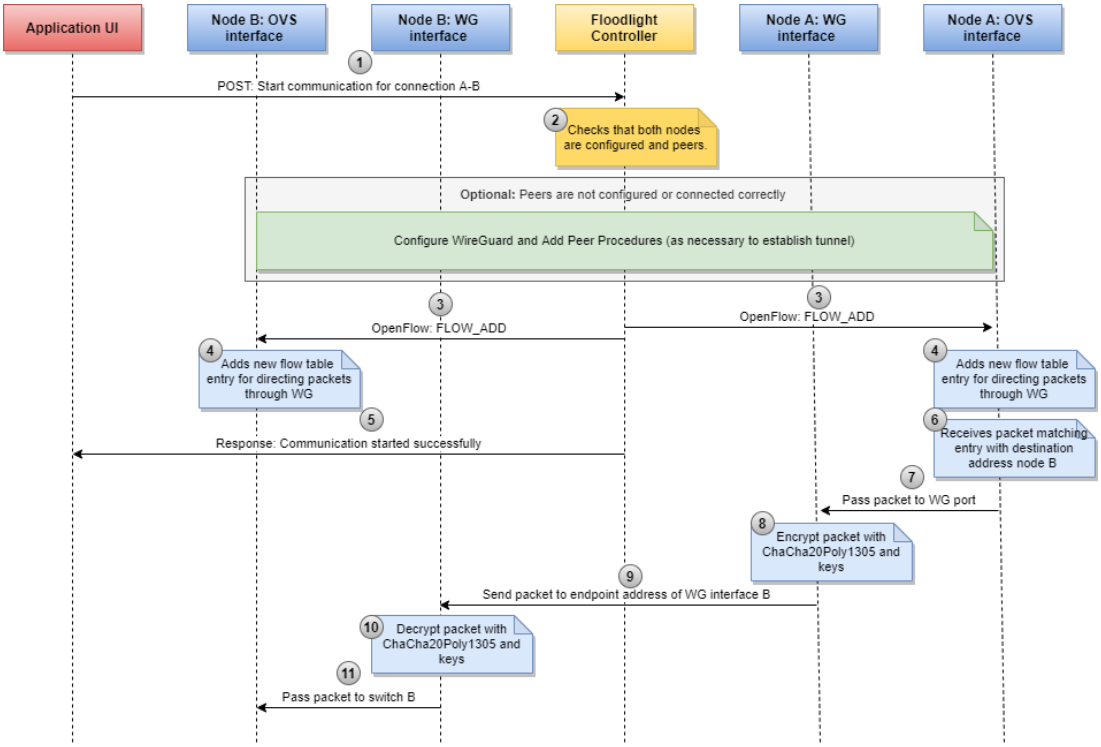

Figure 4.20: Sequence diagram displaying the procedure for starting communication. Steps 1-5 plot the functionality for setting up the flow, 6-11 consider how a packet is handled.

The operation of the sequence diagram in figure 4.20 can be divided into two parts: steps 1-5 establishing the flow rules for routing packets and steps 6-11 demonstrating how a packet is forwarded from switch A to switch B. Starting communication expects preliminary configuration and peer addition of both nodes and will execute the necessary procedures at step 2 before adding new flow rules. This design decision was made to ensure both nodes share the same state and enables the controller to setup both flow tables in parallel. The steps for starting communication are as follows:

Flow Setup:

- **Step 1:** Application makes a `POST` request to the controllers REST API with the identity of the peer connection between A and B wishing to start communication.
- **Step 2:** Controller receives the `POST` request and queries the database checking that both nodes are configured and peers.
- **Step 3:** Controller simultaneously constructs and issues two `flow_add` OpenFlow mes-

sages to the OVS interfaces of nodes A and B. These messages contain new match conditions for the flow table and instructions/actions that should be performed on any matching packets. In this case, the match conditions would specify that any packets received by A with destination address B should be outputted through the WireGuard port (and vice versa).

- **Step 4:** The `flow_add` messages are received by OVS and the flow tables in both nodes are updated to include the new flow entries.
- **Step 5:** Controller informs the application of successful `POST` request and updates the system topology.

Packet Handling:

- **Step 6:** A plaintext packet arrives on node A (OVS component) that matches the new flow table entry and has the destination address of node B.
- **Step 7:** The packet is routed through the port in node A containing the WireGuard interface.
- **Step 8:** The packet is encrypted by WireGuard A using the encryption algorithm ChaCha20Poly1305 with the cryptographic keys (discussed in section 2.3.2).
- **Step 9:** The ciphertext packet is sent over UDP through the network tunnel to the endpoint address and port of peer WireGuard interface B.
- **Steps 10-11:** The packet is decrypted by WireGuard interface B using the same algorithms and passed to OVS interface B.

4.2.6 Ending Communication

Ending encrypted node-to-node communication in DPKM is a procedure that modifies the flow tables of two communicating switches to prevent sending and receiving packets through the WireGuard tunnel. The procedure has two variations: one for ending encrypted communication only, sending all packets unencrypted via the controller, and another for ending all communication between the nodes. In both variations the peer connection is removed entirely by deleting the respective peer entries after modifying the flow conditions. The end communication procedure is an essential part of several other procedures, used as a preliminary operation for unconfiguring WireGuard (4.2.2) and deleting peers (4.2.4), and to mitigate the damage caused by a compromised node in the revocation procedure (4.2.8).

The sequence diagram in figure 4.21 designs the procedure with 6 steps (plus the additional steps of the delete peer procedure), distinguishing the two variations by type A (end encrypted) and type B (end all).

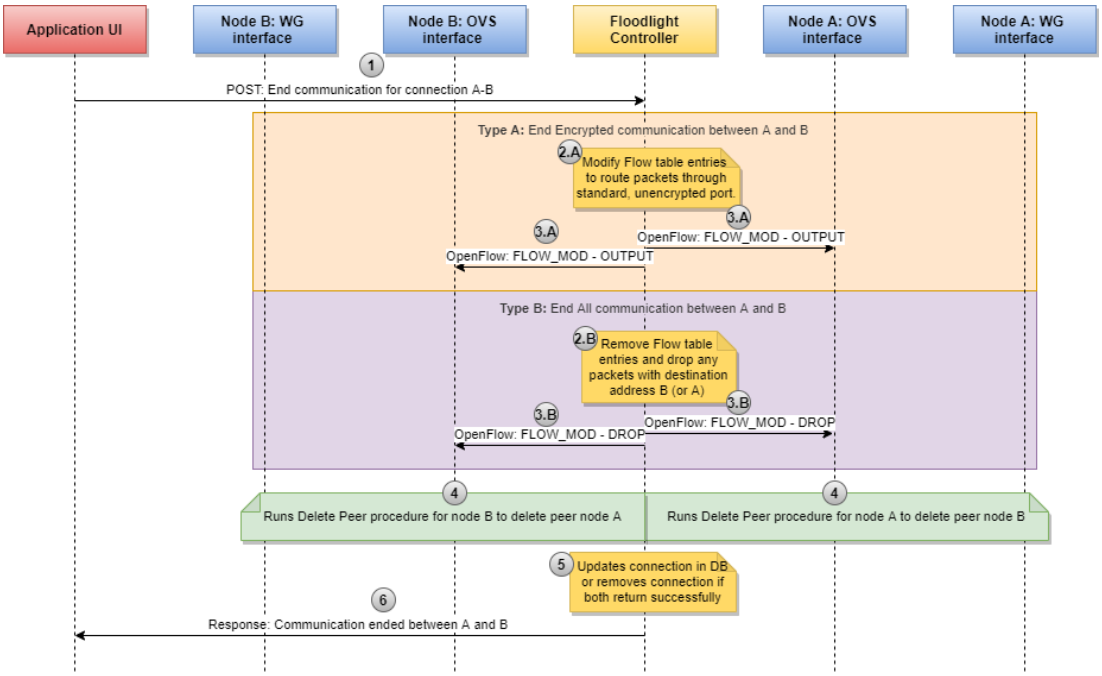

Figure 4.21: Sequence diagram describing two variations of the procedure for ending communication. Type A (orange) reconstructs the flow for unencrypted communication, Type B (purple) terminates all communication. In both variations the peer connection is removed.

When designing the end communication procedure it was important to ensure synchronicity between the nodes by allowing the controller to make simultaneous modifications to the flow conditions set by the *Start Communication* procedure (4.2.5). For type A (end encrypted), the flow table must be modified to output packets destined to the peer through the standard port to the controller as OpenFlow `packet_in` messages. The packets are then processed and passed to the destination node unencrypted as OpenFlow `packet_out` messages. For type B (end all), the flow table entry must be modified to drop any packets destined to the peer, preventing any communication from being possible. Designing two variations of the procedure offers versatility to ending communication and applicability for different situations. For example, ending encrypted communication only allows live updates to the WireGuard configuration and ending all communication assists in security critical situations. The underlying functionality of the end communication procedure consists of the following steps:

- **Step 1:** Application makes a `POST` request to the controllers REST API instructing it to end communication for an active peer connection. This request contains the identity of the connection and the variation of the procedure to perform.
- **Steps 2A-3A (Type A):** Controller simultaneously constructs and sends two `flow_mod` OpenFlow messages to the OVS interfaces of the communicating nodes. These messages modify the existing flow table entry to direct all packets destined to the peer through the standard port to the controller.
- **Steps 2B-3B (Type B):** Controller constructs and sends two different `flow_mod` OpenFlow messages that instruct the OVS interfaces to drop any packets destined to the peer.
- **Steps 4-5:** The delete peer procedure is performed with both nodes, tearing down the WireGuard tunnel. The controller modifies the peer connection in the database based on the status response confirming the state of the nodes.
- **Step 6:** Controller responds to applications `POST` request with success or the error that was encountered, updating the view of the system topology.

4.2.7 Rekeying

During the configuration of a WireGuard interface (4.2.1) a lifespan is given to the generated keys called a cryptoperiod. After expiry of this cryptoperiod the controller automatically initiates a rekeying procedure that pauses encrypted communication with the node, reconfigures the interface with a new set of keys, and rebuilds any former peer connections. The sequence diagram in figure 4.22 designs the rekeying procedure with six steps (and the additional steps of the configuration procedure).

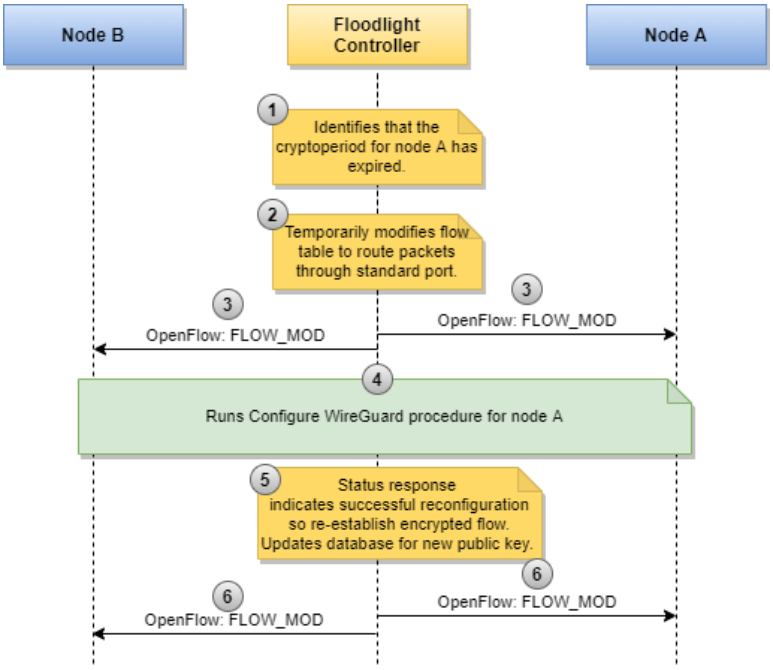

Figure 4.22: Sequence diagram planning the procedure for rekeying the WireGuard interface after expiry of the set cryptoperiod.

Rekeying heightens the overall security of DPKM by keeping a level of freshness that limits the usability of any intercepted keys and prevents brute force attempts at deducing active keys. The procedure takes design aspects from the *End Communication* procedure (4.2.6) by using a similar `flow_mod` message to temporarily make communication unencrypted whilst the keys and peer connections are replaced. In theory this design decision could be interpreted as a security issue since any packets intercepted during this period are plaintext and readable by an attacker. However, the unencrypted time window would be negligible, affecting a very small number of packets, and interception would not compromise the security of any larger part of the system.

Braadland (2017) made an alternative design decision by specifying use of a `flow_mod` queue action that temporarily queues any packets, theoretically maintaining encryption [Bra17]. However, through the background research of this project it was discovered that such a queue action does not exist and the concept of queues in OpenFlow was clearly misunderstood by Braadland. Although queues can be used for rate-limiting packets they cannot reduce the intake rate to zero and must be setup in advance [Iza18], meaning dynamically adjusting

the packet rate is infeasible. The design decision to temporarily continue communication unencrypted is therefore a possible and realistic option for developing the rekeying procedure. The sequence of functionality for the procedure is as follows:

- **Step 1:** Controller compares the set cryptoperiod of a WireGuard interface to the current time of the system and determines that the keys have expired.
- **Steps 2-3:** Controller constructs and sends an OpenFlow `flow_mod` message to the node and any peers, routing packets unencrypted through the normal port.
- **Step 4:** Controller runs the configure WireGuard procedure to reconfigure the node.
- **Steps 5-6:** Controller updates the database with the new keys upon the successful response of the reconfiguration procedure. Following this, the controller constructs and sends an OpenFlow `flow_mod` message to the node and any peers, routing packets through the WireGuard interface.

4.2.8 Key Revocation

Security threats and compromise of entities is a possibility in any system meaning a procedure is essential for mitigating the damage and returning to a normal state of operation. As considered in use case three (3.2) and scenario three of the mock application UI (4.1.3), an administrator needs to be made aware of a compromise and have a method for handling the revocation of corrupted keys. Figure 4.23 presents a sequence diagram with the preliminary actions addressing the compromise (steps 1-5) and then two variations for handling revocation, one reconfiguring the node (type A) and another fully terminating the node (type B).

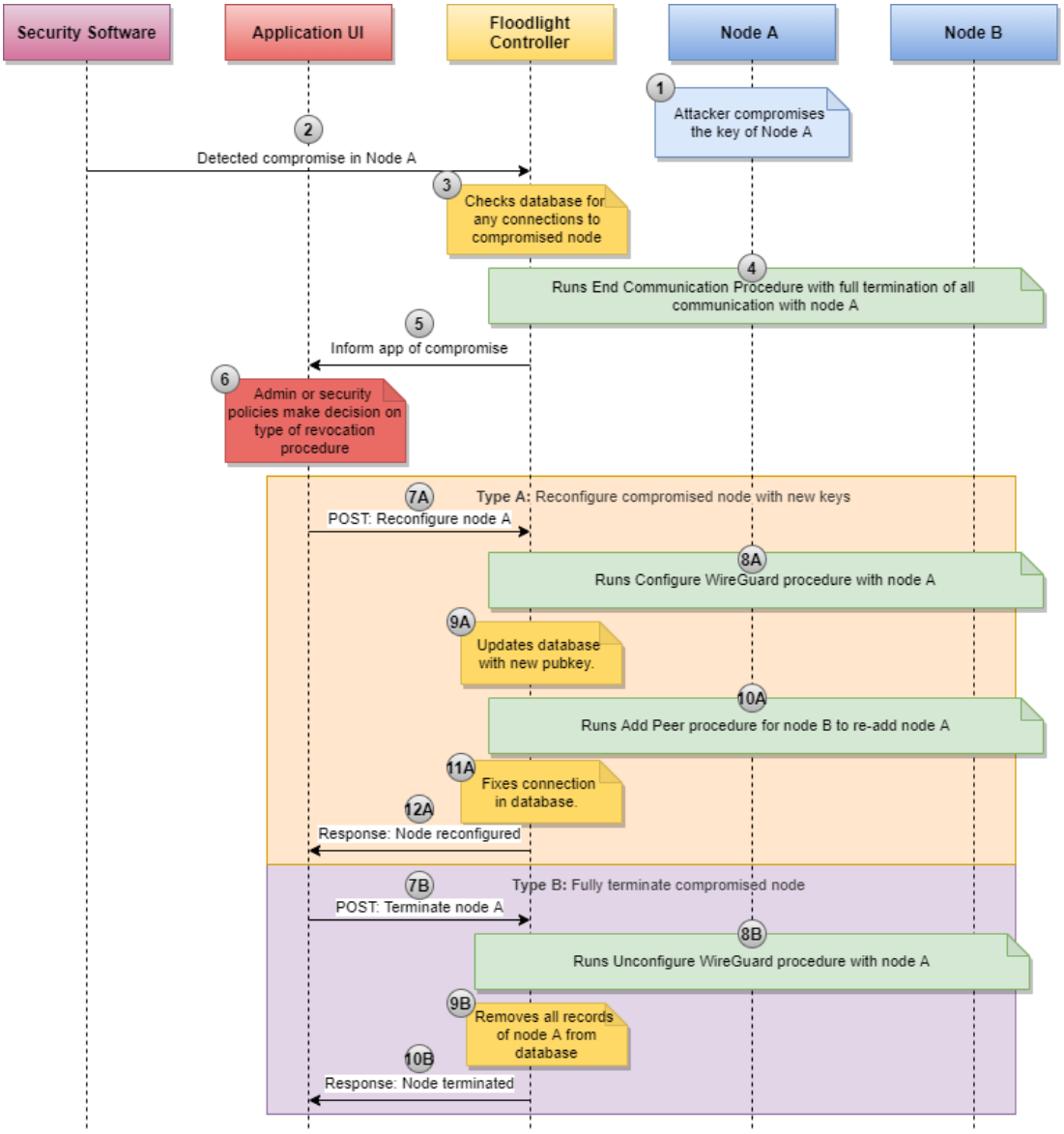

Figure 4.23: Sequence diagram depicting two variations of the revocation procedure after detection of a compromise by security software. All communication is ended with the compromised node and then the node is either reconfigured (type A) or fully terminated (type B).

When designing the revocation procedure it is assumed that the compromise is initially detected by some external security software entity such as an intrusion detection system (IDS) or malware scanner. The realisation of such a component is beyond the scope of this thesis since the primary focus is on facilitating key management. Nevertheless, for testing purposes DPKM should include functionality to simulate a compromise by choosing a node to detect as "com-

promised" through the application UI (implemented in 5.4.8). The design decision was made to run the terminate all communication variation of the *End Communication* procedure (4.2.6) with the node upon immediate detection of the threat. This approach was taken by Braadland (2017) and utilised for DPKM because it allows rapid isolation of the compromised node by dropping all packets [Bra17]. This prevents any window of opportunity for malicious activity between detection and an administrator taking action.

The key revocation procedure is essentially composed of several other procedures previously discussed in this section. In type A, the *Configure WireGuard* procedure (4.2.1) is used for reconfiguring a compromised node with a new set of keys and the *Add Peer* procedure (4.2.3) is used for rebuilding the peer connections. In type B, the *Unconfigure WireGuard* procedure (4.2.2) is used for removing all keys and excluding the node from the DPKM system. The sequence of functionality comprising the revocation procedure is as follows:

Preliminary:

- **Steps 1-2:** Security software entity detects the compromise of the private key of node A and informs the controller.
- **Steps 3-4:** Controller marks the node as compromised, queries database for any peer connections with the node and runs the end communication procedure to terminate all communication with A.
- **Steps 5-6:** Controller updates top-level system status to "action needed" which alerts an administrator through the application UI or initiates some automated security policy in the application.

Type A (Reconfigure):

- **Step 7A:** Application makes `POST` request to the controller requesting reconfiguration of node A.
- **Steps 8A-9A:** Controller runs the full steps of the configure WireGuard procedure, updates the database for the new public key, and sets compromised field to false.
- **Step 10A-11A:** Controller runs the add peer procedure and fixes the connection status in the database.
- **Step 12A:** Controller informs application of successful reconfiguration and changes system status from "action needed" to "active".

Type B (Terminate):

- **Step 7B:** Application makes `POST` request to the controller requesting termination of node A.
- **Steps 8B-9B:** Controller runs the unconfigure WireGuard procedure, deletes all database records for the node, and quarantines the node from the system until an administrator explicitly releases it.
- **Step 10B:** Controller informs application of successful termination and updates top-level system status.

4.3 Messages

Performing the key management capabilities expected of DPKM requires extensions to the OpenFlow protocol to support a new set of messages. The background research established that standard OpenFlow does not provide the necessary message types for DPKM but offers a flexible experimenter type for additional vendor functionality (2.3.1). The architectural designs (4.1) and procedures (4.2) presented in this chapter have given an impression of the responsibilities of the system components and expected sequences of functionality. The final part of the design phase is defining the OpenFlow messages that need to be developed for communication between the controller and nodes within each procedure. This section divides the required DPKM messages into three types: *Controller-to-switch*, *Status response*, and *Error response*. Controller-to-switch messages are constructed by the controller and sent over OpenFlow to the switch to perform DPKM functions and make changes to the node. Status response and error response messages are constructed by the switch and returned to the controller to indicate the success or failure of a controller-to-switch message.

The required messages are presented in two tables displaying the controller-to-switch and status message types in table 4.1 and the error response codes in table 4.2, with the corresponding purpose of each. From these tables the messages were developed in the *Extending OpenFlow* section (5.2) of the subsequent implementation chapter (5).

Message	Function
DPKM_SET_KEY (0)	Controller-to-switch message sent during the *Configure WireGuard* procedure (4.2.1). Instructs the node to generate new cryptographic keys and bind them to the WireGuard interface.
DPKM_DELETE_KEY (1)	Controller-to-switch message sent during the *Unconfigure WireGuard* procedure (4.2.2). Requests that the node deletes all its cryptographic keys, resulting in a blank WireGuard interface.
DPKM_ADD_PEER (2)	Controller-to-switch message sent during the *Add Peer* procedure (4.2.3). Instructs the node to add a new peer entry to the WireGuard interface. The message contains the public key, switch ipv4, and WireGuard ipv4 of the peer to be added. This information is appended to the WireGuard interface upon receival of the message.
DPKM_DELETE_PEER (3)	Controller-to-switch message sent during the *Delete Peer* procedure (4.2.4). Commands the node to remove an existing peer entry from the WireGuard interface. The message contains the public key of the peer to be deleted.
DPKM_GET_STATUS (4)	Controller-to-switch message sent optionally to gauge the current state of a node. Makes no changes to the system state itself.
DPKM_STATUS (5)	Status response message constructed by the switch and sent to the controller. Used in response to all previous messages, informing of the current state of a data plane node. Can contain the public key, switch ipv4, WireGuard ipv4, and additional state information. Fields are populated based on the previous message request.

Table 4.1: The OpenFlow experimenter message extensions needed for DPKM to operate.

Error Message	Reason
DECODE_SET_KEY (0)	Unable to process a SET_KEY message.
DECODE_DELETE_KEY (1)	Unable to process a DELETE_KEY message.
DECODE_ADD_PEER (2)	Unable to process an ADD_PEER message.
DECODE_DELETE_PEER (3)	Unable to process a DELETE_PEER message.
DECODE_GET_STATUS (4)	Unable to extract the status of the node.
EXECUTE_SET_KEY (5)	Failed during execution of script for configuring the WireGuard interface.
EXECUTE_DELETE_KEY (6)	Failed during execution of script for unconfiguring the WireGuard interface.
EXECUTE_ADD_PEER (7)	Failed during execution of script for adding a new peer.
EXECUTE_DELETE_PEER (8)	Failed during execution of script for removing an existing peer.
GET_KEY (9)	Could not retrieve the public key from WireGuard interface.
GET_IP_S (10)	Could not retrieve the ipv4 address of the switch interface.
GET_IP_WG (11)	Could not retrieve the ipv4 address of the WireGuard interface.
MISSING_KEY (12)	ADD/DELETE_PEER message is missing the public key field.
MISSING_IP_S (13)	ADD/DELETE_PEER message is missing the ipv4 switch field.
MISSING_IP_WG (14)	ADD/DELETE_PEER message is missing the ipv4 WireGuard field.

Table 4.2: The required OpenFlow error message extensions reporting issues during DPKM operations.

Chapter 5

Implementing DPKM

This chapter documents the transformation of the conceptualised Data Plane Key Management (DPKM) system into a fully functional product. The implementation involved the extension of four codebases – Floodlight, Floodlight-webui, Loxigen, and Open vSwitch, requiring approximately 7500 lines of code. A system was produced that enhances the capabilities of SDN, enabling remote establishment of encrypted WireGuard-based communication flows and facilitation of the key management phases (2.2.2) through the extended OpenFlow protocol. Most of the specified functionality was delivered with a well-rounded analysis provided in the subsequent evaluation chapter (6).

This chapter begins with section 5.1 where the system setup and development environment is described. Next, section 5.2 investigates the process of extending OpenFlow, studies the experimenter structure and presents the DPKM extensions to the protocol. Following this, section 5.3 provides an overview of the software extensions to supplement the readers understanding of the final product. Thereafter, section 5.4 discusses the bulk of the implementation, dividing the system by procedure, and traversing the sequence of functionality from initiation by the application UI to mediation by the controller to execution in the node. Finally, section 5.5 presents the testing undertaken to substantiate the validity of the implemented product.

5.1 System Setup

The full DPKM system architecture involves the setup of three Ubuntu 18.04 virtual machines (VMs) with VirtualBox [Vir20] – one to host the Floodlight controller, application UI and MySQL database, and two to host network nodes A and B, with one OVS switch and WireGuard interface each. From this point onwards *"VM1"* will refer to the controller VM and *"VM2/3"* will refer to node's A/B respectively. Figure 5.1 outlines the system setup using a bridged networking approach where each VM is connected to the VirtualBox host computer via a bridged network adapter. The host computer provides internet connectivity and assigns a unique IP address to each VM to enable simulation of an SDN. The dashed line represents the internal DPKM network which connects VM2/3 to VM1 via OpenFlow and VM2 to VM3 through the WireGuard tunnel. VM2/3 contain an OVS switch (s1/2) with a WireGuard interface and dummy host integrated into the ports. The controller inherits the IP address of VM1, and the switches inherit the IP addresses of VM2/3. For brevity this section presents a condensed version of the system setup, with a full manual provided in appendix B for replication.

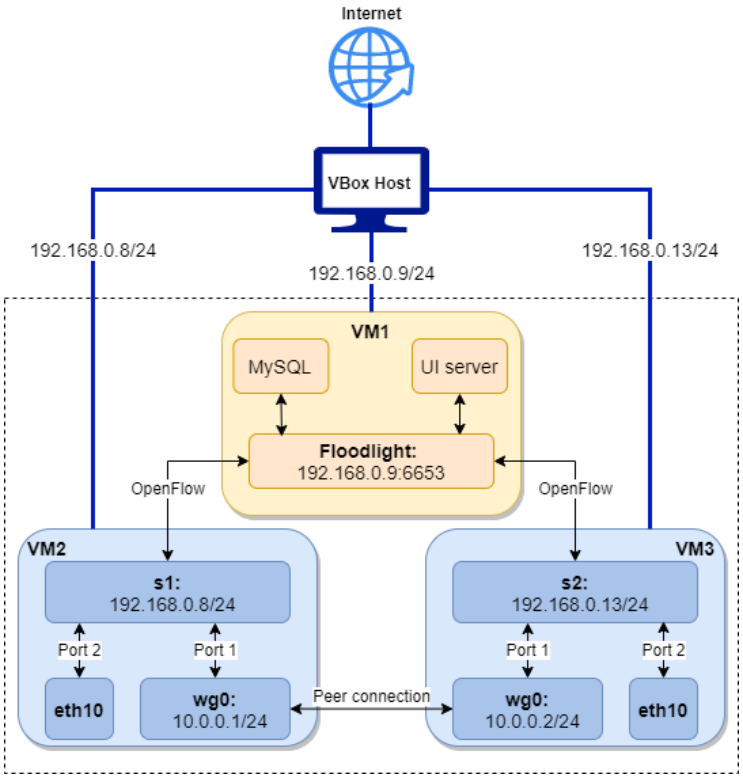

Figure 5.1: Diagram of the full DPKM network topology, consisting of three VMs structured as a bridged network with unique IP addresses for simulating an SDN.

Development was carried out simultaneously across VM1 and VM2, using GitHub [Git20] repositories for version control and pushing changes from VM2 to VM3. Overall the implementation of DPKM required the following four git repositories[1,2,3,4]:

- **DPKM_Floodlight (VM1):** a clone of the Floodlight controller repository that enabled development on the code base for DPKM extensions. The clone was added to the eclipse IDE [Fou20] as a new java project, enabling modification and testing of the controller software.
- **DPKM_Floodlight-webui (VM1):** a clone of the Floodlight web UI repository used for extending the application UI with additional pages, functionality, and styling for DPKM. The clone was opened as an Atom [Ato20] text editor project and is a submodule of the DPKM_Floodlight repository.
- **Loxigen (VM1):** a clone of the floodlight/loxigen repository that enabled generation of DPKM extensions to the OpenFlow protocol for the Floodlight controller. (discussed in 5.2).
- **DPKM_OVS (VM2/3):** a clone of the Open vSwitch repository that allowed a new DPKM variation of an OVS switch to be defined by extending the C code base in Atom. Changes were pushed from VM2 and pulled on VM3 to simultaneously build and test the system with two network nodes.

Setting up the development environment for VM1 was a relatively easy process. After creating the two new workspaces, one for the controller in eclipse and one for the UI in Atom, the next task was creating the database previously specified in figure 4.4 of the design section. MySQL was used to create a database named cntrldb with the three tables ConfiguredPeers, CommunicatingPeers, and ErrorLog (described in 4.1.1). Since the controller is built in java the Java DataBase Connectivity API (JDBC) was used to provide Floodlight with dynamic access to cntrldb. JDBC provides a DriverManager class for creating a connection object from a URL, username and password, enabling SQL queries to be performed [Ree00]. Floodlight was given its own account in cntrldb, with the credentials stored in `db.properties` (5.1).

[1] https://github.com/lukehengstenberg/DPKM_Floodlight
[2] https://github.com/lukehengstenberg/DPKM_Floodlight-webui
[3] https://github.com/lukehengstenberg/loxigen
[4] https://github.com/lukehengstenberg/DPKM_OVS

```
db.url = jdbc:mysql://localhost:3306/cntrldb?autoReconnect=true
db.user = floodlight
db.password = FLPass878876#
```

Listing 5.1: MySQL database credentials providing access to Floodlight.

The ConnectionProvider class was defined to create and return the connection object from the credentials (listing 5.2).

```
public class ConnectionProvider {
  protected static Logger log = LoggerFactory.getLogger(DpkmConfigureWG.class);
  static Properties prop = getCredentials();
    private static final String url = prop.getProperty("db.url");
    private static final String user = prop.getProperty("db.user");
    private static final String password = prop.getProperty("db.password");
    public static Connection getConn() {
      try {
        return DriverManager.getConnection(url, user, password);
      } catch(Exception e) {
        log.error("Failed to get connection to db.");
        return null;
      }
    }
    private static Properties getCredentials() {
    Properties prop = new Properties();
    try (FileInputStream in = new FileInputStream("src/main/resources/db.properties")){
      prop.load(in);
    } catch (IOException e) {
      log.error("Failed to access properties file.");
    }
    return prop;
  }
}
```

Listing 5.2: ConnectionProvider class for creating and returning a connection object to the DPKM database.

Configuring VM2/VM3 was a more complex process since the OVS and WireGuard technologies are not naturally compatible. After cloning the OVS repository and setting up the development environment in atom a sequence of commands were needed for setup/teardown of the interfaces. The arduous process of entering these commands to restart OVS after every modification to the code led to creation of the bash script in listing 5.3 that can be run with a single `./SetupTestNetworkWG.sh` command. The script begins by starting the OVS daemons and servers and deleting the existing OVS and WireGuard interfaces to restore the node to a blank state. A new OVS switch interface named s1 is created and assigned the IP address

192.168.0.8/24 (VM2 address). A new unconfigured WireGuard interface named wg0 is created and assigned the IP address 10.0.0.1/24. Wg0 and an existing host eth10 are added as ports to switch s1. The OpenFlow connection to the controller is then set by providing the controller IP address and listen port – tcp:192.168.0.9:6653 (VM1 address). Finally, the interfaces are started with an error log for debugging. The setup script differs slightly in VM3, with different IP addresses 192.168.0.13/24 (s2) and 10.0.0.2/24 (wg0).

```
echo "Automatic setup script of OVS test environment."
echo "Starting ovs server, switch and managers."
/usr/local/share/openvswitch/scripts/ovs-ctl start
echo "Tearing down old bridge (if exists)."
ovs-vsctl --if-exists del-br s1
echo "Removing old WG interface."
ip link set wg0 down
ip link delete wg0
echo "Creating a switch."
ovs-vsctl add-br s1
echo "Adding Unconfigured WireGuard interface."
ip link add dev wg0 type wireguard
ip addr add 10.0.0.1/24 dev wg0
echo "Configuring switch address and ports."
ovs-vsctl add-port s1 wg0
ovs-vsctl add-port s1 eth10
ip addr add 192.168.0.8/24 dev s1
echo "Setting up OpenFlow controller."
ovs-vsctl set-controller s1 tcp:192.168.0.9:6653
ovs-vsctl show
echo "Configuring the network."
ip link set wg0 up
ip link set eth10 up
ip link set s1 up
ovs-ofctl show s1
ovs-appctl vlog/set warn
echo "Setup complete."
```

Listing 5.3: Bash script performing a series of commands for setting up a network node.

Documenting the establishment of TLS/SSL encryption for the OpenFlow channel, and HTTPS for the REST northbound connection was beyond the scope of this thesis (1.2). However, the subsequent chapters maintain the assumption that these security measures have been put in place. The process is supported by Floodlight, providing multiple key stores and tools for managing keys and certificates. Instructions are provided on the official Floodlight website[5], and will be followed when deploying a stable version of DPKM.

[5] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/5636122/

5.2 Extending OpenFlow

Communication between the controller and node in DPKM requires an extended version of the OpenFlow protocol with support for the new functionality designed in chapter 4. OpenFlow is a flexible protocol that offers a multitude of experimenter object types for extending most operational aspects. Extensions to OpenFlow are well-documented in the state-of-the-art. For example, Snow et al. (2018) use OpenFlow experimenter and Loxigen to introduce new packet matching conditions [SDG18], and Mizrahi and Moses (2013) suggest time-based configuration updates for OpenFlow [MM13]. The proposed extensions in existing research, alongside the OpenFlow experimenter designs conceptualised in Braadland's thesis (2017), provided a basis for developing the extensions for DPKM [Bra17].

OpenFlow extensions must be structured in accordance to the OpenFlow experimenter structure and can be realised with Loxigen. Loxigen is a tool produced by the creators of Floodlight that enables the generation of OpenFlow libraries for Java, Python and C, and greatly simplifies OpenFlow development through abstraction of low-level tasks such as serializing/deserializing messages [Lox20]. Adding new OpenFlow objects to the controller can be done by modifying the codebase used by Loxigen to generate the Floodlight OpenFlow library called OpenFlowJ-Loxi. These changes must then be accepted by the switch by making matching extensions to the OVS codebase.

The OpenFlow Experimenter structure is investigated in 5.2.1, using several experimenter headers to define a set of DPKM headers. Following this, the implemented extensions are documented in 5.2.2 and software integration is discussed in 5.2.3, using Loxigen to generate a new OpenFlowJ-Loxi library for Floodlight. The extensions defined in this section are used in the subsequent sections of this chapter for implementing the procedures and developing a functional DPKM system.

5.2.1 OpenFlow Experimenter Headers

The OpenFlow Experimenter structure provides a message space for vendors to stage features not yet included in the protocol and extend the capabilities of OpenFlow-based switches [ONF13]. Most OpenFlow object types supply an experimenter header and share numerous compulsory fields for integration into a live system. Table 5.1 displays several OpenFlow object types with the corresponding header supporting additional functionality. Only a select

few – message, match, error – have been used in the implementation of DPKM and will be discussed further. Table 5.2 presents the typical fields of an experimenter header with an explanation of each adapted from version 1.4 of the OpenFlow Switch Specification and study of the Loxigen codebase [ONF13; Lox20]. These fields are a requirement in every DPKM experimenter object as exhibited in the subsequent code listings.

Object Type	**Experimenter Header**	**Purpose**
Message	of_experimenter	Adding new messages with the potential to perform functionality beyond OpenFlow through definition of new fields and inclusion of new objects.
Match	of_oxm	Defining OpenFlow Extensible Match (OXM) fields for novel packet matching conditions in the switch flow table.
Action	of_action_experimenter	Establishing new actions to be performed on packets matching flow table entries.
Instruction	of_instruction_experimenter	Extending the available instructions that can be applied to packets matching flow table entries.
Queue	of_queue_prop_experimenter	Increasing Quality-of-Service (QoS) capabilities of queues by adding new statistical messages.
Meter	of_meter_band_experimenter	Defining messages for modifying meters to support some additional QoS functionality.
Error	of_experimenter_error_msg	Adding new error messages for reporting on issues in the extended OpenFlow/SDN architecture.

Table 5.1: OpenFlow object types with the corresponding experimenter header and purpose.

Field	**Size**	**Role**
Type	16 bit	Used persistently in all OpenFlow structures to classify messages or object by a super type. Experimenter messages are classed as type 4 – OFPT_EXPERIMENTER. Experimenter objects are either type 65535 (actions, instructions) or type 0xffff (stats).
Length	16 bit	Used throughout OpenFlow to store the total length of structures (sum of header and fields).
Experimenter	32 bit	Identifies the organisation behind the OpenFlow experimenter extensions and enables grouping of different experimenter structures under one package. The experimenter field is either a zero-byte followed by the registered IEEE OUI of an organisation or a value assigned by the Open Networking Foundation (ONF). Since DPKM is deployed on a private network a unique randomly generated id (0xa20a0323) was used. In future, a valid experimenter id would need to be obtained for DPKM to operate in industry.
Subtype	32 bit	Further classifies OpenFlow objects into groups within the type and experimenter package. Can be freely chosen and is used to define each DPKM object, for example of_dpkm_set_key message is subtype 0 of type 4 and experimenter 0xa20a0323.

Table 5.2: Typical fields in an OpenFlow Experimenter header.

From the preliminary designs it was understood that OpenFlow message and error extensions were essential for facilitating key management operations. Action objects were also considered as a possible approach to interaction with the WireGuard interface. Code listing 5.4 depicts the three DPKM headers developed in the Loxigen class `dpkm.c` for establishing new messages, actions, and errors. Defining a new set of OpenFlow headers means production of cleaner code and better subdivision of DPKM structures. Rather than extending the `of_experimenter` header directly each message can be grouped under `of_dpkm_header` and classified by a unique subtype. This means the correct experimenter id is always applied and allows the switch/controller to distinguish captured DPKM messages. The fields discussed in table 5.2 are used with the experimenter type (4/65535/0xffff) and the DPKM experimenter id (0xa20a0323). The question mark for subtype implies that the structures are expecting extension from subsequent objects.

```
// DPKM extension message header
struct of_dpkm_header : of_experimenter {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == ?;
};

// DPKM extension action header
struct of_action_dpkm : of_action_experimenter {
    uint16_t type == 65535;
    uint16_t len;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == ?;
    pad(4);
};

// DPKM error message header
struct of_dpkm_base_error : of_experimenter_error_msg {
    uint8_t version;
    uint8_t type == 1;
    uint16_t length;
    uint32_t xid;
    uint16_t err_type == 0xffff;
    uint16_t subtype == ?;
    uint32_t experimenter == 0xa20a0323;
    of_octets_t data;
};
```

Listing 5.4: DPKM OpenFlow Experimenter headers defined in the Loxigen class dpkm.c.

5.2.2 Defining OpenFlow Extensions

Implementing the DPKM OpenFlow extensions can be plainly explained in four stages. First, the classes for the new objects are defined within the Loxigen codebase. Secondly, an OpenFlowJ-Loxi library is generated from the classes, replacing the existing jar files in the Floodlight project workspace. Thirdly, a set of identical extensions are implemented in the OVS codebase. Finally, the Floodlight and OVS software is extended with processes using the extensions. The first three of these stages will be covered in this subsection and 5.2.3, with the fourth stage discussed in section 5.4.

Message Extension

DPKM message extensions were proposed in section 4.3, consisting of six new messages for key management related communication between the controller and switch. Each message extends the `of_dpkm_header` defined in the previous section and is assigned a subtype from 0-5 to distinguish between them. Originally, implementation of the message structures proposed by Braadland (2017) was attempted but design issues and misconceptions of functionality were quickly discovered [Bra17]. Braadland proposed the use of time length value (TLV) data structures for storage of key, status, and error data in messages. Although hypothetically possible, this is a feature of multipart messages which are unnecessarily complex to implement and not crucial for DPKM to function. No key management message needs to exceed 1KB, meaning the 64KB single message capacity provides sufficient storage.

Originally, experimenter actions were developed for each key management function to be included in DPKM messages (hence the `of_action_dpkm` header). However, on further analysis it was clear that new actions are easily defined but could not be used for the purpose of modifying WireGuard. These realisations meant the DPKM message structure was kept simple with limited fields and no inclusion of actions. Three messages – `of_dpkm_set_key` (listing 5.5), `of_dpkm_delete_key` (listing 5.6), and `of_dpkm_get_status` (listing 5.7) – were implemented with identical fields to the DPKM message header aside from defining the subtype and containing some padding. No additional data is needed since the messages act as instructions for the node to configure WireGuard, unconfigure WireGuard, and get the status respectively.

```
struct of_dpkm_set_key : of_dpkm_header {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == 0;
    pad(8);
};
```

Listing 5.5: DPKM_SET_KEY OpenFlow Experimenter message for configuring WireGuard defined in the Loxigen class dpkm_set_key.c.

```
struct of_dpkm_delete_key : of_dpkm_header {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == 1;
    pad(8);
};
```

Listing 5.6: DPKM_DELETE_KEY OpenFlow Experimenter message for unconfiguring WireGuard defined in the Loxigen class dpkm_delete_key.c.

```
struct of_dpkm_get_status : of_dpkm_header {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == 4;
    pad(8);
};
```

Listing 5.7: DPKM_GET_STATUS OpenFlow Experimenter message for extracting the node status defined in the Loxigen class dpkm_get_status.c.

To add a peer the `of_dpkm_add_peer` message (listing 5.8) was implemented with three additional fields – `key, ipv4_addr, ipv4_wg` – for distributing peer information to the switch. Transporting larger amounts of information, such as the 32-byte public key, posed a challenge since standard OpenFlow fields are a maximum of uint64 or 64-bits. Initially an approach was taken to divide the key into eight 32-bit segments and populate the message as an array or list object. However, the limitations of either the technology, developer's knowledge, or both meant no success in this approach.

An unconventional solution was formulated by using the `desc_str` and `serial_num` char array datatypes that can store 256-bytes and 32-bytes respectively. These datatypes are used in standard OpenFlow for returning descriptive information about the switch to the controller. It was discovered that they provided enough bytes of storage for the key and ipv4 addresses, preventing the difficulty of implementing a new OpenFlow type. One limitation of this approach is that although the `serial_num` field can store 32-bytes the full size of the key string is marginally bigger, meaning the 256-byte `desc_str` field must be used. This leaves empty bytes and makes the total message size larger than necessary.

```
struct of_dpkm_add_peer : of_dpkm_header {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == 2;
    pad(8);
    of_desc_str_t key;
    of_serial_num_t ipv4_addr;
    of_serial_num_t ipv4_wg;
};
```

Listing 5.8: DPKM_ADD_PEER OpenFlow Experimenter message for adding a new peer defined in the Loxigen class dpkm_add_peer.c.

To delete a peer the `of_dpkm_delete_peer` message (listing 5.9) was developed with a key field for storing the public key of the peer to delete. This message took inspiration from the `add_peer` message, using the `desc_str` description type.

```
struct of_dpkm_delete_peer : of_dpkm_header {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == 3;
    pad(8);
    of_desc_str_t key;
};
```

Listing 5.9: DPKM_DELETE_PEER OpenFlow Experimenter message for deleting a peer defined in the Loxigen class dpkm_delete_peer.c.

Responding to the previously defined messages requires a status message that can be returned by the switch. The `of_dpkm_status` message (listing 5.10) was implemented for this purpose and is the largest DPKM message with five data fields. The status message contains an enumerator object `status_flag` which can be one of the five options shown in listing 5.11. This indicates the state of the node and the successful completion of a key management operation. The status message also stores the public key, switch ipv4, and WireGuard ipv4 of the node, and the ipv4 address of the peer. This is to further synchronise that system state and assists in the functionality of the procedures (5.4).

```
struct of_dpkm_status : of_dpkm_header {
    uint8_t version;
    uint8_t type == 4;
    uint16_t length;
    uint32_t xid;
    uint32_t experimenter == 0xa20a0323;
    uint32_t subtype == 5;
    enum ofp_dpkm_status_flag status_flag;
    of_desc_str_t key;
    of_serial_num_t ipv4_addr;
    of_serial_num_t ipv4_wg;
    of_serial_num_t ipv4_peer;
};
```

Listing 5.10: DPKM_STATUS OpenFlow Experimenter message for returning the state of the node defined in the Loxigen class dpkm_status.c.

```
enum ofp_dpkm_status_flag (wire_type=uint32_t) {
    OFP_DPKM_STATUS_FLAG_CONFIGURED = 0,
    OFP_DPKM_STATUS_FLAG_PEER_ADDED = 1,
    OFP_DPKM_STATUS_FLAG_PEER_REMOVED = 2,
    OFP_DPKM_STATUS_FLAG_CONNECTED = 3,
    OFP_DPKM_STATUS_FLAG_REVOKED = 4,
};
```

Listing 5.11: Enumerator Status Flags for representing the state of the node defined in the Loxigen class dpkm_status_flag.c.

Match Extension

Handling packets in OpenFlow-based switches involves the construction of flow table entries that specify actions/instructions to apply to packets satisfying the match conditions. OpenFlow Extensible Match (OXM) fields are used to determine the packets accepted/rejected by the switch, implementing specific packet matching constraints such as an ipv4 destination address or message type. The `of_oxm` experimenter structure can be used to define new match fields

and implement additional packet handling logic in the switch. The research of Snow et al. (2018) extended OpenFlow match conditions and documented the process of introducing experimenter labels for increasing the granularity of switch forwarding logic [SDG18]. This served as inspiration for the implementation of a new OXM field for matching DPKM packets (listing 5.12). This extension was introduced to enable matching on the DPKM experimenter id (0xa20a0323) and subsequently route traffic through the WireGuard tunnel.

```
struct of_oxm_dpkm_method : of_oxm {
    uint32_t type_len == 0xffff0205;
    uint32_t experimenter == 0xa20a0323;
    uint8_t value;
};
```

Listing 5.12: OXM extension for a new packet matching condition in the Loxigen class oxm-dpkm.c.

Error Extension

Reporting on errors during the operation of DPKM is essential for diagnosing and addressing a fault in the node. Fifteen error messages, displayed in table 4.2 of the previous chapter (4.3), were implemented as OpenFlow error extensions. Using a wide range of error messages for each key management operation enabled easier location of a bug or issue in the node software. Each error message is an extension of the `of_dpkm_base_error` header, with a unique subtype and a data array that is populated with diagnostic information by the switch. When an error is encountered the switch constructs and sends the appropriate error response to the controller instead of a status response. For brevity, one of the fifteen errors is displayed in code listing 5.13 since all errors have a matching structure. The code detecting the error and handling the response will be investigated in section 5.3. A full list of the errors and other program code can be found in appendix C of this thesis.

```
struct of_dpkm_decode_set_key_error : of_dpkm_base_error {
    uint8_t version;
    uint8_t type == 1;
    uint16_t length;
    uint32_t xid;
    uint16_t err_type == 0xffff;
    uint16_t subtype == 0;
    uint32_t experimenter == 0xa20a0323;
    of_octets_t data;
};
```

Listing 5.13: One of fifteen OpenFlow error extensions in the Loxigen class dpkm_error.c.

5.2.3 Integrating OpenFlow Extensions

Integration of DPKM OpenFlow extensions into the Floodlight codebase was a simple process thanks to Loxigen. Loxigen has been designed to generate an OpenFlowJ-Loxi library from any classes in the `openflow_input` folder of the codebase. Adding all extension classes to this folder and running the console command `make package-java` instructs the tool to generate java interfaces for every class with builders for the new OpenFlow objects. Loxigen checks the structures for any errors or incompatibilities such as using a non-OpenFlow type, halting the generation process on error, and producing .jar files on success. This ensures the new OpenFlowJ-Loxi library is OpenFlow compliant and usable in Floodlight. Code listing 5.14 shows an example of the java interface generated from the `of_dpkm_add_peer` message structure. This can be used by Floodlight as an object builder to construct an add peer message and is integral to the *Add Peer* procedure (section 5.4.3).

```
1  public interface OFDpkmAddPeer extends OFObject, OFDpkmHeader {
2      OFVersion getVersion();
3      OFType getType();
4      long getXid();
5      long getExperimenter();
6      long getSubtype();
7      String getKey();
8      String getIpv4Addr();
9      String getIpv4Wg();
10
11     void writeTo(ByteBuf channelBuffer);
12
13     Builder createBuilder();
14     public interface Builder extends OFDpkmHeader.Builder {
15         OFDpkmAddPeer build();
16         OFVersion getVersion();
17         OFType getType();
18         long getXid();
19         Builder setXid(long xid);
20         long getExperimenter();
21         long getSubtype();
22         String getKey();
23         Builder setKey(String key);
24         String getIpv4Addr();
25         Builder setIpv4Addr(String ipv4Addr);
26         String getIpv4Wg();
27         Builder setIpv4Wg(String ipv4Wg);
28     }
29 }
```

Listing 5.14: Builder for the of_dpkm_add_peer message generated by Loxigen.

Despite its benefits a bug was found in the Loxigen tool during the development. In its current state Loxigen only produces one jar file with the .class java files and outputs the respective .java files to a standard folder. To use the extensions Floodlight needs two jars containing the complete OpenFlow .class and .java files. A workaround was found by manually copying all .java files to a folder and generating the second jar file with the command `jar cvf [folder name] *`. The two jars were then imported into the Floodlight workspace as referenced libraries.

Adding support for OpenFlow extensions in the OVS codebase posed more of a challenge since the DPKM objects must be individually implemented whilst adhering to the same structure. The class `dpkm-ext.h` (appendix C) was created with matching definitions for the previously defined structures. Following this, support for the extensions had to be added across several existing classes to ensure messages with the DPKM experimenter id were accepted and could be mapped to OpenFlow identifiers. Finding the correct classes to modify was an arduous, time consuming process due to the lack of documentation in extending OVS software. Successful integration of OpenFlow extensions required extensive study of raw, mostly uncommented OVS code and debugging of compiler errors. Unlike the controller, OVS required definition of functions to encode/decode messages since no builders could be automatically generated. These functions will be presented in section 5.4 during the documentation of the DPKM procedures. Extensions to the OVS codebase will be summarised in section 5.3.2.

5.3 Overview of Software Extensions

Implementing the full DPKM system involved the addition and modification of a multitude of classes and functions across four codebases – Floodlight, Floodlight-webui, Loxigen, and Open vSwitch (OVS). Approximately 7500 lines of code were needed, with the bulk of the functionality and processing logic implemented in the Floodlight controller (4700 lines). Extensions to OVS and the UI required a similar number of lines – 1100 and 1200 respectively, with the Loxigen OpenFlow extensions (5.2) using the least at under 500. It is not feasible to document all classes and functions within the main body of this thesis due to the scale and complexity of the implementation.

This section aims to provide a condensed overview of the software extensions, covering Floodlight (5.3.1), OVS (5.3.2), and the UI (5.3.3). This section explores the main message

listener, error handler, and REST extensions provided by the controller (5.3.1), outlines several additions to OVS (5.3.2), and gives an insight into the structure of the UI (5.3.3). Deeper investigation into the functionality will be undertaken during the documentation of the procedures (5.4), using this section and the appendix (C) to supplement the readers understanding of DPKM.

5.3.1 Floodlight

Extending Floodlight involved the definition of a new package in the codebase named `net.floodlightcontroller.dpkmconfigurewg`, containing the seventeen classes described in section 4.1.1 of the design chapter. Table 5.3 presents a full summary of the classes with a short description of each and a link to the definition in the appendix.

Class	**Description**	**Methods**
ConnectionProvider	Provides connection to cntrldb MySQL db.	2 (C.1.1)
DpkmConfigureWG	Main handler for DPKM functionality, integrating the module, sending messages, and capturing responses.	25 (C.1.4)
Dpkm	Internal methods for querying db and sending certain messages.	22 (C.1.2)
DpkmFlows	Internal methods for managing switch flow tables.	4 (C.1.10)
DpkmSwitch	Object representing a single node.	1 (C.1.16)
DpkmPeers	Object representing a peer connection.	1 (C.1.12)
DpkmErrorHandler	Main handler for capturing and processing DPKM error messages.	17 (C.1.9)
DpkmError	Internal methods for logging errors and querying db.	8 (C.1.8)
IDpkmConfigureWGService	Defines abstract functions implemented by DpkmConfigureWG and links REST APIs to underlying module.	14(C.1.17)
DpkmConfigureWGWebRoutable	Registers paths to REST resources for externally calling APIs.	2 (C.1.6)
DpkmConfigureWGResource	GET nodes, POST *Configure WG* (5.4.1), DELETE *Unconfigure WG* (5.4.2), deserialize json.	4 (C.1.5)
DpkmManagePeerResource	GET peers, POST *Add Peer* (5.4.3), DELETE *Delete Peer* (5.4.4), deserialize json.	4 (C.1.11)
DpkmStartCommunicationResource	POST *Start Communication* (5.4.5).	1 (C.1.15)
DpkmEndCommunicationResource	POST *End Communication* (5.4.6).	1 (C.1.7)
DpkmRekeyResource	POST *Rekey* (5.4.7).	1 (C.1.13)
DpkmCompromiseNodeResource	POST *Compromise* (5.4.8).	1 (C.1.3)
DpkmRevocationResource	POST *Revocation* (5.4.8).	1 (C.1.14)

Table 5.3: All classes implemented in the net.floodlightcontroller.dpkmconfigurewg Floodlight package.

Message Handling

DpkmConfigureWG was the first class to be built, registering a new Floodlight listener module by implementing the two interfaces `IFloodlightModule` and `IOFMessageListener`. Several dependencies were necessary for integration such as `IFloodlightProviderService` for interacting with connected switches, `IOFSwitchService` for instantiating switches, and `IRestApiService` for adding the REST APIs. The message listener was initialized for capturing OpenFlow experimenter messages by implementing the `receive` method from `IOFMessageListener` (listing 5.15). When the controller sends or receives messages over OpenFlow all running listeners inspect the type field and either process the message or pass it on to the next module. If the message is type experimenter (4), it is cast to the DPKM header defined in 5.2.1 to check the subtype value. The receive method calls `processStatusMessage` (listing 5.16) if the subtype is 5 (`of_dpkm_status`), casting the message to an `OFDpkmStatus` object.

The message is processed further by checking the status flag to determine which controller-to-switch message triggered the response, calling `processConfigured` if `of_dpkm_set_key`, `processRevoked` if `of_dpkm_delete_key`, `processPeerAdded` if `of_dpkm_add_peer`, and `processPeerRemoved` if `of_dpkm_delete_peer`. These methods use Dpkm to access/modify the database, sending subsequent messages based on the procedure and system state. Message handling in the context of DPKM procedures will be covered in depth in section 5.4.

```
1  @Override
2    public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx){
3      switch(msg.getType()) {
4          case EXPERIMENTER:
5            // Cast the message to a DPKM OpenFlow experimenter message.
6            OFDpkmHeader inExperimenter = (OFDpkmHeader) msg;
7
8            // Subtype 5 means a status message has been received and should be processed.
9            if(inExperimenter.getSubtype() == 5) {
10             processStatusMessage(sw, (OFDpkmStatus)msg);
11             break;
12           }
13
14       default:
15         break;
16     }
17     return Command.CONTINUE;
18   }
```

Listing 5.15: Condensed version of the receive method implementing the DPKM message listener capturing OpenFlow experimenter messages, taken from the DpkmConfigureWG.java class.

```
private void processStatusMessage(IOFSwitch sw, OFDpkmStatus msg) {
    // Executed if the status response shows WG has been configured.
    if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_CONFIGURED) {
      DpkmSwitch node = statusToNode(sw, msg);
      processConfigured(sw, node);
    }
    // Executed if the status response shows a peer has been added.
    if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_PEER_ADDED) {
      DpkmPeers peer = statusToPeer(sw, msg);
      processPeerAdded(sw, peer);
    }
    // Executed if the status response shows a peer has been deleted.
    if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_PEER_REMOVED) {
      DpkmPeers peer = statusToPeer(sw, msg);
      processPeerRemoved(sw, peer);
    }
    // Executed if the status response shows a key has been revoked (unconfigured).
    if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_REVOKED) {
      DpkmSwitch node = statusToNode(sw, msg);
      processRevoked(sw, node);
    }
  }
```

Listing 5.16: processStatusMessage method determining the controller-to-switch message that triggered the status response and passing the information as an object for further processing, taken from the DpkmConfigureWG.java class.

Error Handling

When a problem is encountered during the operation of a DPKM procedure the controller receives an error response from OVS. Much like status responses, functionality for processing and logging the errors is essential for diagnosing and fixing issues. As described in section 5.2.2, fifteen error message extensions were created and must be handled by the controller. DpkmErrorHandler is a new Floodlight listener that captures both experimenter and error messages in the receive method (listing 5.17). Error messages are cast to the DPKM base error header (5.2.1) and processed with `processErrorMessage` (listing 5.18). The subtype is compared to fifteen cases, setting a unique error code and note for each, and calling `handleErrorSetKey`, `handleErrorDeleteKey`, `handleErrorAddPeer`, or `handleErrorOther` based on the request triggering the error response. Each handler logs the error in the ErrorLog database table, sets resolved to false, and attempts to resend the message a further two times to determine if the error was an anomaly. If the max number of attempts is reached, the error is marked as "ACTION NEEDED" and the node is blocked until resolved is set to true. This prevents initiation of procedures with a faulty node and ensures the underlying issue is addressed.

```
@Override
  public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
    switch(msg.getType()) {
      case EXPERIMENTER:
        // Cast the message to a DPKM OpenFlow experimenter message.
          OFDpkmHeader inExperimenter = (OFDpkmHeader) msg;
          // Subtype 5 means a status message has been received and should be processed.
          if(inExperimenter.getSubtype() == 5) {
            if(checkError(sw.getId().toString()) > 0) {
              processResponseMessage(sw.getId().toString(), (OFDpkmStatus)msg);
            }
            break;
          }
          else {
            // Ignore other DPKM messages.
            break;
          }
      case ERROR:
        // Cast the message to a DPKM OpenFlow error message.
        OFDpkmBaseError inError = (OFDpkmBaseError) msg;
        if(inError.getErrType() == OFErrorType.EXPERIMENTER) {
          processErrorMessage(sw, inError);
        } else {
          log.error("An error unrelated to DPKM occurred. See log for details.");
        }
        break;
    }
  }
```

Listing 5.17: Receive method for the DPKM error message listener capturing error messages for logging issues and experimenter messages for resolving errors, taken from the DpkmErrorHandler.java class.

```
private synchronized void processErrorMessage(IOFSwitch sw, OFDpkmBaseError inError) {
    String errCode, note, ipv4Peer;
    switch(inError.getSubtype()) {
      case 0:
        errCode = "DECODE_SET_KEY";
        note = "failed to decode a DPKM_SET_KEY message";
        handleErrorSetKey(sw, errCode, note);
        break;
      case 7:
        errCode = "EXECUTE_ADD_PEER";
        note = "failed to execute WG add peer script";
        ipv4Peer = extractIp(inError);
        handleErrorAddPeer(sw, errCode, note, ipv4Peer);
        break;
      case 8:
        errCode = "EXECUTE_DELETE_PEER";
        note = "failed to execute WG delete peer script";
        ipv4Peer = extractIp(inError);
        handleErrorDeletePeer(sw, errCode, note, ipv4Peer);
        break;
      case 9:
        errCode = "GET_KEY";
        note = "failed to retrieve the public key";
```

```
24          handleErrorOther(sw, inError, errCode, note);
25          break;
26        case 12:
27          errCode = "MISSING_KEY";
28          note = "sent a message with missing public key";
29          handleErrorOther(sw, inError, errCode, note);
30          break;
31      }
32    }
```

Listing 5.18: Condensed version of the processErrorMessage method showing 5 out of 15 cases for checking error type and calling the respective handler, taken from the DpkmErrorHandler.java class.

Status responses are always captured in DpkmErrorHandler before DpkmConfigureWG to resolve errors. DpkmErrorHandler uses the DpkmError class to query the database, calling `processResponseMessage` (listing 5.19) if the node has any unresolved errors. The status flag is inspected, resolving any associated errors since the request is now returning as expected. For example, if a set key message returned an execution error on attempt one, the error is logged, a second set key message is sent, and the error is marked as resolved if a status response is received. The developer aims to enhance the managerial qualities of the system in future developments (6.6) by implementing functionality for viewing and resolving errors from the application UI.

```
1  private void processResponseMessage(String dpid, OFDpkmStatus msg) {
2      // Executed if the status response shows WG has been configured.
3      if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_CONFIGURED) {
4        resolveError(dpid, "SET_KEY");
5      }
6      // Executed if the status response shows a peer has been added.
7      if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_PEER_ADDED) {
8        resolveError(dpid, "ADD_PEER");
9      }
10     // Executed if the status response shows a peer has been deleted.
11     if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_PEER_REMOVED) {
12       resolveError(dpid, "DELETE_PEER");
13     }
14     // Executed if the status response shows a key has been revoked (unconfigured).
15     if(msg.getStatusFlag() == OFDpkmStatusFlag.DPKM_STATUS_FLAG_REVOKED) {
16       resolveError(dpid, "DELETE_KEY");
17     }
18   }
```

Listing 5.19: processResponseMessage method marking unresolved errors as resolved if the node is now returning a status message as expected, taken from the DpkmErrorHandler.java class.

REST

Integrating REST functionality in the DPKM Floodlight module consisted of creating and adding the `IDpkmConfigureWGService` interface to DpkmConfigureWG with the methods required by the procedures, defining seven `ServerResource` classes with the necessary `GET`, `POST`, and `DELETE` functions, and establishing a `RestletRoutable` class for registering the endpoint paths. Upon loading the module the REST APIs are added by defining a new instance of `DpkmConfigureWGWebRoutable` (listing 5.20) to support external requests to controller resources. The UI uses the URLs to submit requests to methods in the corresponding resource classes. IDpkmConfigureWGService is instantiated in each resource to access methods in DpkmConfigureWG and perform the key management functionality.

```
1  public class DpkmConfigureWGWebRoutable implements RestletRoutable {
2    @Override
3      public Restlet getRestlet(Context context) {
4          Router router = new Router(context);
5          router.attach("/retrieve/json", DpkmConfigureWGResource.class);
6          router.attach("/configure/json", DpkmConfigureWGResource.class);
7          router.attach("/delete/key/json", DpkmConfigureWGResource.class);
8          router.attach("/retrieve/peers/json", DpkmManagePeerResource.class);
9          router.attach("/add/peer/json", DpkmManagePeerResource.class);
10         router.attach("/delete/peer/json", DpkmManagePeerResource.class);
11         router.attach("/start/json", DpkmStartCommunicationResource.class);
12         router.attach("/end/json", DpkmEndCommunicationResource.class);
13         router.attach("/rekey/json", DpkmRekeyResource.class);
14         router.attach("/compromise/json", DpkmCompromiseNodeResource.class);
15         router.attach("/revoke/json", DpkmRevocationResource.class);
16         return router;
17     }
18
19     @Override
20     public String basePath() {
21         return "/wm/dpkm";
22     }
23 }
```

Listing 5.20: DpkmConfigureWGWebRoutable class registering the REST resource classes to URLs so external calls can be made from the UI.

5.3.2 Open vSwitch

Adding new functionality to Open vSwitch (OVS) relied predominantly on extending existing code rather than developing a new set of classes. Only one new class `dpkm-ext.h` was created for defining the new OpenFlow experimenter messages, modifying approximately 45 existing classes. Implementing DPKM in OVS was more challenging than Floodlight due to the inconsistent commenting and lack of guidance resulting in a time-consuming trial and error development approach. A list of noteworthy classes and extensions will be presented, with the full list of code additions included in the appendix (C.2).

dpkm-ext.h: Defines the enums and structs for DPKM messages and flags using a matching format to the Loxigen files in the controller (5.2.2). Any inconsistencies such as differences in message size cause an error to be returned or misalignment of message fields. All messages must be within 64-bit boundaries and were padded with empty bits.

extract-ofp-msgs: Used to extract OpenFlow message types from headers and categorise messages by vendor. To extract DPKM messages the experimenter ID (0xa20a0323) was defined and the following code (listing 5.21) was added to the `extract_ofp_msgs` function.

```
DPKM_VENDOR_ID = 0xa20a0323

elif type_ == 'DPKM':
                hdrs = (version, OFPT_VENDOR, 0, DPKM_VENDOR_ID, number)
```

Listing 5.21: Snippet from extract-ofp-msgs for extracting OpenFlow message headers and mapping to DPKM.

ofp-msgs.h: Contains abstractions of OpenFlow message headers and was extended to add the DPKM message identifiers. Two enums `ofpraw` (raw identifiers) and `ofptype` (semantic identifiers) were expanded as in listing 5.22. The comments are used by OVS macros to identify captured messages, using the version number and subtype for classification, mapping to structs in `dpkm-ext.h`. The `get_status` message was never implemented and will be discussed in the limitations section (6.5).

```
enum ofpraw {
    /* DPKM 1.0+ (0): struct ofp_dpkm_set_key. */
    OFPRAW_DPKM_SET_KEY,

    /* DPKM 1.0+ (1): struct ofp_dpkm_delete_key. */
    OFPRAW_DPKM_DELETE_KEY,

    /* DPKM 1.0+ (2): struct ofp_dpkm_add_peer. */
    OFPRAW_DPKM_ADD_PEER,

    /* DPKM 1.0+ (3): struct ofp_dpkm_delete_peer. */
    OFPRAW_DPKM_DELETE_PEER,

    /* DPKM 1.0+ (5): struct ofp_dpkm_status. */
    OFPRAW_DPKM_STATUS,
};
enum ofptype{
    OFPTYPE_DPKM_SET_KEY,          /* OFPRAW_DPKM_SET_KEY. */
    OFPTYPE_DPKM_DELETE_KEY,       /* OFPRAW_DPKM_DELETE_KEY. */
    OFPTYPE_DPKM_ADD_PEER,         /* OFPRAW_DPKM_ADD_PEER. */
    OFPTYPE_DPKM_DELETE_PEER,      /* OFPRAW_DPKM_DELETE_PEER. */
    OFPTYPE_DPKM_STATUS,           /* OFPRAW_DPKM_STATUS. */
};
```

Listing 5.22: Snippet from ofp-msg.h adding the raw (ofpraw) and semantic (ofptype) identifiers for DPKM messages.

ofp-util.c: Expanded to include four DPKM message decoder functions: `ofputil_decode_dpkm_set_key` (269-284), `ofputil_decode_dpkm_delete_key` (292-307), `ofputil_decode_dpkm_add_peer` (315-342), `ofputil_decode_dpkm_delete_peer` (350-377). These are used during the procedures by several handler functions to translate message fields into structs and map from network byte order to host byte order. Class accompanied by the header file `ofp-util.h`.

ofproto.c: Main class for handling all OpenFlow messages, including deserializing, performing events, and sending responses to the controller. Majority of the DPKM functionality was added here. The function `handle_single_part_openflow` (8915-9121) was extended to call four handlers – `handle_dpkm_set_key` (6465-6510), `handle_dpkm_delete_key` (6518-6556), `handle_dpkm_add_peer` (6564-6610), `handle_dpkm_delete_peer` (6618-6664) – based on the `OFPTYPE` (listing 5.23). The four handlers call the respective decode functions in `ofp-util.c`, execute commands against the WG interface, and send status responses.

Six functions were added to interact with WireGuard – `configure_wg` (6281-6301), `unconfigure_wg` (6310-6331), `add_peer_wg` (6341-6366), `delete_peer_wg` (6373-6397),

get_pubkey (6403-6420), get_wg_addr (6427-6457) – and will be discussed in the context of the procedures (5.4). Class accompanied by the header file ofproto.h.

```
1  static enum ofperr
2  handle_single_part_openflow(struct ofconn *ofconn, const struct ofp_header *oh,
3                              enum ofptype type)
4  {
5      switch (type) {
6      case OFPTYPE_DPKM_SET_KEY:
7          return handle_dpkm_set_key(ofconn, oh);
8      case OFPTYPE_DPKM_DELETE_KEY:
9          return handle_dpkm_delete_key(ofconn, oh);
10     case OFPTYPE_DPKM_ADD_PEER:
11         return handle_dpkm_add_peer(ofconn, oh);
12     case OFPTYPE_DPKM_DELETE_PEER:
13         return handle_dpkm_delete_peer(ofconn, oh);
14     case OFPTYPE_DPKM_STATUS:
15         return 0;
16     }
17 }
```

Listing 5.23: Snippet from ofproto.c that checks the OpenFlow message type and calls the respective handler function.

ofp-errors.h: Contains abstractions of OpenFlow error message headers and was extended to add fifteen DPKM OFPERR error identifiers. Enum ofperr was modified to include the error messages with matching subtype values to those in Loxigen for compatibility with the controller.

Match: Adding a new flow matching field involved modification of 16 files including: extract-ofp-fields for defining the DPKM vendor, meta-flow.c/meta-flow.h for integrating the dpkm_method OXM, and flow.c/flow.h for defining how the field is extracted and used for packet matching in the flow table.

5.3.3 Application UI

Supporting DPKM functionality on the UI involved the creation of a new web page named dpkm.html. Integration of the DPKM page was relatively easy, following the header style from existing files and linking it via the navbar. The DPKM UI page is comprised of the following elements:

- Two tables displaying records from the ConfiguredPeers and CommunicatingPeers database tables using the `GetWGSwitches` and `GetWGPeers` functions to make `GET` requests for returning data from the controller.
- Five modal forms for configuring nodes (5.4.1), adding peers (5.4.3), ending communication (5.4.6), simulating compromise (5.4.8), and revoking keys (5.4.8).
- Twenty-five JavaScript/JQuery functions for making the page dynamic, initiating procedures through REST calls, and returning errors/information.

Figure 5.2 displays the full application UI and labels the elements by the underlying functions and procedure initiated. Focused analysis will be provided in 5.4 by traversing the relevant UI components for each procedure. When compared to the mock UI (4.1.3), the styling approach is more basic but delivers all the expected functionality. This was due to the challenging time constraints of the project timeline and the complexity of control/data plane development leaving little time for aesthetics. The developer feels that the basic interface does not devalue the delivered product since the focus was on implementing data plane key management. Future developments on the UI will commence after project submission and are discussed in the succeeding evaluation chapter (6). Full documentation of the UI code is not feasible within the main body of this thesis and has therefore been linked in the appendix (C.3).

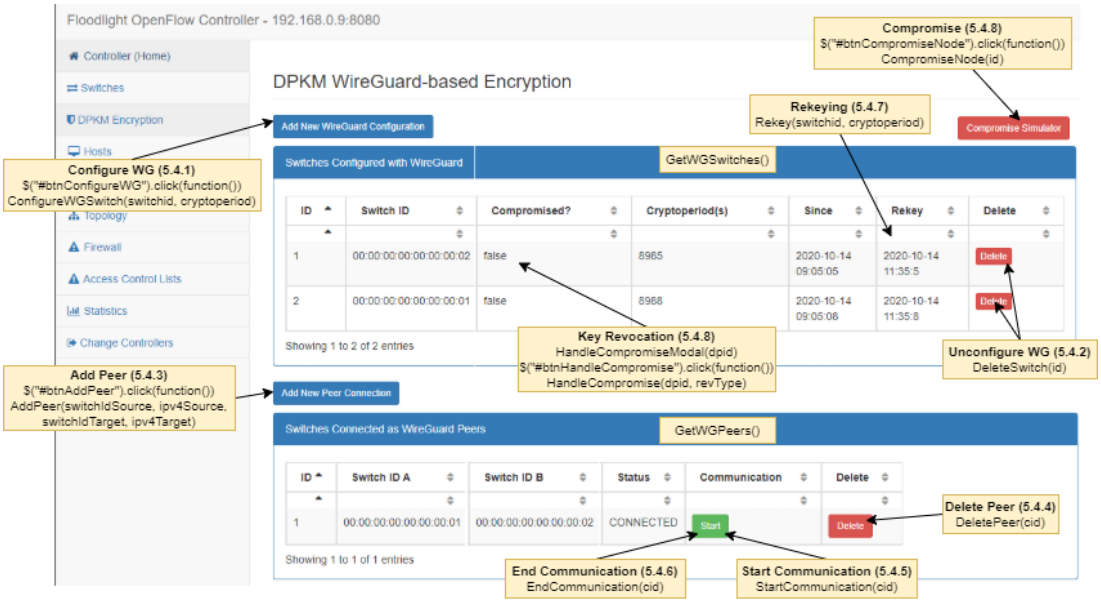

Figure 5.2: DPKM User Interface labelled with the functions called and the procedures initiated. Multiple table fields have been removed for visibility.

5.4 Developing Procedures

Developing the underlying key management functionality of DPKM involved the implementation of the procedures designed in section 4.2 across the application, Floodlight, and OVS codebases. As described in the previous section, Floodlight was extended with a new DPKM module that provides REST APIs for the procedures, constructs the new OpenFlow experimenter objects proposed in section 5.2, sends DPKM messages over OpenFlow, and handles status and error responses. A new web page was added to the Floodlight Web UI to initialise the procedures over REST and accurately reflect the state of the system. The OVS codebase was modified to process DPKM messages, perform key management operations to adjust the WireGuard setup, and send responses to the controller. The packet analyser Wireshark [Wir20] was used for analysis of communication between the controller and switch with a Lua script generated by Loxigen specifically for the DPKM OpenFlow extensions. The system was built with three VMs (5.1) and operates as a minimal SDN with one controller and two nodes.

This section documents the bulk of the implementation and divides the content by procedure, traversing each sequence of functionality and respective code from initiation by the application UI to mediation by Floodlight and finally execution by OVS. Project source code and screenshots from Wireshark will be presented throughout for replication and as evidence of a functional system. The scale and novelty of the implementation was challenging for a single developer, but evidence of the successful delivery of most proposed functionality is provided, with any missing features discussed in the subsequent evaluation chapter (6). For brevity, only relevant code snippets have been extracted, with the full source code linked in appendix C.

5.4.1 Configuring WireGuard

The functionality of the *Configure WireGuard* procedure (4.2.1) was previously designed in a nine-step sequence diagram (figure 4.16) which was used as the basis for implementation. Configuring WireGuard is always the first procedure to be executed because all subsequent procedures rely on a configured or multiple configured WireGuard interfaces. When the controller is running the application UI, previously shown in figure 5.2, is deployed and can initiate the procedure. The administrator clicks the "Add New WireGuard Configuration" button to display the modal form in figure 5.3. The cryptoperiod is entered and the switch is selected from a drop-down menu populated by the `LoadSwitches` function calling the standard `GET` switches REST API.

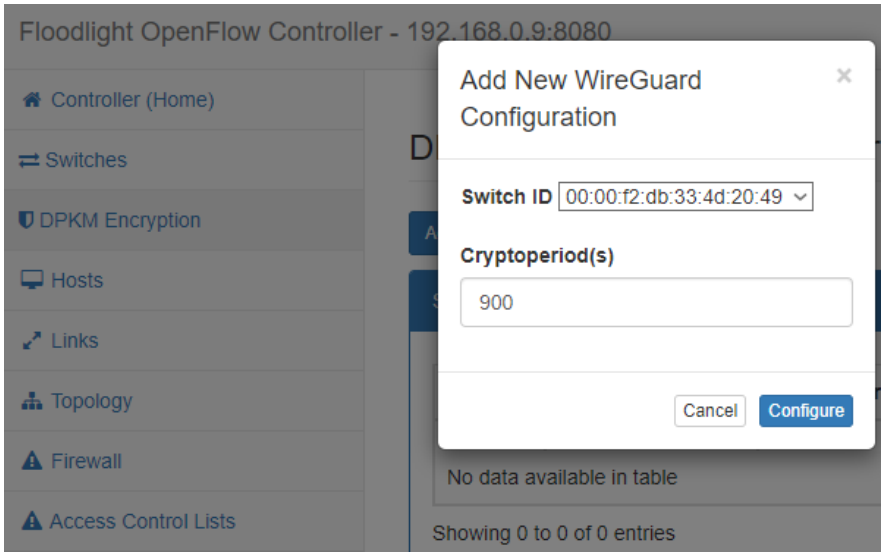

Figure 5.3: Screenshot captured from the application UI, displaying the modal popup form for initiating the Configure WireGuard procedure.

Clicking the "Configure" button makes a `POST` call to the endpoint `/wm/dpkm/configure/json` which is registered to the DpkmConfigureWGResource class, passing the switch ID and cryptoperiod as a json structure to the `configure` method (listing 5.24). The json is translated to a DpkmSwitch object using the method `jsonToDpkmSwitch`, which converts json fields into java attributes. An instance of the IDpkmConfigureWGService interface is initialised to access the main controller module and execute the `sendSetKeyMessage` method (listing 5.25).

```
@Post
public String configure(String fmJson) {
    IDpkmConfigureWGService configureWG =
        (IDpkmConfigureWGService) getContext().getAttributes()
        .get(IDpkmConfigureWGService.class.getCanonicalName());
    DpkmSwitch node = jsonToDpkmSwitch(fmJson);
    String status = null;
    if (node == null) {
        status = "Error! Could not parse switch info, see log for details.";
        log.error(status);
        return ("{\"status\" : \"" + status + "\"}");
    }
    if (configureWG.checkError(node.getDpId()) > 0) {
        status = "Switch has an unresolved error that requires administrator " +
            "attention. See log for details.";
        log.error(status);
        return ("{\"status\" : \"" + status + "\"}");
    }
    configureWG.sendSetKeyMessage(DatapathId.of(node.getDpId()), node.getCryptoperiod());
```

```
    status = "DPKM_SET_KEY message sent to switch.";
    return ("{\"status\" : \"" + status + "\"}");
}
```

Listing 5.24: POST method for configuring a WireGuard interface, taken from the DpkmConfigureWGResource.java class.

```
@Override
public void sendSetKeyMessage(DatapathId dpid, int cryptoperiod) {
    IOFSwitch sw = switchService.getSwitch(dpid);
    currentCryptoperiod = cryptoperiod;
    try {
        OFDpkmSetKey setKeyMsg = sw.getOFFactory().buildDpkmSetKey()
            .build();
        sw.write(setKeyMsg);
    } catch (Exception e) {
        e.printStackTrace();
        log.error("Unable to send DPKM_SET_KEY message.");
    }
}
```

Listing 5.25: sendSetKeyMessage method constructing and sending an of_dpkm_set_key message from controller to switch, taken from the DpkmConfigureWG.java class.

`sendSetKeyMessage` gets an instance of the switch matching the DatapathId (dpid), temporarily stores the cryptoperiod, constructs an `of_dpkm_set_key` message using the corresponding Loxigen builder, and sends the message over OpenFlow to the OVS switch. Figure 5.4 shows the OpenFlow packet captured in Wireshark, sent from the controller address (192.168.0.9) to the node A address (192.168.0.8).

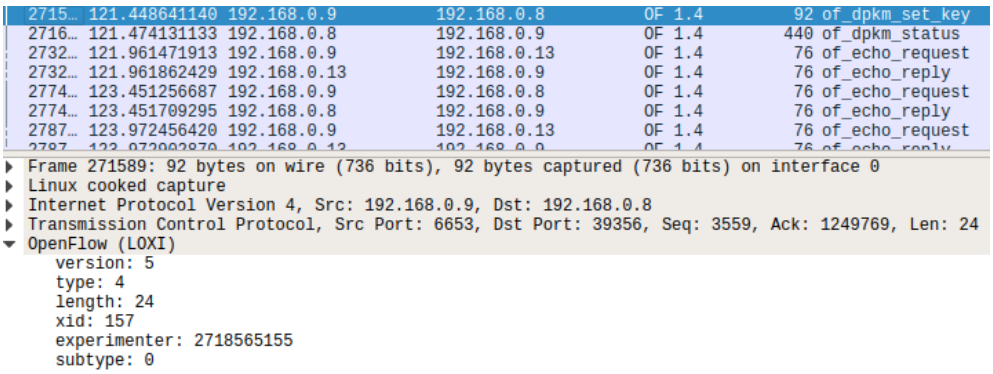

Figure 5.4: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_set_key message.

The switch receives the message, maps the subtype (0) to the OpenFlow type (`OFPTYPE_DPKM_SET_KEY`), and calls the corresponding handler – `handle_dpkm_set_key`. The handler calls the `ofputil_decode_dpkm_set_key` message decoder and executes the `configure_wg` function (listing 5.26) to run the configuration script (listing 5.27), returning a decode or execute error if unsuccessful.

```
enum ofperr
configure_wg(void)
{
    char *cmd = "./ConfigureWG.sh";
    char buf[BUFSIZE];
    FILE *fp;
    if ((fp = popen(cmd, "r")) == NULL)
    {
        return OFPERR_DPKM_EXECUTE_SET_KEY;
    }
    while (fgets(buf, BUFSIZE, fp) != NULL )
    {
        printf("OUTPUT: %s", buf);
    }
    if(pclose(fp))
    {
        return OFPERR_DPKM_EXECUTE_SET_KEY;
    }
    return 0;
}
```

Listing 5.26: OVS function for executing the underlying WireGuard configuration script, taken from the ofproto.c class.

```
wg genkey | tee privatekey | wg pubkey > publickey
chmod 0 privatekey
chmod 444 publickey
wg set wg0 listen-port 51820 private-key ./privatekey
```

Listing 5.27: Configuration script generating an asymmetric key pair and binding the private key to the WireGuard interface.

The configuration script generates a public and private key pair, modifies the file access permissions, binds the private key to the WireGuard interface, and sets the receiving port to UDP port 51820. This updates the WireGuard interface from a blank interface to the interface in figure 5.5.

```
luke@luke-nodea:~$ sudo wg
interface: wg0
  listening port: 33264
luke@luke-nodea:~$ sudo wg
interface: wg0
  public key: ynNORyKDQNt5CDpQZbiwVyXsaAsyyBnsRy0e4fnczmY=
  private key: (hidden)
  listening port: 51820
```

Figure 5.5: Screenshot of the node console showing the transition from a blank WireGuard interface to a configured interface with a public key.

If decoding of the message and execution of the script is successful, the handler will create a status response (listing 5.28) with the new public key and switch/WireGuard ipv4 addresses, sending it to the controller. Figure 5.6 shows the status response in Wireshark, with the `CONFIGURED` status flag informing the controller of the state of the node.

```
static enum ofperr
handle_dpkm_set_key(struct ofconn *ofconn, const struct ofp_header *oh)
{
    /* ..................................... */
    /* Create and initialise status object. */
    buf = ofpraw_alloc_reply(OFPRAW_DPKM_STATUS, oh, 0);
    cstatus = ofpbuf_put_zeros(buf, sizeof *cstatus);
    /* Set flag as 0: CONFIGURED. */
    cstatus->status_flag = 0;
    /* Copy values into message fields.*/
    ovs_strlcpy(cstatus->key, public_key, sizeof cstatus->key);
    ovs_strlcpy(cstatus->ipv4_addr, ipv4_addr, sizeof cstatus->ipv4_addr);
    ovs_strlcpy(cstatus->ipv4_wg, wg_addr, sizeof cstatus->ipv4_wg);
    /* Send response to controller. */
    ofconn_send_reply(ofconn, buf);
}
```

Listing 5.28: Extract of the OVS handle_dpkm_set_key function for creating and sending a status response to the controller, taken from the ofproto.c class.

```
2716… 121.474131133 192.168.0.8      192.168.0.9      OF 1.4   440 of_dpkm_status
2732… 121.961471913 192.168.0.9      192.168.0.13     OF 1.4    76 of_echo_request
2732… 121.961862429 192.168.0.13     192.168.0.9      OF 1.4    76 of_echo_reply
2774… 123.451256687 192.168.0.9      192.168.0.8      OF 1.4    76 of_echo_request
2774… 123.451709295 192.168.0.8      192.168.0.9      OF 1.4    76 of_echo_reply
2787… 123.972456420 192.168.0.9      192.168.0.13     OF 1.4    76 of_echo_request
2787… 123.972992870 192.168.0.13     192.168.0.9      OF 1.4    76 of_echo_reply

▸ Frame 271663: 440 bytes on wire (3520 bits), 440 bytes captured (3520 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.0.8, Dst: 192.168.0.9
▸ Transmission Control Protocol, Src Port: 39356, Dst Port: 6653, Seq: 1249769, Ack: 3583, Len: 372
▾ OpenFlow (LOXI)
    version: 5
    type: 4
    length: 372
    xid: 157
    experimenter: 2718565155
    subtype: 5
    status_flag: OFP_DPKM_STATUS_FLAG_CONFIGURED (0)
    key: ynNORyKDQNt5CDpQZbiwVyXsaAsyyBnsRy0e4fnczmY=\n
    ipv4_addr: 192.168.0.8
    ipv4_wg: 10.0.0.1
    ipv4_peer:
```

Figure 5.6: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating successful configuration of the node.

The controller captures the response with the DPKM message listener discussed in section 5.3.1, checks the status flag, and calls the `processConfigured` method (listing 5.29). This method uses the Dpkm class to query the cntrldb database for any existing records associated with the node and performs one of the following operations:

- **No record exists:** new record is added to the ConfiguredPeers database table with the `writeSwitchToDB` method, adding the key and IP information contained in the status message. Any other configured nodes are automatically added as peers by initiating the add peer procedure (5.4.3).
- **Cannot access DB:** database cannot be queried for existing records and an error is logged.
- **Record exists (no peers):** existing ConfiguredPeers record for node is updated with current information using the `updateSwitchInDB` method.
- **Record exists (peers):** existing record is updated, and peer information is retrieved from the CommunicatingPeers table. If the key of one peer has changed, the connection status is updated to `PID1ONLY`, and the peer is deleted and re-added to replace the old public key. If the key of both peers has changed, the connection status is updated to `BOTH REMOVED`, and both peers are deleted and re-added to replace the public keys. These cases will be explained in section 5.4.7 since the configuration procedure is also used during rekeying.

```java
private synchronized void processConfigured(IOFSwitch sw, OFDpkmStatus msg) {
    // Adds the new WG switch to the database.
    if (checkIPExists(msg.getIpv4Addr()).equals("0")) {
        writeSwitchToDB(msg, sw);
        // Checks for and adds any configured switches as peers.
    } else if (checkIPExists(msg.getIpv4Addr()).equals("Error")) {
        log.error("An error was encountered when adding switch info to DB.");
    } else {
        // Updates existing switch details for new configuration.
        updateSwitchInDB(msg, sw);
        // If switch has peers then deletes peer on one or both interfaces based on state.
    }
}
```

Listing 5.29: Condensed version of the processConfigured method for processing the status response to a set key message and updating the database accordingly. Taken from DpkmConfigureWG.java.

The following database fields are added/updated for a node in the ConfiguredPeers table by the controller during the configuration procedure:

- **Dpid:** set to switches `DatapathId`.
- **Cryptoperiod:** added from the stored form input.
- **PubKey1/IPv4Addr/IPv4WG:** populated from the status response
- **PubKey2:** blank if no previous record, replaced by old `PubKey1` if existing record.
- **Status:** set to `CONFIGURED`.
- **Compromised:** set to `false`.
- **Since:** updated to timestamp of current system date/time.

Finally, the controller logs the success or failure of the configuration procedure (figure 5.7) and updates the database table on the UI with the new record. The node is now in the *"WireGuard Configured"* state (figure 4.15) and ready for the *Add Peer* procedure (5.4.3).

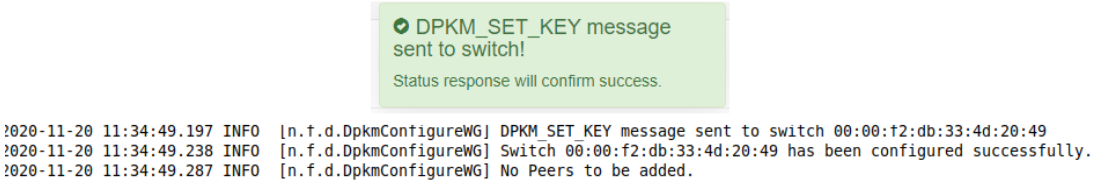

Figure 5.7: Compilation of two screenshots: one showing the status response to a set key message on the application UI, and one showing the output in the controller console.

5.4.2 Unconfiguring WireGuard

The *Unconfigure WireGuard* procedure (4.2.2) is responsible for restoring a node to a blank state by removing all key and peer information from the WireGuard interface. The procedure operates in a similar nine step sequence of functionality to configuring WireGuard and was implemented based on the sequence diagram in figure 4.17. The procedure can be initiated by clicking the "Delete" button next to a configured node in the "Switches Configured with WireGuard" table of the application UI (figure 5.8). To ensure a node is not accidentally removed the popup in figure 5.9 is displayed, asking for confirmation from the administrator.

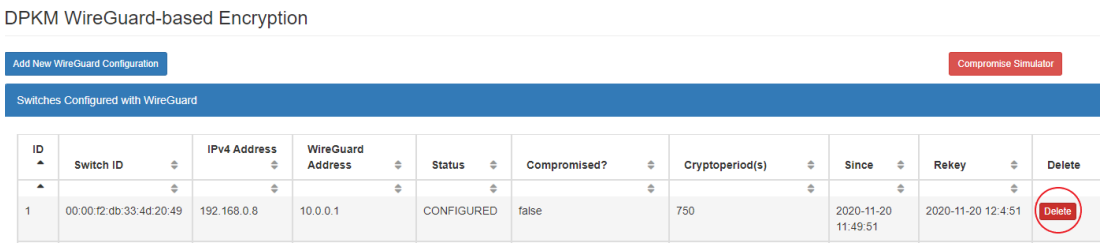

Figure 5.8: Screenshot of the configured switches table displayed in the application UI with a delete button (red circle) for unconfiguring a node.

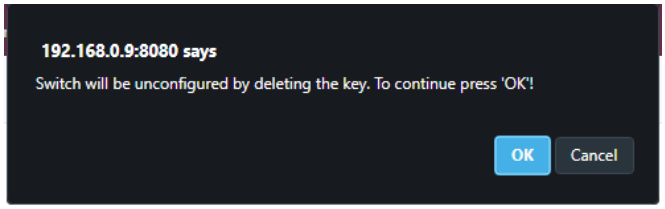

Figure 5.9: Popup notification prompting the administrator to confirm initiation of the Unconfigure WireGuard procedure.

Clicking the "OK" button passes the database row id to the `DeleteSwitch` function which makes a `DELETE` call to the REST endpoint `/wm/dpkm/delete/key/json` registered to the `delete` method (listing 5.30) of the DpkmConfigureWGResource class. An instance of IDpkmConfigureWGService is initialised to access the `getSwitches` method of the DpkmConfigureWG class, retrieving the switch dpid, and executing the `sendDeleteKeyMessage` method (listing 5.31).

```java
@Delete
  public String delete(String fmJson) {
    IDpkmConfigureWGService configureWG =
        (IDpkmConfigureWGService)getContext().getAttributes()
        .get(IDpkmConfigureWGService.class.getCanonicalName());
    DpkmSwitch node = jsonToDpkmSwitch(fmJson);
    String status = null;
    if (node == null) {
      status = "Error! Could not parse switch info, see log for details.";
      log.error(status);
      return ("{\"status\" : \"" + status + "\"}");
    }
    boolean exists = false;
    Iterator<DpkmSwitch> iter = configureWG.getSwitches().iterator();
    // Loop through switch records to find switch information.
    while (iter.hasNext()) {
      DpkmSwitch s = iter.next();
      if (s.id == node.id) {
        node.dpid = s.dpid;
        exists = true;
        break;
      }
    }
    if (!exists) {
      status = "Error! No switch with this id exists.";
      log.error(status);
      return ("{\"status\" : \"" + status + "\"}");
    } else {
      configureWG.sendDeleteKeyMessage(DatapathId.of(node.dpid));
      status = "DPKM_DELETE_KEY message sent to switch.";
      return ("{\"status\" : \"" + status + "\"}");
    }
  }
```

Listing 5.30: DELETE method for unconfiguring a WireGuard interface, taken from the DpkmConfigureWGResource.java class.

```java
@Override
  public void sendDeleteKeyMessage(DatapathId dpid) {
    IOFSwitch sw = switchService.getSwitch(dpid);
    try {
      OFDpkmDeleteKey delKeyMsg = sw.getOFFactory().buildDpkmDeleteKey()
          .build();
      sw.write(delKeyMsg);
    } catch(Exception e) {
      e.printStackTrace();
      log.error("Unable to send DPKM_DELETE_KEY message.");
    }
  }
```

Listing 5.31: sendDeleteKeyMessage method constructing and sending an of_dpkm_delete_key message from controller to switch, taken from the DpkmConfigureWG.java class.

The `sendDeleteKeyMessage` method instantiates a switch object matching the dpid and uses the `OFDpkmDeleteKey` Loxigen builder to construct and send an `of_dpkm_delete_key` message over OpenFlow. Figure 5.10 displays a screenshot from Wireshark with the captured packet containing the delete key experimenter message sent from the controller (192.168.0.9) to node A (192.168.0.8).

2464…	911.149739894	192.168.0.9	192.168.0.8	OF 1.4	92	of_dpkm_delete_key
2464…	911.150493474	192.168.0.8	192.168.0.9	OF 1.4	440	of_dpkm_status
2465…	911.405313660	00:00:00_00:00:00	LLDP_Multicast	OF 1.4	183	of_packet_out
2465…	911.405335151	00:00:00_00:00:00	LLDP_Multicast	OF 1.4	183	of_packet_out
2465…	911.433709945	00:00:00_00:00:00	Broadcast	OF 1.4	191	of_packet_out
2465…	911.433730735	00:00:00_00:00:00	Broadcast	OF 1.4	191	of_packet_out
2467…	912.184402744	192.168.0.9	192.168.0.13	OF 1.4	92	of_dpkm_set_key
2467…	912.190509086	192.168.0.13	192.168.0.9	OF 1.4	440	of_dpkm_status
2471…	913.435013860	192.168.0.9	192.168.0.8	OF 1.4	76	of_echo_request
2474…	914.186543268	192.168.0.9	192.168.0.13	OF 1.4	76	of_echo_request
2474…	914.186935775	192.168.0.13	192.168.0.9	OF 1.4	76	of_echo_reply
2477…	915.439309328	192.168.0.9	192.168.0.8	OF 1.4	76	of_echo_request
2479…	916.151116567	192.168.0.8	192.168.0.9	OF 1.5	76	of_hello

```
Frame 2464278: 92 bytes on wire (736 bits), 92 bytes captured (736 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 192.168.0.9, Dst: 192.168.0.8
Transmission Control Protocol, Src Port: 6653, Dst Port: 39356, Seq: 19513, Ack: 5649, Len: 24
OpenFlow (LOXI)
    version: 5
    type: 4
    length: 24
    xid: 1492
    experimenter: 2718565155
    subtype: 1
```

Figure 5.10: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_delete_key message.

Upon delivery OVS uses the `handle_single_part_openflow` function to map the subtype (1) to the OpenFlow type (`OFPTYPE_DPKM_DELETE_KEY`) and calls the corresponding handler – `handle_dpkm_delete_key`. The handler decodes the message with the `ofputil_decode_dpkm_delete_key` function and executes `unconfigure_wg` (listing 5.32) to run the `unconfigure` script (listing 5.33).

```
enum ofperr
unconfigure_wg(void)
{
    char *cmd = "./UnconfigureWG.sh";
    char buf[BUFSIZE];
    FILE *fp;

    if ((fp = popen(cmd, "r")) == NULL)
    {
        return OFPERR_DPKM_EXECUTE_DELETE_KEY;
    }
    while (fgets(buf, BUFSIZE, fp) != NULL )
    {
        printf("OUTPUT: %s", buf);
    }
```

```
16      if(pclose(fp))
17      {
18          return OFPERR_DPKM_EXECUTE_DELETE_KEY;
19      }
20      return 0;
21  }
```

Listing 5.32: OVS function for executing the underlying unconfigure WireGuard script, taken from the ofproto.c class.

```
1  /usr/local/share/openvswitch/scripts/ovs-ctl stop
2  ip link set s1 down
3  ovs-vsctl del-port s1 wg0
4  ip link set wg0 down
5  ip link delete wg0
6  rm -f privatekey
7  rm -f publickey
8  ip link add dev wg0 type wireguard
9  ip addr add 10.0.0.1/24 dev wg0
10 ovs-vsctl add-port s1 wg0
11 ip link set wg0 up
12 ip link set s1 up
13 /usr/local/share/openvswitch/scripts/ovs-ctl start
```

Listing 5.33: Unconfigure WireGuard script shutting down OVS, removing the interface and keys, creating a new unconfigured interface, and restarting OVS.

Gracefully unconfiguring the WireGuard interface of a live node was an implementation challenge that presented several issues. Firstly, WireGuard does not provide the functionality to unconfigure an interface, the only solution is to fully delete the interface. Secondly, if the interface is fully deleted the node cannot be reconfigured by the configuration procedure and is unusable in DPKM until manually repaired. Thirdly, the switch must be temporarily shut down to modify the interface otherwise the node will freeze/crash. Lastly, if the node is shutdown then a response to the controller cannot be sent.

The implemented solution is far from optimal and has its own limitations but manages to address these issues. The unconfigure script temporarily shuts down OVS, removing the current WireGuard interface and keys, before creating a new unconfigured interface and restarting OVS. This makes the node usable in the future but disconnects and reconnects the connection causing a nondisruptive error to be logged in the controller. The only feasible way of returning a status response was sending it before running the unconfigure script whilst the node is still connected. This is a design flaw because the status response does not accurately reflect the state of the node and if an error is encountered during execution a second (error) response is

sent. This limitation is mitigated slightly by allowing the controller to use the error response to accurately update the network state. Nevertheless, the *Unconfigure WireGuard* procedure still has room for improvement and will be evaluated further in section 6.5 of this thesis.

The status response is constructed by OVS and sent to the controller, as in the configuration procedure (listing 5.28). Figure 5.11 displays a screenshot of the message in Wireshark, sent from node A (192.168.0.8) to the controller (192.168.0.9) with the `REVOKED` status flag.

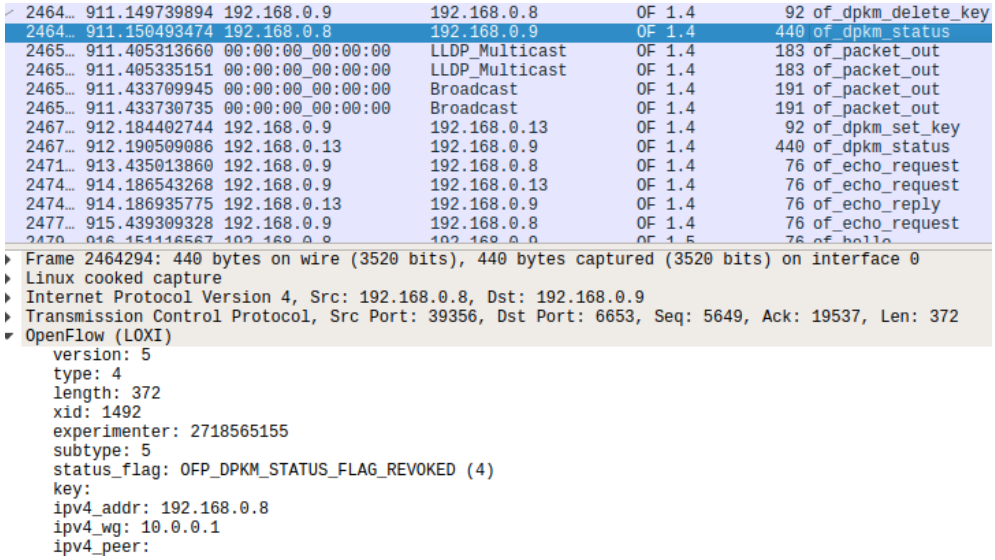

Figure 5.11: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating that the node has been unconfigured.

After capturing the response the controller checks the flag and calls the `processRevoked` method (listing 5.34). The `checkConnectedAny` method of the Dpkm class is used to discover if any connections to the node exist and delete the unconfigured node as a peer on all connected interfaces. The peer connection status is set to `REMOVED` in the CommunicatingPeers table, so the node and respective connections are deleted once the status responses indicate that no connections remain. The `removeSwitch` method is called to wipe all records of the node from the database. Finally, the controller reports on the success or failure of the unconfigure procedure (figure 5.12) and reflects the transition from the *"WireGuard Configured"* to the *"Node Waiting"* state (figure 4.15).

```
private synchronized void processRevoked(IOFSwitch sw, OFDpkmStatus msg) {
    log.info(String.format("Switch %s has successfully deleted it's key.", sw.getId().toString
        ()));
    // If connections to unconfigured switch exist set connection status 'REMOVED'.
    // Send delete peer messages to all target peers.
    // Otherwise, remove switch from db.
    if(checkConnectedAny(msg.getIpv4Addr()) >= 1) {
      // 2 == key removed.
      updatePeerInfo(msg.getIpv4Addr(), msg.getIpv4Peer(), 2);
      Iterator<DpkmPeers> iter = getPeers().iterator();
      while (iter.hasNext()) {
        DpkmPeers p = iter.next();
        if (p.ipv4AddrA.equalsIgnoreCase(msg.getIpv4Addr())) {
          sendDeletePeerMessage(p.dpidB, p.dpidA, false);
        }
        if (p.ipv4AddrB.equalsIgnoreCase(msg.getIpv4Addr())) {
          sendDeletePeerMessage(p.dpidA, p.dpidB, false);
        }
      }
    } else {
      removeSwitch(msg.getIpv4Addr());
    }
  }
```

Listing 5.34: The method for processing the status response to a delete key message, removing an remaining connections, and updating the database accordingly. Taken from DpkmConfigureWG.java.

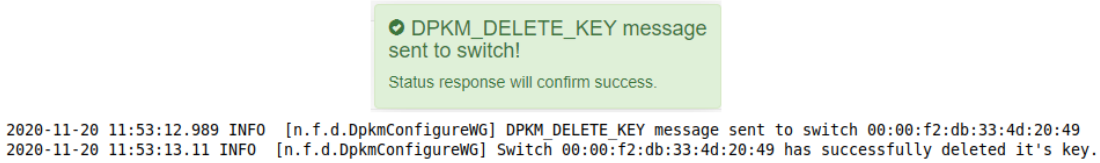

Figure 5.12: Compilation of two screenshots: one showing the status response to a delete key message on the application UI, and one showing the output in the controller console.

5.4.3 Adding a Peer

The *Add Peer* procedure (4.2.3) establishes a tunnel between the WireGuard interfaces of two configured nodes and sets up the conditions for encrypted node-to-node communication. The sequence diagram in figure 4.18 was mostly followed for implementation aside from one design decision. In the implementation the controller blocks a new tunnel from being constructed between two connected nodes rather than running the *Delete Peer* procedure (5.4.4) to delete and re-add the peers. This decision was made to constrain the frequency of tunnel rebuilding since active tunnels should not be disturbed, forcing the administrator to explicitly remove and re-add peer connections.

The process of adding a peer is either initiated internally by the controller or externally from the application UI by clicking the "Add New Peer Connection" button to toggle the modal form in figure 5.13. Peers are selected by choosing from the list of switch dpid identities in the dropdown menus. The dropdowns are populated using the `GetWGSwitchesModal` function to make a `GET` request to the `/wm/dpkm/retrieve/json` endpoint, retrieving a list of configured nodes from the ConfiguredPeers database table. JavaScript is used to enhance the modals intelligence by removing a selected peer in dropdown A as an option from dropdown B and dynamically updating the read-only IP address fields based on the selections.

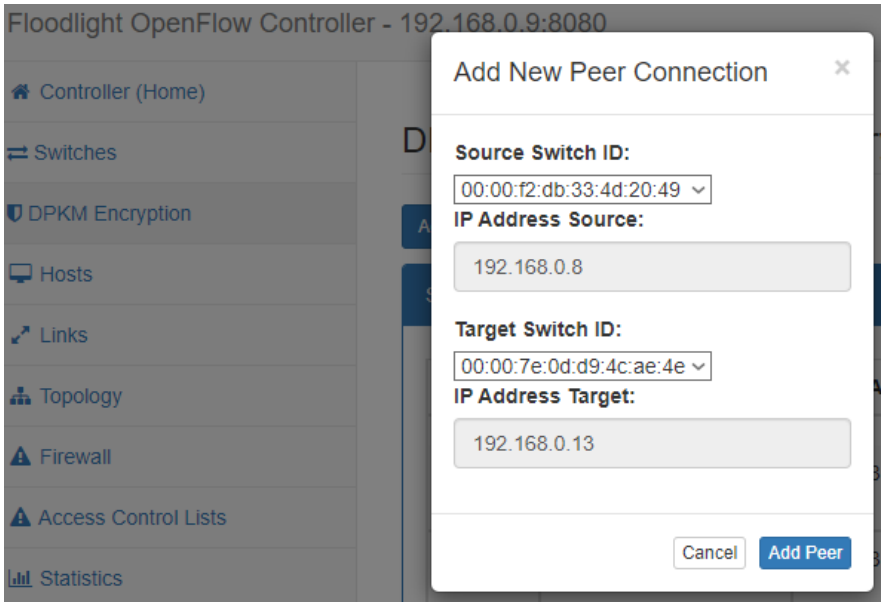

Figure 5.13: Screenshot captured from the application UI, displaying the modal popup form for creating a new peer connection by initiating the Add Peer procedure.

Clicking the "Add Peer" button makes a `POST` request containing the two dpids and two IP addresses of the peers to the endpoint `/wm/dpkm/add/peer/json` registered to the `add` method (listing 5.35) of the DpkmManagePeerResource class. The input data is passed to the `add` method in json format and translated to a DpkmPeers object using the deserialize method `jsonToDpkmPeer`. The IDpkmConfigureWGService interface is used to access the `checkConnected` and `sendAddPeerMessage` (listing 5.36) methods.

```java
@Post
    public String add(String fmJson) {
      IDpkmConfigureWGService configureWG =
        (IDpkmConfigureWGService)getContext().getAttributes()
        .get(IDpkmConfigureWGService.class.getCanonicalName());
    DpkmPeers peers = jsonToDpkmPeer(fmJson);
    String status = null;
    if (peers == null) {
      status = "Error! Could not parse switch info, see log for details.";
      return ("{\"status\" : \"" + status + "\"}");
    }
    if (configureWG.checkConnected(ipv4AddrA, ipv4AddrB, "CONNECTED") > 0) {
      status = "Error! A peer connection with this switch already exists.";
      log.error(status);
      return ("{\"status\" : \"" + status + "\"}");
    }
    else if (configureWG.checkConnected(ipv4AddrA, ipv4AddrB, "CONNECTED") == -1) {
      status = "Error! Failed to access the database.";
      log.error(status);
      return ("{\"status\" : \"" + status + "\"}");
    } else {
      configureWG.sendAddPeerMessage(peers.dpidA, peers.dpidB);
      status = "DPKM_ADD_PEER message sent to switch.";
      return ("{\"status\" : \"" + status + "\"}");
    }
    }
```

Listing 5.35: POST method for adding a new peer, taken from DpkmManagePeerResource.java.

```java
  public void sendAddPeerMessage(String sourceDpid, String targetDpid) {
    IOFSwitch sw = switchService.getSwitch(DatapathId.of(sourceDpid));
    String getSQL = String.format("SELECT PubKey1, IPv4Addr, IPv4AddrWG "
        + "FROM cntrldb.ConfiguredPeers WHERE Dpid = '%s';", targetDpid);
    // Connects to the database and executes the SQL statement.
    try(Connection connect = ConnectionProvider.getConn();
        PreparedStatement peerInfo = connect.prepareStatement(getSQL);) {
      boolean isResult = peerInfo.execute();
        do {
          try (ResultSet rs = peerInfo.getResultSet()) {
            while (rs.next()) {
              // Calls internal function to build and write message.
              sendAddPeerMessageInternal(sw, rs.getString(1),rs.getString(2),rs.getString(3));
            }
            isResult = peerInfo.getMoreResults();
          }
        } while (isResult);
        connect.close();
    } catch (SQLException e) {
      log.error("Failed to access the database when sending DPKM_ADD_PEER message.");
    }
  }
```

Listing 5.36: sendAddPeerMessage method retrieving peer information from the database and passing it to the sendAddPeerMessageInternal method, taken from the DpkmConfigureWG.java class.

The `checkConnected` method queries the database for any peer connections between the two nodes, displaying an error on the UI if a connection exists (figure 5.14). The `sendAddPeerMessage` method retrieves the target peer information from the database and executes `sendAddPeerMessageInternal` (listing 5.37) in the Dpkm class to construct and send an add peer message over OpenFlow to peer switch A with peer B's public key, switch ipv4, and WireGuard ipv4. A database record for the connection is made pre-emptively to ensure synchronisation of the system in circumstances where multiple add peer messages have been sent and are awaiting response.

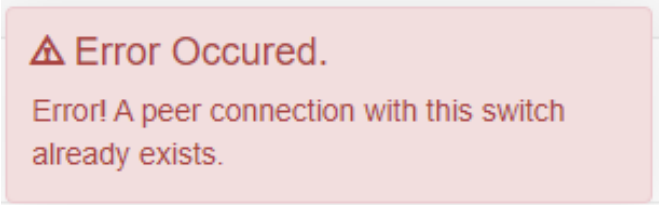

Figure 5.14: Popup error notification informing the administrator that the connection already exists.

```
protected void sendAddPeerMessageInternal(IOFSwitch sw, String peerPubKey,
      String peerIPv4, String peerIPv4WG) {
    try {
      String sourceIPv4 = getIp(sw.getId().toString(), false);
        // If no connection exists add new record with status 'PID1ONLY'.
        // Otherwise, update connection to status 'BOTH'.
        if(checkConnected(sourceIPv4, peerIPv4, "CONNECTED") == 0) {
          addPeerConnection(sourceIPv4, peerIPv4);
        }
        else if(checkConnected(sourceIPv4, peerIPv4, "BOTH REMOVED") > 0) {
          updatePeerInfo(sourceIPv4, peerIPv4, "PID1ONLY");
        }
        else {
          updatePeerInfo(sourceIPv4, peerIPv4, "BOTH");
        }
        OFDpkmAddPeer addPeerMsg = sw.getOFFactory().buildDpkmAddPeer()
            .setKey(peerPubKey)
            .setIpv4Addr(peerIPv4)
            .setIpv4Wg(peerIPv4WG)
            .build();
        sw.write(addPeerMsg);
    }
    catch(Exception e) {
      DpkmConfigureWG.log.error("Failed to send ADD_PEER message.");
    }
  }
```

Listing 5.37: sendAddPeerMessageInternal method constructing and sending an of_dpkm_add_peer message from controller to switch, taken from the Dpkm.java class.

Figure 5.15 presents the captured `of_dpkm_add_peer` experimenter message in Wireshark sent from the controller (192.168.0.9) to node B (192.168.0.13), followed by another message to node A (192.168.0.8) after the first status response.

23641	8.983079350	192.168.0.9	192.168.0.13	OF 1.4	412	of_dpkm_add_peer
23656	8.986307158	192.168.0.13	192.168.0.9	OF 1.4	440	of_dpkm_status
24432	9.085195405	192.168.0.9	192.168.0.8	OF 1.4	412	of_dpkm_add_peer
24450	9.087708837	192.168.0.8	192.168.0.9	OF 1.4	440	of_dpkm_status
28128	10.134567564	00:00:00_00:00:00	LLDP_Multicast	OF 1.4	183	of_packet_out
28129	10.134595210	00:00:00_00:00:00	LLDP_Multicast	OF 1.4	183	of_packet_out
28408	10.232573839	00:00:00_00:00:00	Broadcast	OF 1.4	191	of_packet_out
28409	10.232597726	00:00:00_00:00:00	Broadcast	OF 1.4	191	of_packet_out
34092	12.233490708	192.168.0.9	192.168.0.13	OF 1.4	76	of_echo_request
34093	12.233633838	192.168.0.9	192.168.0.8	OF 1.4	76	of_echo_request

```
Frame 23641: 412 bytes on wire (3296 bits), 412 bytes captured (3296 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 192.168.0.9, Dst: 192.168.0.13
Transmission Control Protocol, Src Port: 6653, Dst Port: 39990, Seq: 57, Ack: 405, Len: 344
OpenFlow (LOXI)
    version: 5
    type: 4
    length: 344
    xid: 401
    experimenter: 2718565155
    subtype: 2
    key: ynNORyKDQNt5CDpQZbiwVyXsaAsyyBnsRy0e4fnczmY=\n
    ipv4_addr: 192.168.0.8
    ipv4_wg: 10.0.0.1
```

Figure 5.15: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_add_peer message.

As seen in the previous procedures, OVS uses the `handle_single_part_openflow` function to map the message subtype (2) to the OpenFlow type (`OFPTYPE_DPKM_ADD_PEER`) and calls the corresponding handler – `handle_dpkm_add_peer`. The handler decodes the message with `ofputil_decode_dpkm_add_peer` (listing 5.38), copying the message fields into a struct, and returning an error if the key or ip addresses are missing. The `add_peer_wg` function is used to add the peer information to interface wg0 with the command in listing 5.39. This enables messages to be received from the peer ipv4 address and instructs the interface to send messages to the peer's endpoint and UDP listen port 51820.

```
enum ofperr
ofputil_decode_dpkm_add_peer(const struct ofp_header *oh,
                             struct ofputil_dpkm_add_peer *pin)
{
    struct ofpbuf b = ofpbuf_const_initializer(oh, ntohs(oh->length));
    enum ofpraw raw = ofpraw_pull_assert(&b);

    if (raw == OFPRAW_DPKM_ADD_PEER) {
        const struct ofp_dpkm_add_peer *apm = b.msg;
        if (apm->key[0] == '\0'){
            return OFPERR_DPKM_MISSING_KEY;
        }
```

```
13          if (apm->ipv4_addr[0] == '\0'){
14              return OFPERR_DPKM_MISSING_IP_S;
15          }
16          if (apm->ipv4_wg[0] == '\0'){
17              return OFPERR_DPKM_MISSING_IP_WG;
18          }
19          pin->experimenter = ntohl(apm->experimenter);
20          pin->subtype = ntohl(apm->subtype);
21          ovs_strlcpy(pin->key, apm->key, sizeof pin->key);
22          ovs_strlcpy(pin->ipv4_addr, apm->ipv4_addr, sizeof pin->ipv4_addr);
23          ovs_strlcpy(pin->ipv4_wg, apm->ipv4_wg, sizeof pin->ipv4_wg);
24      } else {
25          return OFPERR_DPKM_DECODE_ADD_PEER;
26      }
27      return 0;
28  }
```

Listing 5.38: OVS function decoding an of_dpkm_add_peer message and mapping to an ofputil_dpkm_add_peer struct, taken from the ofp-util.c class.

```
1  wg set wg0 peer [key] allowed-ips [ipv4_wg] allowed-ips [192.168.0.0/24] endpoint [ipv4_addr
       ]:51820 persistent-keep-alive 30
```

Listing 5.39: Command for adding a new peer to the WireGuard interface.

The handler constructs a status response as in the configuration procedure (listing 5.28) but additionally sets the `ipv4_peer` field to the ipv4 address of the add peer message. Figure 5.16 shows the `of_dpkm_status` message captured in Wireshark, sent from node B (192.168.0.13) to the controller (192.168.0.9) with a `PEER_ADDED` flag.

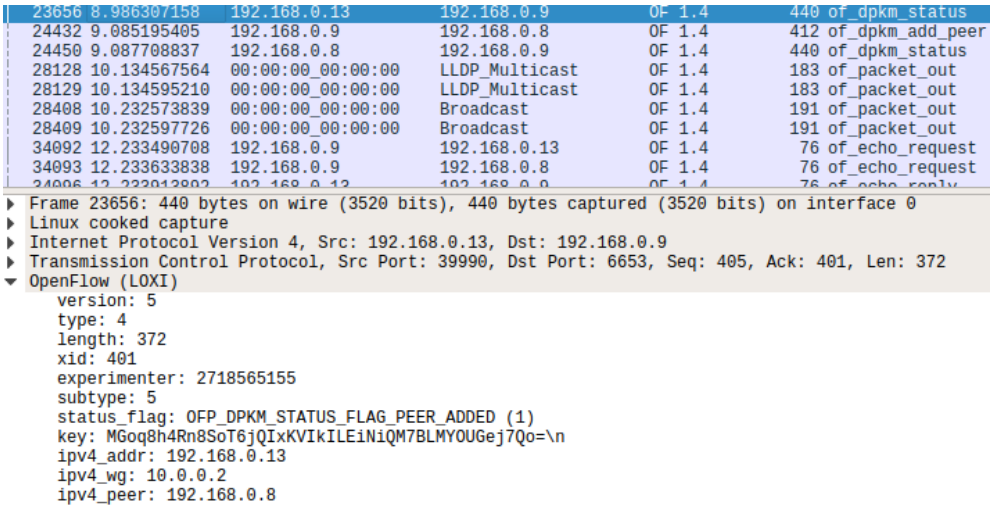

Figure 5.16: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating that a peer has been successfully added.

The controller handles the add peer status response with the `processPeerAdded` method and makes changes to the database based on the connection status. The two methods `updatePeerInfo` and `checkConnected` accept a string value `statusChange/statusType` that is used to build an SQL statement for setting or querying a connection status. Listings 5.40 and 5.41 present two example SQL statements used by `updatePeerInfo` and `checkConnected`.

```
updateQuery = String.format("UPDATE cntrldb.CommunicatingPeers SET Status='%s'"
        + "WHERE (PID1=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s') AND "
        + "PID2=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s')) OR "
        + "(PID1=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s') AND "
        + "PID2=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s'));",
        statusChange, ipv4Addr, ipv4AddrPeer, ipv4AddrPeer, ipv4Addr);
```

Listing 5.40: Example SQL statement used by the updatePeerInfo method to modify the connection status in the database to statusChange. Taken from the Dpkm.java class.

```
checkQ = String.format("SELECT COUNT(*) FROM CommunicatingPeers "
          + "WHERE ((PID1=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s') AND "
          + "PID2=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s')) OR "
          + "(PID1=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s') AND "
          + "PID2=(SELECT id FROM ConfiguredPeers WHERE IPv4Addr='%s'))) AND "
          + "Status = '%s';",
          ipv4AddrA,ipv4AddrB,ipv4AddrB,ipv4AddrA,statusType);
```

Listing 5.41: Example SQL statement used by the checkConnected method to retrieve a count of the connections matching the connection status (statusType). Taken from the Dpkm.java class.

Ensuring synchronicity during procedures involving multiple nodes was an implementation challenge requiring careful use of the database. Eight different connection statuses, displayed in table 5.4, are used to maintain the system state and influence future message sequences. The connection status is checked and updated during all procedures meaning a multitude of different transitions are possible.

Connection Status	**Meaning**
CONNECTED	Default connection state between two peers.
KEY CHANGED	Public key linked to one of the peers has changed.
REMOVED	One of the interfaces has had its key removed.
COMMUNICATING	Active communication over the connection.
PID1ONLY	Only one half of the connection is established.
BOTH	Add Peer message has been sent to both but not confirmed.
BOTH CHANGED	Both public keys have changed.
BOTH REMOVED	Both interfaces have had their keys removed.

Table 5.4: Possible states of a peer connection.

Whilst processing an add peer response the connection status can be changed in the following ways:

- If the connection is `PID1ONLY` only one half of the connection has been established so the add peer procedure is performed again with the target peer.
- If the connection is `BOTH` then no further peers need to be added and the connection is updated to `CONNECTED` or `COMMUNICATING` for restoring flow after rekeying (discussed in 5.4.7).
- If the connection is `CONNECTED` then no further action needs to be taken and the procedure is complete. The controller logs the success or failure of the procedure (figure 5.17) and reflects the state change on the UI.

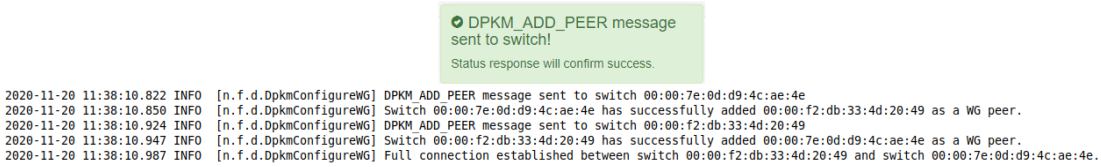

Figure 5.17: Compilation of two screenshots: one showing the status response to an add peer message on the application UI, and one showing the output in the controller console.

Successful execution of the *Add Peer* procedure for both peers updates the WireGuard interfaces from the configured interface (figure 5.5) to the two interfaces in figure 5.18. Setting persistent-keepalive means a single blank packet is sent between the endpoints every 30 seconds to keep the tunnel active and ready for communication. In terms of key management, WireGuard has now performed the handshake discussed in section 2.3.2, creating the sending and receiving session keys from the peer public key and interface's private key. These session keys will be periodically rekeyed by WireGuard and do not require management in DPKM. The nodes have transitioned from the *"WireGuard Configured"* to *"Tunnel Ready"* state (figure 4.15) and can be used by the *Start Communication* procedure (5.4.5) to begin data transport.

```
interface: wg0
  public key: ynNORyKDQNt5CDpQZbiwVyXsaAsyyBnsRy0e4fnczmY=
  private key: (hidden)
  listening port: 51820

peer: MGoq8h4Rn8SoT6jQIxKVIkILEiNiQM7BLMYOUGej7Qo=
  endpoint: 192.168.0.13:51820
  allowed ips: 10.0.0.2/32, 192.168.0.0/24
  latest handshake: 26 seconds ago
  transfer: 464 B received, 272 B sent
  persistent keepalive: every 30 seconds
```

```
interface: wg0
  public key: MGoq8h4Rn8SoT6jQIxKVIkILEiNiQM7BLMYOUGej7Qo=
  private key: (hidden)
  listening port: 51820

peer: ynNORyKDQNt5CDpQZbiwVyXsaAsyyBnsRy0e4fnczmY=
  endpoint: 192.168.0.8:51820
  allowed ips: 10.0.0.1/32, 192.168.0.0/24
  latest handshake: 1 minute, 2 seconds ago
  transfer: 272 B received, 676 B sent
  persistent keepalive: every 30 seconds
```

Figure 5.18: Side-by-side comparison of the node A and node B consoles after the add peer procedure, displaying the WireGuard interfaces connected as peers.

5.4.4 Deleting a Peer

The *Delete Peer* procedure (4.2.4) tears down an existing peer connection by removing peer information from the WireGuard interface of a node. The procedure operates similarly to the *Add Peer* procedure, discussed in the previously section (5.4.3), but reverses the tunnel building process based on the sequence diagram designed in figure 4.19. The structure of the REST classes is akin to the *Unconfigure WireGuard* procedure (5.4.2) since both make `DELETE` requests meaning only unique elements will be presented. The delete peer procedure can be initiated internally by the controller or externally from the application UI by clicking the "Delete" button next to a connection in the "Switches Connected as WireGuard Peers" table (figure 5.19). Much like the *Unconfigure WireGuard* procedure, a popup notification (figure 5.20) is displayed asking for confirmation before making any requests.

Switches Connected as WireGuard Peers

ID	Switch ID A	IPv4 Address A	WireGuard Address A	Switch ID B	IPv4 Address B	WireGuard Address B	Status	Communication	Delete
1	00:00:7e:0d:d9:4c:ae:4e	192.168.0.13	10.0.0.2	00:00:f2:db:33:4d:20:49	192.168.0.8	10.0.0.1	CONNECTED	Start	Delete

Showing 1 to 1 of 1 entries

Figure 5.19: Screenshot of the connected peers table displayed in the application UI with a delete button (red circle) for removing a peer connection.

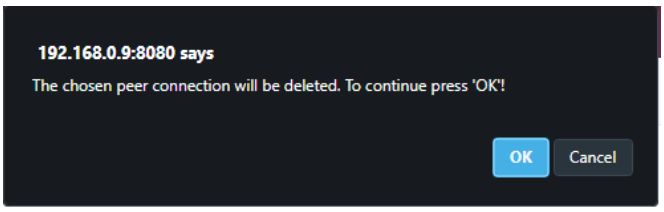

Figure 5.20: Popup notification prompting the administrator to confirm initiation of the Delete Peer procedure.

Clicking "OK" passes the connection id (cid) to the `DeletePeer` function which makes a `DELETE` call to the REST endpoint `/wm/dpkm/delete/peer/json` registered to the `delete` method of the DpkmManagePeerResource class. The method uses `jsonToDpkmPeer` to convert the cid from json format to a field in a DpkmPeers object (as in listing 5.35). The functionality is most like the DpkmConfigureWGResource `delete` method (listing 5.30), instantiating an IDpkmConfigureWGService interface to access `getPeers` and `sendDeletePeerMessage` (listing 5.42) of the main DpkmConfigureWG class. The dpid's of the two switches are retrieved from the database with `getPeers` and passed to `sendDeletePeerMessage`, initiating the procedure.

```
@Override
  public void sendDeletePeerMessage(String sourceDpid, String targetDpid, boolean keyChange) {
    IOFSwitch sw = switchService.getSwitch(DatapathId.of(sourceDpid));
    String getSQL = String.format("SELECT PubKey1 FROM cntrldb.ConfiguredPeers "
        + "WHERE Dpid = '%s';", targetDpid);
    if (keyChange) {
      getSQL = String.format("SELECT PubKey2 FROM cntrldb.ConfiguredPeers "
          + "WHERE Dpid = '%s';", targetDpid);
    }
    // Connects to the database and executes the SQL statement.
    try(Connection connect = ConnectionProvider.getConn();
        PreparedStatement peerInfo = connect.prepareStatement(getSQL);) {
      boolean isResult = peerInfo.execute();
        do {
          try (ResultSet rs = peerInfo.getResultSet()) {
            while (rs.next()) {
              // Calls internal function to build and write message.
              sendDeletePeerMessageInternal(sw, rs.getString(1));
            }
            isResult = peerInfo.getMoreResults();
          }
        } while (isResult);
        connect.close();
    } catch (SQLException e) {
      e.printStackTrace();
      log.error("Failed to access the database when sending DPKM_DELETE_PEER message.");
    }
  }
```

Listing 5.42: sendDeletePeerMessage method initialising a switch, retrieving peer information from the database, and passing it to the sendDeletePeerMessageInternal method. Taken from the DpkmConfigureWG.java class.

The `sendDeletePeerMessage` method initialises the source switch and retrieves the target peer public key. The Boolean `keyChange` selects the old public key (if true) for rekeying scenarios (explained in 5.4.7). The switch and key are passed to `sendDeletePeerMessageInternal`

(listing 5.43) in the Dpkm class to build and send an `of_dpkm_delete_peer` message containing the key over OpenFlow to the source switch. Figure 5.21 displays the captured message in Wireshark sent from the controller (192.168.0.9) to node B (192.168.0.13), followed by an additional delete peer message to node A (192.168.0.8) after the first status response.

```
1  protected void sendDeletePeerMessageInternal(IOFSwitch sw, String peerPubKey) {
2      try {
3          OFDpkmDeletePeer deletePeerMsg = sw.getOFFactory().buildDpkmDeletePeer()
4              .setKey(peerPubKey)
5              .build();
6          sw.write(deletePeerMsg);
7      }
8      catch(Exception e) {
9        DpkmConfigureWG.log.error("Failed to send DELETE_PEER message.");
10     }
11   }
```

Listing 5.43: Method constructing and sending an of_dpkm_delete_peer message from controller to switch, taken from the Dpkm.java class.

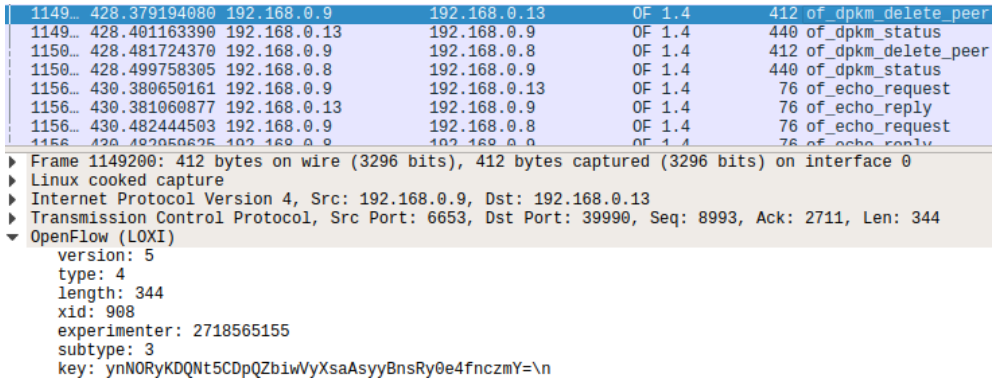

Figure 5.21: Screenshot of Wireshark, displaying a captured of_dpkm_delete_peer message.

Upon receipt OVS uses `handle_single_part_openflow` to map the message subtype (3) to the OpenFlow type (`OFPTYPE_DPKM_DELETE_PEER`) and calls `handle_dpkm_delete_peer`. The function `ofputil_decode_dpkm_delete_peer` is used to convert the message fields into a struct, returning an error if the message cannot be decoded or the key is missing. The `delete_peer_wg` function (listing 5.44) is called to remove any peer matching the public key from the nodes WireGuard interface wg0, returning an error if execution fails. An `of_dpkm_status` response is constructed, as in listing 5.28, and sent from node B (192.168.0.13) to the controller (192.168.0.9) with a `PEER_REMOVED` flag (figure 5.22).

```c
enum ofperr
delete_peer_wg(struct ofputil_dpkm_delete_peer din)
{
    char *cmd = (char*)malloc(300 * sizeof(char));
    din.key[strcspn(din.key, "\n")] = 0;
    sprintf(cmd, "wg set wg0 peer %s remove",
            din.key);
    char buf[BUFSIZE];
    FILE *fp;

    if ((fp = popen(cmd, "r")) == NULL)
    {
        return OFPERR_DPKM_EXECUTE_DELETE_PEER;
    }
    while (fgets(buf, BUFSIZE, fp) != NULL )
    {
        printf("OUTPUT: %s", buf);
    }
    if(pclose(fp))
    {
        return OFPERR_DPKM_EXECUTE_DELETE_PEER;
    }
    free(cmd);
    return 0;
}
```

Listing 5.44: OVS function for executing the delete peer command, removing a peer matching the key from WireGuard interface wg0, taken from the ofproto.c class.

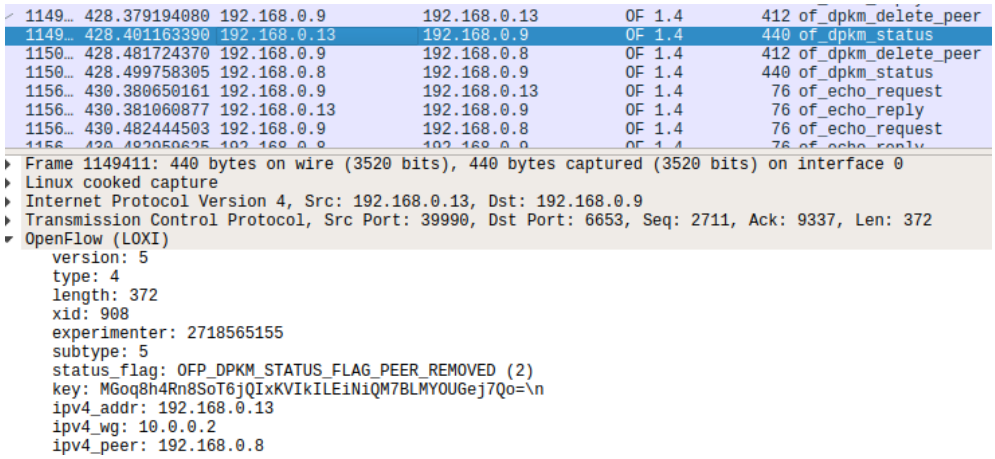

Figure 5.22: Screenshot of the Wireshark packet analyzer, displaying a captured of_dpkm_status message indicating that a peer has been successfully removed.

The delete peer response is captured by the controller and processed with the `processPeerRemoved` method (listing 5.45), using `checkConnected` to retrieve the connection status as a conditional statement for performing additional operations. As explained in the previous section, the connection can be in one of eight different states (table 5.4) and must be handled accordingly. Deleting a peer is a crucial part of several other procedures, meaning the following scenarios are considered in the status response:

- If the connection is `REMOVED` it is no longer valid and can be wiped from the database with `removePeerConnection`. The peer node has been unconfigured so can also be deleted if no other connections exist. This ensures that every tunnel is fully terminated during the *Unconfigure WireGuard* procedure (5.4.2) before deleting records of the node.
- If the connection is `PID1ONLY` peer information with an expired key has been removed from one interface during rekeying. The connection is re-added by sending an `of_dpkm_add_peer` message with the new key.
- If the connection is `BOTH REMOVED` peer information has been removed from both interfaces during rekeying. The connection is rebuilt by sending an `of_dpkm_add_peer` message, which will subsequently lead to another after the first add peer status response.
- If the connection is `CONNECTED` it has not been removed by one of the peers. The *Delete Peer* procedure is performed again with the opposite peer and the connection is removed from the database. The subsequent response will simply trigger the log to display the message "Peer removed from both switches" (figure 5.23).

```
1  private synchronized void processPeerRemoved(IOFSwitch sw, OFDpkmStatus msg) {
2      log.info(String.format("Switch %s has successfully removed %s as a WG peer.",
3          sw.getId().toString(), getDpId(msg.getIpv4Peer(), 0, false)));
4      // Remove connection if exists and status 'REMOVED' (key removed).
5      if(checkConnected(msg.getIpv4Addr(), msg.getIpv4Peer(), "REMOVED") > 0) {
6        removePeerConnection(msg.getIpv4Addr(),msg.getIpv4Peer());
7        // Remove switch if no remaining peer connections.
8        if(checkConnectedAny(msg.getIpv4Peer()) == 0) {
9          removeSwitch(msg.getIpv4Peer());
10       }
11     }
12     // Called if a new connection has been added (PID1ONLY) whilst awaiting second
13     // DELETE_PEER message response.
14     else if(checkConnected(msg.getIpv4Addr(), msg.getIpv4Peer(), "PID1ONLY") > 0) {
15       // Re-add other half of the peer connections with new key.
16       sendAddPeerMessage(sw.getId().toString(), getDpId(msg.getIpv4Peer(), 0, false));
17     }
18     // Called if connection is removed from both interfaces (BOTH REMOVED), but should
19     // be rebuilt.
```

```
20      else if(checkConnected(msg.getIpv4Addr(), msg.getIpv4Peer(), "BOTH REMOVED") > 0) {
21        sendAddPeerMessage(sw.getId().toString(), getDpId(msg.getIpv4Peer(), 0, false));
22      }
23      // Sends delete peer message to target peer, deleting peer info on both WG interfaces. (if
          exists).
24      // Removes connection from db.
25      else if(checkConnected(msg.getIpv4Addr(), msg.getIpv4Peer(), "CONNECTED") > 0) {
26        sendDeletePeerMessage(getDpId(msg.getIpv4Peer(), 0, false), sw.getId().toString(), false
            );
27        removePeerConnection(msg.getIpv4Addr(),msg.getIpv4Peer());
28      }
29      // Connection removed from both switches.
30      else if(checkConnected(msg.getIpv4Addr(), msg.getIpv4Peer(), "CONNECTED") == 0) {
31        log.info("Peer removed from both switches.");
32      }
33    }
```

Listing 5.45: Method for processing the status response to a delete peer message, sending subsequent messages based on the connection status. Taken from DpkmConfigureWG.java.

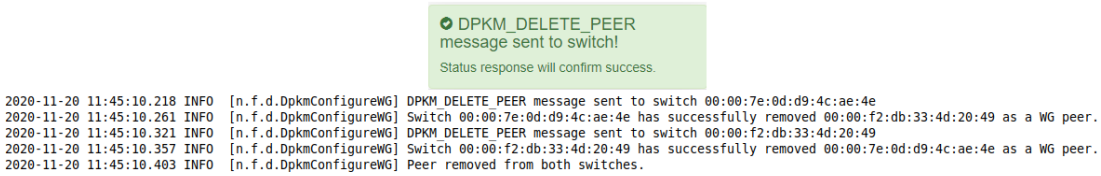

Figure 5.23: Compilation of two screenshots: one showing the status response to a delete peer message on the application UI, and one showing the output in the controller console.

Successful completion of the *Delete Peer* procedure for both peers updates the WireGuard interfaces from the two connected interfaces (figure 5.18) to the two interfaces in figure 5.24. The nodes have transitioned from the *"Tunnel Ready"* to *"WireGuard Configured"* state (figure 4.15) and cannot be used for encrypted communication until a new connection is setup.

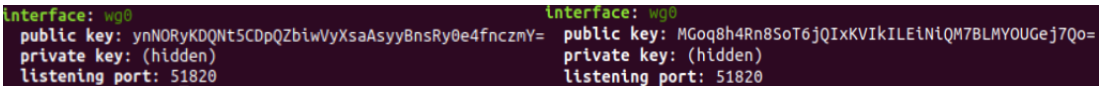

Figure 5.24: Side-by-side comparison of the node A and node B consoles after the delete peer procedure, displaying two unconnected but configured WireGuard interfaces.

5.4.5 Starting Communication

The *Start Communication* procedure (4.2.5) modifies the flow tables of two peer nodes with a new flow entry directing packets through the OVS port containing the WireGuard interface. The purpose of this procedure is to encrypt and route traffic directly to the peer, instead of sending packets unencrypted via the controller. Implementing this procedure was challenging due to the developer's inexperience, technological limitations, and novelty of combining OVS and WireGuard in the node. WireGuard operates strictly at layer three, rejecting any non-IP packets, whereas OVS handles many types of packets. The developer attempted to implement OVS extensions for encapsulation/decapsulation of non-IP packets but could not achieve a working solution within the project timeline. The current working version of DPKM only provides the functionality for routing IP packets. This is an obvious limitation that will be discussed further in the evaluation chapter (6.5) and addressed in future work (6.6).

Communication can only be started with a peer connection in the `CONNECTED` state and is initiated externally from the application UI by clicking the green "Start" button in the "Communication" column of the "Switches Connected as WireGuard Peers" table (figure 5.25). Once communication has started the connection status transitions to `COMMUNICATING` and JavaScript changes the "Communication" column to a red "End" button for running the *End Communication* procedure (5.4.6). The delete option is now disabled because ending communication removes the peer connection.

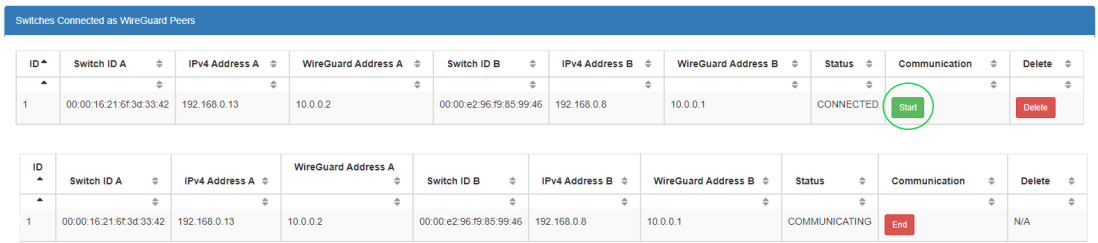

Figure 5.25: Two screenshots of the connected peers table, taken from the application UI, showing the change in the table after clicking start (green circle) to begin communication.

Clicking "Start" passes the cid to the `StartCommunication` function which makes a `POST` call to the REST endpoint `/wm/dpkm/start/json` registered to the `start` method of the DpkmStartCommunicationResource class. Start uses `jsonToDpkmPeer` from the DpkmManagePeerResource class to convert the json into a DpkmPeers object. Start shares many

similarities with the `add` (DpkmManagePeerResource) method (listing 5.35), aside from returning an error if the peers are already communicating and using IDpkmConfigureWGService to call `startCommunication` (listing 5.46) with the two dpids.

```
1  public void startCommunication(String dpidA, String dpidB) {
2        try {
3          // Initialise the switches.
4          IOFSwitch peerA = switchService.getSwitch(DatapathId.of(dpidA));
5        IOFSwitch peerB = switchService.getSwitch(DatapathId.of(dpidB));
6        String ipv4A = getIp(dpidA, false);
7        String ipv4B = getIp(dpidB, false);
8        OFFlowAdd flowA = constructFlowAdd(peerA, peerB);
9        OFFlowAdd flowB = constructFlowAdd(peerB, peerA);
10       // Write FLOW_MOD (ADD) messages to switches.
11       peerA.write(flowA);
12       peerB.write(flowB);
13       // Update DB status, 3 == start communication.
14       updatePeerInfo(ipv4A, ipv4B, "COMMUNICATING");
15       log.info(String.format("Communication between switch %s and switch %s started.",dpidA,
             dpidB));
16     }
17     catch(Exception e) {
18       e.printStackTrace();
19       log.error("Failed to access the database when sending FLOW_MOD message.");
20     }
21     }
```

Listing 5.46: startCommunication method for constructing and sending a flow_add message to each peer, taken from the DpkmConfigureWG.java class.

Using `startCommunication` the controller instantiates the two peer switches and uses the `constructFlowAdd` method of the DpkmFlows class to build two OpenFlow `flow_add` messages. Constructing a flow entry can be broken down into four steps. First the match conditions are setup for accepting/rejecting packets in the flow table (listing 5.47). Four match fields are set to match packets that have the new oxm type `dpkm_method`, arrive on the default switch port, are type ipv4 (IP packets), and have the destination IP address of the peer switch.

```
1  Match dpkmMatch = peerA.getOFFactory().buildMatch()
2          .setExact(MatchField.DPKM_METHOD, U8.of((short) 1))
3          .setExact(MatchField.IN_PORT, OFPort.LOCAL)
4          .setExact(MatchField.ETH_TYPE, EthType.IPv4)
5          .setExact(MatchField.IPV4_DST, IPv4Address.of(ipv4B))
6          .build();
```

Listing 5.47: Code extract from the constructFlowAdd method building a match object with four flow matching conditions, taken from the DpkmFlows.java class.

Next a list of actions is created for OVS to apply on any matching packets (listing 5.48). Three actions are built in `constructFlowAdd` and added to an array list `actionList`. The two actions `setIpSrc` and `setIpDst` change the ipv4 source and destination fields of the packets to the addresses of the WireGuard interfaces. This allows packets from OVS to be accepted by WireGuard since the source/destination fields match the IP addresses of both peer interfaces. The third action `output` sets the forwarding behaviour and tells OVS to send packets out of port number 1 – the port containing the wg0 interface. This is key to routing any matching packets through the WireGuard tunnel.

```
ArrayList<OFAction> actionList = new ArrayList<OFAction>();
    OFActions actions = peerA.getOFFactory().actions();
    OFOxms oxms = peerA.getOFFactory().oxms();
    // Modify ipv4 source field in packet to be WG source address.
    OFActionSetField setIpSrc = actions.buildSetField()
        .setField(
            oxms.buildIpv4Src()
            .setValue(IPv4Address
                .of(getIp(peerA.getId().toString(),true)))
            .build()).build();
    actionList.add(setIpSrc);
    // Modify ipv4 dest field in packet to be WG dest address.
    OFActionSetField setIpDst = actions.buildSetField()
        .setField(
            oxms.buildIpv4Dst()
            .setValue(IPv4Address
                .of(getIp(peerB.getId().toString(),true)))
            .build()).build();
    actionList.add(setIpDst);
    // Set action as output aka output on WireGuard port.
    OFActionOutput output = actions.buildOutput()
        .setMaxLen(0xffFFffFF)
        .setPort(OFPort.of(1))
        .build();
    actionList.add(output);
```

Listing 5.48: Code extract from the constructFlowAdd method defining three actions for OVS to perform on packets, taken from the DpkmFlows.java class.

To determine how OVS will apply the list of actions an instruction is needed (listing 5.49). A single `applyActions` instruction is used to instruct OVS to immediately apply the full action list in order to any matching packets.

```
OFInstructionApplyActions applyActions = peerA.getOFFactory()
        .instructions().buildApplyActions()
        .setActions(actionList)
        .build();
    ArrayList<OFInstruction> instructionList = new ArrayList<OFInstruction>();
    instructionList.add(applyActions);
```

Listing 5.49: Code extract from the constructFlowAdd method creating an instruction to apply all actions, taken from the DpkmFlows.java class.

Finally, `constructFlowAdd` builds and returns the full `flow_add` message using the match and actions/instructions created in the previous three steps (listing 5.50). The priority is set to a high value to ensure matching is attempted on this flow entry before any others and the timeout is set to 0 so the flow does not expire.

```
OFFlowAdd flow = peerA.getOFFactory().buildFlowAdd()
        .setOutPort(OFPort.of(1))
        .setBufferId(OFBufferId.NO_BUFFER)
        .setHardTimeout(0)
        .setIdleTimeout(0)
        .setPriority(32768)
        .setMatch(dpkmMatch)
        .setInstructions(instructionList)
        .build();
    return flow;
```

Listing 5.50: Code extract from the constructFlowAdd method building and returning the flow_add object, taken from the DpkmFlows.java class.

The controller sends one `flow_add` message to each peer node over OpenFlow and uses `updatePeerInfo` to set the connection status to `COMMUNICATING`. Figure 5.26 displays a captured `flow_add` message in Wireshark, sent from the controller (192.168.0.9) to node A (192.168.0.8), followed by a second to node B (192.168.0.13). The message contains the full lists of oxm match fields, instructions, and actions. Flow_add messages are controller-to-switch messages that do not require a response [ONF13], meaning as far as the controller is concerned encrypted communication has now started.

```
1322… 482.102895697 192.168.0.9    192.168.0.8    OF 1.4    212 of_flow_add
1322… 482.102922047 192.168.0.9    192.168.0.13   OF 1.4    212 of_flow_add
1327… 484.105318355 192.168.0.9    192.168.0.8    OF 1.4     76 of_echo_request

    length: 144
    xid: 632
    cookie: 0
    cookie_mask: 0
    table_id: 0
    _command: 0
    idle_timeout: 0
    hard_timeout: 0
    priority: 32768
    buffer_id: 4294967295
    out_port: 1
    out_group: 4294967295
    flags: Unknown (0x00000000)
    importance: 0
  ▼ of_match
      type: OFPMT_OXM (1)
      length: 35
    ▼ of_oxm list
      ▶ of_oxm_in_port
      ▶ of_oxm_eth_type
      ▶ of_oxm_ipv4_dst
      ▶ of_oxm_dpkm_method
  ▼ of_instruction list
    ▼ of_instruction_apply_actions
        type: OFPIT_APPLY_ACTIONS (0x00000004)
        len: 56
      ▼ of_action list
        ▼ of_action_set_field
            type: OFPAT_SET_FIELD (25)
            len: 16
          ▼ of_oxm_ipv4_src
              type_len: 2147489284
              value: 10.0.0.1
        ▼ of_action_set_field
            type: OFPAT_SET_FIELD (25)
            len: 16
          ▼ of_oxm_ipv4_dst
              type_len: 2147489796
              value: 10.0.0.2
        ▼ of_action_output
            type: OFPAT_OUTPUT (0)
            len: 16
            port: 1
            max_len: 65535
```

Figure 5.26: Screenshot of the Wireshark packet analyzer, displaying the full content of a captured flow_add message for adding a new flow entry to node A.

Running the command – `ovs-ofctl dump-flows s1` – on the node A console (figure 5.27) confirms that the flow has been successfully setup. The flow entry modifies the source/destination ip addresses of any DPKM, IP packets destined to node B (192.168.0.13) to 10.0.0.1 (WGA) and 10.0.0.2 (WGB) and forwards them through interface wg0. The tool `ovs-appctl ofproto/trace` can be used for testing and tracking packet flow in OVS by simulating the handling of a packet in the switch flow table. Figure 5.28 traces the DPKM flow by mocking a packet containing all four match criteria, demonstrating that OVS applies all required actions and forwards matching packets through WireGuard as expected. The nodes have now transitioned from the *"Tunnel Ready"* to *"Communicating"* state (figure 4.15) and are fully setup.

```
luke@luke-nodea:~$ sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=642.252s, table=0, n_packets=0, n_bytes=0, dpkm_method=1,ip,in_port=
LOCAL,nw_dst=192.168.0.13 actions=mod_nw_src:10.0.0.1,mod_nw_dst:10.0.0.2,output:wg0
```

Figure 5.27: Screenshot of the node A console confirming setup of the flow for encrypting and routing packets through interface wg0.

```
luke@luke-nodea:~$ sudo ovs-appctl ofproto/trace s1 dpkm_method=1,in_port=LOCAL,ip,nw_dst=
192.168.0.13
Flow: dpkm_method=1,ip,in_port=LOCAL,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_dst=00:00
:00:00:00:00,nw_src=0.0.0.0,nw_dst=192.168.0.13,nw_proto=0,nw_tos=0,nw_ecn=0,nw_ttl=0

bridge("s1")
------------
 0. dpkm_method=1,ip,in_port=LOCAL,nw_dst=192.168.0.13, priority 32768
    set_field:10.0.0.1->ip_src
    set_field:10.0.0.2->ip_dst
    output:1

Final flow: dpkm_method=1,ip,in_port=LOCAL,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_dst
=00:00:00:00:00:00,nw_src=10.0.0.1,nw_dst=10.0.0.2,nw_proto=0,nw_tos=0,nw_ecn=0,nw_ttl=0
Megaflow: recirc_id=0,dpkm_method=1,eth,ip,in_port=LOCAL,dl_src=00:00:00:00:00:00,dl_dst=0
0:00:00:00:00:00,nw_src=0.0.0.0,nw_dst=192.168.0.13,nw_proto=0,nw_frag=no
Datapath actions: 2
```

Figure 5.28: Screenshot of the node A console after running the ofproto/trace command to simulate arrival of a packet matching the flow entry conditions.

5.4.6 Ending Communication

The *End Communication* procedure (4.2.6) terminates encrypted node-to-node communication and tears down the WireGuard tunnel connecting two nodes. Two variations have been implemented, based on the sequence diagram (4.21), that construct different `flow_mod` messages to modify the flow established in the *Start Communication* procedure (5.4.5). The first variation ends encrypted communication only and instructs OVS to route packets unencrypted via the controller. The second variation terminates all communication by instructing OVS to drop all DPKM packets destined to the peer. In both variations the *Delete Peer* procedure (5.4.4) is used to remove peer information from the interfaces, deleting the WireGuard tunnel. Communication can be ended externally from the application UI or internally by the controller during the *Revocation* procedure (5.4.8) for any connections in the `COMMUNICATING` state. Clicking the "End" button in the connected peers table displays the modal form in figure 5.29 with the connection id and a checkbox for selecting the variation of the procedure.

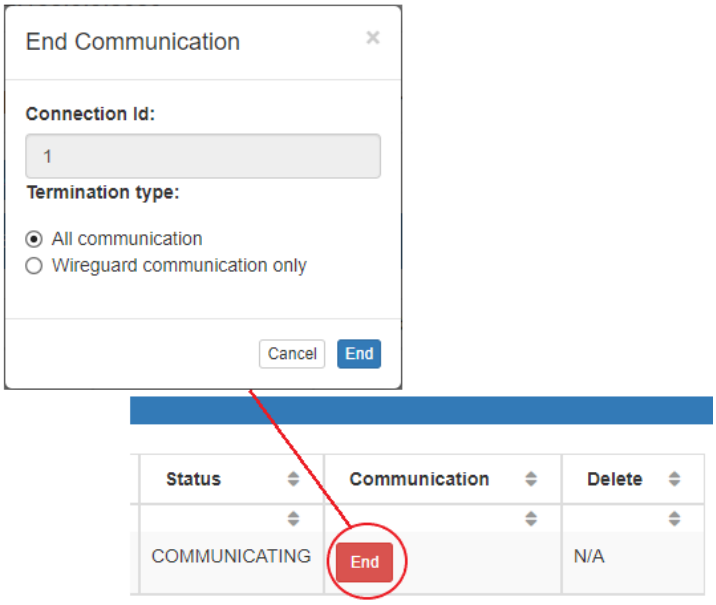

Figure 5.29: Screenshot captured from the application UI, displaying the modal popup form for ending communication toggled by clicking the delete button in the connected peers table.

Clicking "End" submits the modal form, passing the cid and a value endType to the `EndCommunication` function. Based on the selected option endType is either endWG or endAll and is used to initiate the corresponding variation. A `POST` request containing the json data is made to the REST endpoint `/wm/dpkm/end/json` registered to the `end` method of the DpkmEndCommunicationResource class. End deserializes the json into a DpkmPeers object, returns an error if the peers are not connected/communicating, and passes the two dpid's and endType to the `endCommunication` method (listing 5.51).

```
@Override
    public void endCommunication(String dpidA, String dpidB, String endType) {
      // Initialise the switches.
    IOFSwitch peerA = switchService.getSwitch(DatapathId.of(dpidA));
    IOFSwitch peerB = switchService.getSwitch(DatapathId.of(dpidB));

    // Continue communication unencrypted else end all.
    if(endType.equalsIgnoreCase("endWG")) {
      OFFlowModify flowA = constructFlowModify(peerA, peerB);
      OFFlowModify flowB = constructFlowModify(peerB, peerA);
      // Write FLOW_MOD (MODIFY) messages to switches.
      peerA.write(flowA);
      peerB.write(flowB);
      log.info(String.format("Communication between switch %s and switch %s now unencrypted.",
          dpidA, dpidB));
```

```
15      } else {
16        OFFlowModify flowA = constructFlowDrop(peerA, peerB);
17        OFFlowModify flowB = constructFlowDrop(peerB, peerA);
18        // Write FLOW_MOD (DROP) messages to switches.
19        peerA.write(flowA);
20        peerB.write(flowB);
21        log.info(String.format("Communication between switch %s and switch %s ended.",dpidA,
              dpidB));
22      }
23      // Delete peer connection one interface at a time.
24      sendDeletePeerMessage(dpidA, dpidB, false);
25      }
```

Listing 5.51: endCommunication method for sending a flow_mod message to each peer, instructing OVS to either route packets via the controller or drop packets. Taken from DpkmConfigureWG.java.

The controller instantiates the two switches from the dpids and uses endType as a conditional for constructing and sending two different types of `flow_mod` messages. If communication should continue unencrypted (endWG) two `flow_mods` are constructed by `constructFlowModify` (listing 5.52) in the DpkmFlows class. The `constructFlowModify` method creates a new list of actions with a single output action for forwarding packets through the port connected to the controller. An `applyActions` instruction is defined to apply the action to any matching packets. The `constructFlowAdd` method used during the *Start Communication* procedure (5.4.5) is called to recreate a `flow_add` object with the same match conditions as the existing encrypted flow. The instruction list contained in the `flow_add` object is replaced with the new instruction/action list for directing packets via the controller port instead of wg0. To ensure the existing flow entry is overwritten the `flow_add` object needs to be converted into a `flow_mod` object. Loxigen provides a useful flow conversion interface that can transform flow messages to any other type and is used for this purpose. Finally, the `flow_mod` object is returned and is ready to be sent by the controller.

To fully terminate communication by dropping all packets between the peers (endAll) two different `flow_mods` are assembled by `constructFlowDrop` (listing 5.53) in the DpkmFlows class. OpenFlow does not explicitly support a drop action but OVS has been designed to drop any packets with no output actions or a clear-actions instruction [ONF13]. A `clearActions` instruction is built ordering OVS to remove all actions from and drop matched packets. As in `constructFlowModify`, the `flow_add` object is recreated and the instruction list is replaced, keeping the OXM match conditions. The Loxigen `FlowModUtils` flow conversion interface is used to transform the `flow_add` into a `flow_mod` object, returning it to the controller.

```
1  protected OFFlowModify constructFlowModify(IOFSwitch peerA, IOFSwitch peerB) {
2      // Create new actions for switch.
3      ArrayList<OFAction> actionList = new ArrayList<OFAction>();
4      OFActions actions = peerA.getOFFactory().actions();
5      // Set action to output via normal port.
6      OFActionOutput output = actions.buildOutput()
7          .setMaxLen(0xffFFffFF)
8          .setPort(OFPort.CONTROLLER)
9          .build();
10     actionList.add(output);
11     // Write instruction to apply action list to packet on switch end.
12     OFInstructionApplyActions applyActions = peerA.getOFFactory()
13         .instructions().buildApplyActions()
14         .setActions(actionList)
15         .build();
16     ArrayList<OFInstruction> instructionList = new ArrayList<OFInstruction>();
17     instructionList.add(applyActions);
18     // Recreate identical flow to the existing.
19     OFFlowAdd flow = constructFlowAdd(peerA, peerB);
20     // Use the flow as a builder to update port.
21     OFFlowAdd newFlow = flow.createBuilder()
22         .setInstructions(instructionList)
23         .build();
24
25     OFFlowModify flowModify = FlowModUtils.toFlowModify(newFlow);
26     return flowModify;
27   }
```

Listing 5.52: Method for creating a flow_mod message that modifies the existing flow to route packets via the controller. Taken from the DpkmFlows.java class.

```
1  protected OFFlowModify constructFlowDrop(IOFSwitch peerA, IOFSwitch peerB) {
2      // Instruction to clear action list on switch end, dropping packet.
3      OFInstructionClearActions clearActions = peerA.getOFFactory()
4          .instructions().clearActions();
5      ArrayList<OFInstruction> instructionList = new ArrayList<OFInstruction>();
6      instructionList.add(clearActions);
7      // Recreate identical flow to the existing.
8      OFFlowAdd flow = constructFlowAdd(peerA, peerB);
9      // Use the flow as a builder to update instruction.
10     OFFlowAdd newFlow = flow.createBuilder()
11         .setInstructions(instructionList)
12         .build();
13
14     OFFlowModify flowModify = FlowModUtils.toFlowModify(newFlow);
15     return flowModify;
16   }
```

Listing 5.53: constructFlowDrop method for creating a flow_mod message modifying the existing flow with a new instruction that drops any matching packets. Taken from the DpkmFlows.java class.

The controller sends one `flow_mod` message to each peer node, followed by an `of_dpkm_delete_peer` message to remove the connection and update the database. Figure 5.30 presents a captured `flow_mod` message with an action routing packets via the controller, sent from the controller (192.168.0.9) to node A (192.168.0.8), followed by a second to node B (192.168.0.13), and two delete peer requests.

```
58564 21.011236366  192.168.0.9     192.168.0.8     OF 1.4   180 of_flow_modify
58565 21.011288103  192.168.0.9     192.168.0.13    OF 1.4   180 of_flow_modify
58649 21.020000346  192.168.0.9     192.168.0.8     OF 1.4   412 of_dpkm_delete_peer
58713 21.037247197  192.168.0.8     192.168.0.9     OF 1.4   440 of_dpkm_status
59555 21.122122797  192.168.0.9     192.168.0.13    OF 1.4   412 of_dpkm_delete_peer
59698 21.148723359  192.168.0.13    192.168.0.9     OF 1.4   440 of_dpkm_status
65805 23.021014404  192.168.0.9     192.168.0.8     OF 1.4    76 of_echo_request

OpenFlow (LOXI)
    version: 5
    type: 14
    length: 112
    xid: 432
    cookie: 0
    cookie_mask: 0
    table_id: 0
    _command: 1
    idle_timeout: 0
    hard_timeout: 0
    priority: 32768
    buffer_id: 4294967295
    out_port: 1
    out_group: 4294967295
    flags: Unknown (0x00000000)
    importance: 0
  ▼ of_match
      type: OFPMT_OXM (1)
      length: 35
    ▶ of_oxm list
  ▼ of_instruction list
    ▼ of_instruction_apply_actions
        type: OFPIT_APPLY_ACTIONS (0x00000004)
        len: 24
      ▼ of_action list
        ▼ of_action_output
            type: OFPAT_OUTPUT (0)
            len: 16
            port: 4294967293
            max_len: 65535
```

Figure 5.30: Screenshot of the Wireshark packet analyzer, displaying the full content of a captured flow_mod message for modifying the flow table of node A.

Tracing the packet flow with the `ofproto/trace` tool on the node A console (figure 5.31) confirms that any packets satisfying the match conditions set during the *Start Communication* procedure (5.4.5) will be forwarded to the controller as expected. No flow entries routing packets via WireGuard remain and the interfaces are identical to figure 5.24 (after deleting a peer). The nodes have now transitioned from the *"Communicating"* to *"WireGuard Configured"* state (figure 4.15). This variation can only be undone by running the *Add Peer* and *Start Communication* procedures to reconstruct the peer connection and flow.

```
luke@luke-nodea:~$ sudo ovs-appctl ofproto/trace s1 dpkm_method=1,in_port=LOCAL,ip,nw_dst=
192.168.0.13
Flow: dpkm_method=1,ip,in_port=LOCAL,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_dst=00:00
:00:00:00:00,nw_src=0.0.0.0,nw_dst=192.168.0.13,nw_proto=0,nw_tos=0,nw_ecn=0,nw_ttl=0

bridge("s1")
------------
 0. dpkm_method=1,ip,in_port=LOCAL,nw_dst=192.168.0.13, priority 32768
    CONTROLLER:65535

Final flow: unchanged
Megaflow: recirc_id=0,dpkm_method=1,eth,ip,in_port=LOCAL,dl_src=00:00:00:00:00:00,dl_dst=0
0:00:00:00:00:00,nw_src=0.0.0.0/1,nw_dst=192.168.0.13,nw_proto=0,nw_frag=no
Datapath actions: userspace(pid=4081628219,controller(reason=1,dont_send=1,continuation=0,
recirc_id=14,rule_cookie=0,controller_id=0,max_len=65535))
```

Figure 5.31: Screenshot of the node A console after running the ofproto/trace command to simulate arrival of a packet matching the unencrypted communication flow conditions.

Figure 5.32 displays a captured `flow_mod` message with an instruction for dropping all packets, sent from the controller (192.168.0.9) to node B (192.168.0.13), followed by a second to node A (192.168.0.8), and two delete peer requests.

```
51151 17.566879258  192.168.0.9     192.168.0.13    OF 1.4   164 of_flow_modify
51152 17.566906439  192.168.0.9     192.168.0.8     OF 1.4   164 of_flow_modify
51244 17.578315954  192.168.0.9     192.168.0.8     OF 1.4   412 of_dpkm_delete_peer
51328 17.604102449  192.168.0.8     192.168.0.9     OF 1.4   440 of_dpkm_status
52107 17.722053824  192.168.0.9     192.168.0.13    OF 1.4   412 of_dpkm_delete_peer
52207 17.735393096  192.168.0.13    192.168.0.9     OF 1.4   440 of_dpkm_status
58076 19.579113991  192.168.0.9     192.168.0.8     OF 1.4    76 of_echo_request

Internet Protocol Version 4, Src: 192.168.0.9, Dst: 192.168.0.13
Transmission Control Protocol, Src Port: 6653, Dst Port: 55198, Seq: 685, Ack: 65, Len: 96
OpenFlow (LOXI)
   version: 5
   type: 14
   length: 96
   xid: 78
   cookie: 0
   cookie_mask: 0
   table_id: 0
   _command: 1
   idle_timeout: 0
   hard_timeout: 0
   priority: 32768
   buffer_id: 4294967295
   out_port: 1
   out_group: 4294967295
   flags: Unknown (0x00000000)
   importance: 0
▼ of_match
      type: OFPMT_OXM (1)
      length: 35
   ▼ of_oxm list
      ▶ of_oxm_in_port
      ▶ of_oxm_eth_type
      ▶ of_oxm_ipv4_dst
      ▶ of_oxm_dpkm_method
▼ of_instruction list
   ▼ of_instruction_clear_actions
         type: 5
         len: 8
```

Figure 5.32: Screenshot of the Wireshark packet analyzer, displaying the full content of a captured flow_mod message for modifying the flow table of node B.

Utilising the `ofproto/trace tool` again to simulate the packet flow on the node A console (figure 5.33) confirms that any packets satisfying the match criteria will now be dropped by OVS. Whilst this flow entry exists all future communication between node A and node B is prohibited, transitioning the nodes from the *"Communicating"* to *"Node Waiting"* state (figure 4.15). Restoring communication requires either a `flow_delete` message to remove the flow entry or reconfiguration of the nodes, transitioning to the *"WireGuard Configured"* state, followed by the procedures discussed above to return to the *"Communicating"* state.

```
luke@luke-nodea:~$ sudo ovs-appctl ofproto/trace s1 dpkm_method=1,in_port=LOCAL,ip,nw_dst=
192.168.0.13
Flow: dpkm_method=1,ip,in_port=LOCAL,vlan_tci=0x0000,dl_src=00:00:00:00:00:00,dl_dst=00:00
:00:00:00:00,nw_src=0.0.0.0,nw_dst=192.168.0.13,nw_proto=0,nw_tos=0,nw_ecn=0,nw_ttl=0

bridge("s1")
------------
 0. dpkm_method=1,ip,in_port=LOCAL,nw_dst=192.168.0.13, priority 32768
    clear_actions
     -> action set was empty

Final flow: unchanged
Megaflow: recirc_id=0,dpkm_method=1,eth,ip,in_port=LOCAL,dl_src=00:00:00:00:00:00,dl_dst=0
0:00:00:00:00:00,nw_src=0.0.0.0/1,nw_dst=192.168.0.13,nw_proto=0,nw_frag=no
Datapath actions: drop
```

Figure 5.33: Screenshot of the node A console after running the ofproto/trace command to simulate arrival of a packet matching the terminate communication flow conditions.

5.4.7 Rekeying

The *Rekeying* procedure (4.2.7) replaces the keys bound to a WireGuard interface after expiry of the cryptoperiod lifespan, which is set during the *Configure WireGuard* procedure (5.4.1). Rekeying of the session keys is the responsibility of WireGuard, meaning only rekeying functionality for the asymmetric keys was developed in DPKM. Implementation of this procedure was challenging and caused synchronicity issues between communicating peers at earlier stages. Much of the system is built around request/response message sequences, meaning rekeying two communicating nodes at the same time confused the controller of the system state, causing race conditions that led to sending too many or too little add/delete peer messages. This was solved by enforcing a half second timeout in between each rekey request, allowing full handling of one node before moving on to another.

Although rekeying was implemented as planned the developer feels it serves little practical purpose. The procedure replaces asymmetric Curve25519 keys that are almost unbreakable and have a recommended lifespan of 1-2 years according to NIST [Gir20]. The DPKM system

could arguably function securely without this procedure, manually changing the keys when needed. On the other hand, this procedure has a positive impact on security by providing key freshness and causing minimal interruption if set to be infrequent (high cryptoperiod). These points will be discussed further in the evaluation chapter (6).

Unlike the originally designed sequence of functionality (4.22), rekeying is automatically initiated by the application instead of the controller. A JavaScript approach was developed to visually display a countdown of the cryptoperiod for each node in the configured switches table. Creating a countdown in the Floodlight codebase was initially attempted but abandoned due to the excess number of database calls required to continuously check for expiry. The implemented approach calculates expiry when refreshing the table by adding the cryptoperiod to the `datetime` last configured (since), subtracting the current `datetime`, and calling the `Rekey` function with the dpid if the resulting value is zero or negative. This approach does not increase the number of `GET` requests since retrieving switch information is part of the standard operation. The administrator is informed of expiry and initiation of the rekeying procedure by changing the cryptoperiod column to "Expired" and displaying a popup notification (figure 5.34).

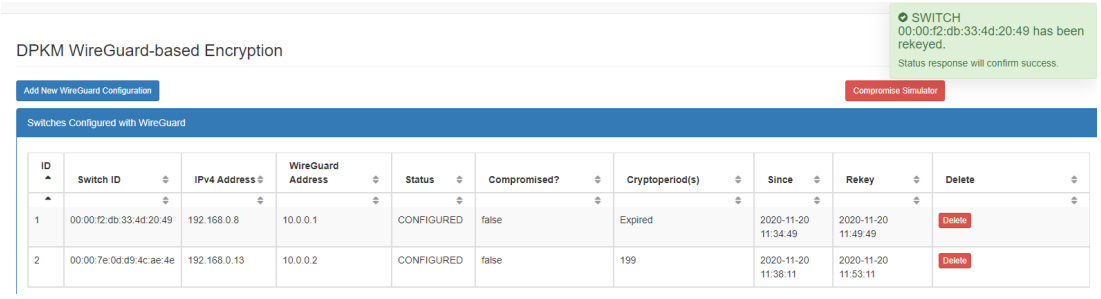

Figure 5.34: Screenshot of the application UI showing expiry of the node A cryptoperiod and automated initiating of the rekey procedure, indicated by the popup notification.

A `POST` request is made to the REST endpoint `/wm/dpkm/rekey/json` registered to the DpkmRekeyResource class. Near identical code to the `configure` method (listing 5.24) is used but calls the `rekey` method in the DpkmConfigureWG class. Rekey uses a series of conditional statements to mark all connections to the node as `KEY CHANGED` or `BOTH CHANGED` (if rekeying two interfaces at once). If any nodes are communicating the `handleFlowRekeying` method (listing 5.54) in the DpkmFlows class is used to manage the flow. When `handleFlowRekeying` is called before rekeying, the `constructFlowModify` method (listing 5.52), presented in the *End Communication* procedure (5.4.6), is used to temporarily redirect packets via the

controller instead of through WireGuard, preventing packet loss during reconfiguration. In all cases the standard *Configure WireGuard* procedure (5.4.1) is initiated to reconfigure the interface.

```
protected void handleFlowRekeying(IOFSwitch peerA, IOFSwitch peerB, boolean before) {

    if(before) {
      OFFlowModify flowA = constructFlowModify(peerA,peerB);
      OFFlowModify flowB = constructFlowModify(peerB,peerA);
      peerA.write(flowA);
      peerB.write(flowB);
      log.warn(String.format(
          "Communication between switch %s and switch %s temporarily unencrypted.",
          peerA.getId().toString(),peerB.getId().toString()));
    } else {
      OFFlowAdd flowAddA = constructFlowAdd(peerA,peerB);
      OFFlowAdd flowAddB = constructFlowAdd(peerB,peerA);
      OFFlowModify flowA = FlowModUtils.toFlowModify(flowAddA);
      OFFlowModify flowB = FlowModUtils.toFlowModify(flowAddB);
      peerA.write(flowA);
      peerB.write(flowB);
      log.info(String.format(
          "Communication between switch %s and switch %s restored.",
          peerA.getId().toString(),peerB.getId().toString()));
    }
  }
```

Listing 5.54: handleFlowRekeying method for creating a flow_mod message to route packets via the controller before rekeying and restore the encrypted flow after rekeying. Taken from the DpkmFlows.java class.

Figures 5.35 and 5.36 show the full message exchanges in Wireshark and the controller log after rekeying node A. The message sequence can be broken down into the following steps:

- If communicating, controller (192.168.0.9) modifies the flow tables of node A (192.168.0.8) and node B (192.168.0.13) to stop sending packets through WireGuard.
- Controller sends an `of_dpkm_set_key` message to node A, reconfiguring the interface.
- Controller processes set key status response with `processConfigured` (listing 5.29), checking for connections and using the connection status to execute subsequent procedures.
- Controller updates connection A-B to `PID1ONLY` and sends an `of_dpkm_delete_peer` message to node B, removing old public key A from the interface.

- Controller processes delete peer status response with `processPeerRemoved` (listing 5.45), sending an `of_dpkm_add_peer` message to node B with new public key A.
- Controller processes add peer status response with `processPeerAdded`, updating connection A-B to `CONNECTED` or `COMMUNICATING`.
- If previously communicating, controller calls `handleFlowRekeying` (listing 5.54), restoring the encrypted flow.

Successful completion of the *Rekey* procedure maintains the state of the node but updates the interfaces with new key and peer information. The whole process is automated and does not require any interaction from the administrator.

17142	5.464463405	192.168.0.9	192.168.0.13	OF 1.4	180	of_flow_modify
17143	5.464488462	192.168.0.9	192.168.0.8	OF 1.4	180	of_flow_modify
17144	5.464588239	192.168.0.9	192.168.0.8	OF 1.4	92	of_dpkm_set_key
17155	5.470203331	192.168.0.8	192.168.0.9	OF 1.4	440	of_dpkm_status
18075	5.571737285	192.168.0.9	192.168.0.13	OF 1.4	412	of_dpkm_delete_peer
18136	5.588312076	192.168.0.13	192.168.0.9	OF 1.4	440	of_dpkm_status
19012	5.699407204	192.168.0.9	192.168.0.13	OF 1.4	412	of_dpkm_add_peer
19026	5.702210355	192.168.0.13	192.168.0.9	OF 1.4	440	of_dpkm_status
20607	5.854447528	192.168.0.9	192.168.0.8	OF 1.4	212	of_flow_modify
20608	5.854551046	192.168.0.9	192.168.0.13	OF 1.4	212	of_flow_modify

```
▸ Frame 17144: 92 bytes on wire (736 bits), 92 bytes captured (736 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.0.9, Dst: 192.168.0.8
▸ Transmission Control Protocol, Src Port: 6653, Dst Port: 41494, Seq: 375, Ack: 25, Len: 24
▾ OpenFlow (LOXI)
    version: 5
    type: 4
    length: 24
    xid: 473
    experimenter: 2718565155
    subtype: 0
```

Figure 5.35: Screenshot of the Wireshark packet analyzer, displaying the full message exchanges of the rekeying procedure.

```
2020-11-24 14:20:21.482 WARN  [n.f.d.DpkmConfigureWGResource] Communication between switch 00:00:16:21:6f:3d:33:42 and switch 00:00:e2:96:f9:85:99:46 temporarily unencrypted.
2020-11-24 14:20:21.482 INFO  [n.f.d.DpkmConfigureWG] DPKM_SET_KEY message sent to switch 00:00:e2:96:f9:85:99:46
2020-11-24 14:20:21.497 INFO  [n.f.d.DpkmConfigureWG] Switch 00:00:e2:96:f9:85:99:46 has been configured successfully.
2020-11-24 14:20:21.590 INFO  [n.f.d.DpkmConfigureWG] DPKM_DELETE_PEER message sent to switch 00:00:16:21:6f:3d:33:42
2020-11-24 14:20:21.625 INFO  [n.f.d.DpkmConfigureWG] Switch 00:00:16:21:6f:3d:33:42 has successfully removed 00:00:e2:96:f9:85:99:46 as a WG peer.
2020-11-24 14:20:21.717 INFO  [n.f.d.DpkmConfigureWG] DPKM_ADD_PEER message sent to switch 00:00:16:21:6f:3d:33:42
2020-11-24 14:20:21.739 INFO  [n.f.d.DpkmConfigureWG] Switch 00:00:16:21:6f:3d:33:42 has successfully added 00:00:e2:96:f9:85:99:46 as a WG peer.
2020-11-24 14:20:21.775 INFO  [n.f.d.DpkmConfigureWG] Full connection established between switch 00:00:16:21:6f:3d:33:42 and switch 00:00:e2:96:f9:85:99:46.
2020-11-24 14:20:21.872 INFO  [n.f.d.DpkmConfigureWGResource] Communication between switch 00:00:16:21:6f:3d:33:42 and switch 00:00:e2:96:f9:85:99:46 restored.
```

Figure 5.36: Controller log reporting on state changes and successful completion of DPKM functions during the rekeying procedure.

5.4.8 Compromise and Revocation

The *Revocation* procedure (4.2.8) handles a compromised node by revoking its keys to mitigate the damage caused by a security breach. Two revocation types have been developed, based on the sequence diagram (4.23), that either reconfigure the node using the *Configure WireGuard* procedure (5.4.1) or terminate the node using the *Unconfigure WireGuard* procedure (5.4.2). In practise, detection of a compromise should be the responsibility of a third-party security software entity but implementation of such a device was beyond the scope of this thesis (1.2). Instead a compromise simulator was developed that performs these duties by allowing the tester to mark a node as compromised, initiating the revocation process.

Simulating a compromise begins at the application UI by clicking the "Compromise Simulator" button to toggle a modal popup form and enter the row id of a node (figure 5.37). Submitting the form passes the id to the `CompromiseNode` function which makes a `POST` request to the REST endpoint `/wm/dpkm/compromise/json` registered to the `compromise` method of the DpkmCompromiseNodeResource class. Compromise calls the `compromiseNode` method in the main DpkmConfigureWG class which marks the node by setting the Boolean `Compromised` field in the ConfiguredPeers table to `true`.

The *End Communication* procedure (5.4.6) is then initiated with the terminate all communication variation, modifying the flow to drop any packets to/from the compromised node. The message sequence resembles figure 5.32, sending `flow_mod` and delete peer messages to the node and any peers. A popup notification is displayed (figure 5.38) indicating that the system environment has been prepared for testing the revocation procedure.

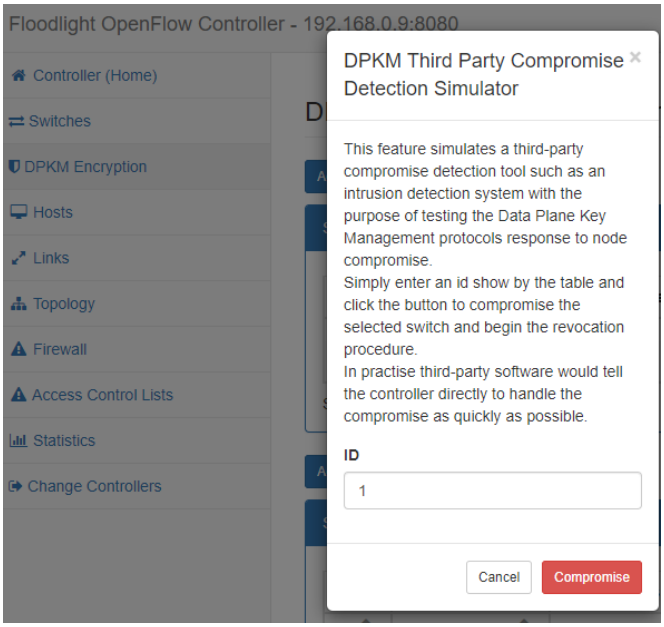

Figure 5.37: Screenshot captured from the application UI, with the modal popup form for simulating a compromise.

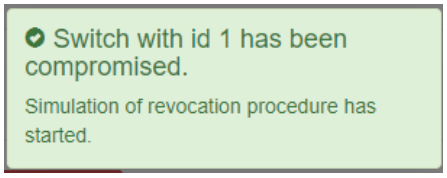

Figure 5.38: Popup notification indicating commencement of the key revocation simulator

When the configured switches table is loaded or refreshed on the UI the `GetWGSwitches` function populating the table checks the value of the `compromised` field. If compromised is true, `HandleCompromiseModal` is called to toggle the modal form in figure 5.39, with the switch dpid and two options for handling revocation. The administrator is forced to act by preventing the form from being closed, fixing it over the display, and blocking access to all other UI components. The form will only close after submission by clicking "OK", passing the dpid and a value revType to the `HandleCompromise` function. Based on the selection revType is either reConf (reconfigure) or unConf (unconfigure). `HandleCompromise` makes a `POST` request to the REST endpoint `/wm/dpkm/revoke/json` registered to the `revoke` method of the DpkmRevocationResource class. The json is mapped to a DpkmSwitch object and uses IDpkmConfigureWGService to call the `revoke` method in the main DpkmConfigureWG class.

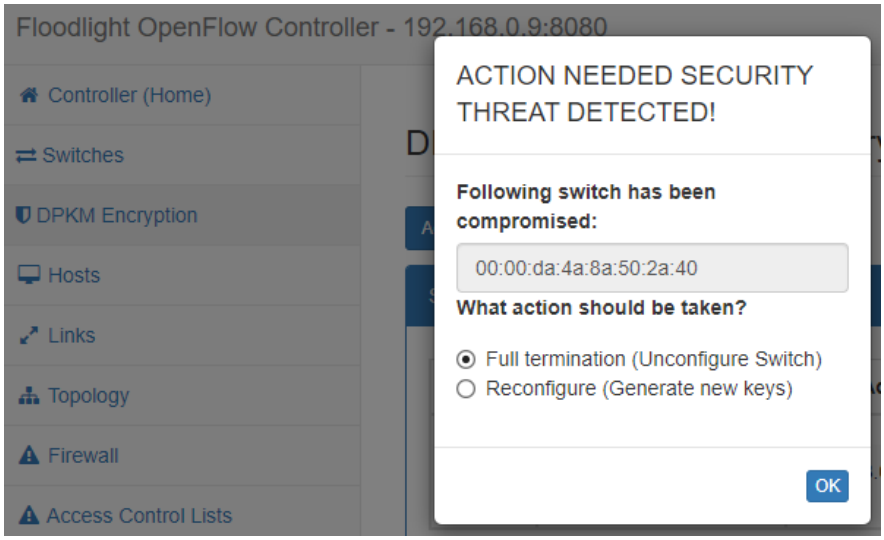

Figure 5.39: Screenshot captured from the application UI, displaying a fixed modal popup form that can only be closed by initiating the revocation procedure.

If the node should be reconfigured, the controller sends a set key message to initiate the *Configure WireGuard* procedure (5.4.1). This instructs the node to generate a new set of keys for the WireGuard interface and replace the insecure keys. The message exchanges resemble figure 5.4, overwriting the database record with the new public key contained in the status response and setting compromised to false. All peer connections are then rebuilt using the *Add Peer* procedure (5.4.3), completing this type of revocation. If the node should be fully terminated, the controller sends a delete key message to initiate the *Unconfigure WireGuard* procedure (5.4.2). The node removes all keys from the WireGuard interface and deletes all corresponding files, leaving a blank unconfigured interface. The message exchanges resemble figure 5.10, removing all record of the node upon receipt of the status response. The node is now in a *"Quarantined"* state and must be manually reintegrated into the system. This allows the administrator to fully contain the compromise until a thorough reset of the node can be performed.

5.5 Testing

Testing the DPKM system posed a challenge due to the lack of benchmarking tools suited to measuring performance of the novel software extensions. An unconventional approach was formulated by modifying the controller code to repeatedly loop procedures, using the tools Mininet [Min20], Wireshark [Wir20], and Microsoft Excel to generate, capture, and analyse data. This section breaks down the test environment and methodology to enable recreation of all experiments. The obtained results will be discussed in section 6.3 of the subsequent evaluation chapter.

5.5.1 Test Environment

Tests were conducted across the three VirtualBox VM's described in section 5.1, with the host PC distributing CPU's and RAM. Performance of the procedures was recorded for a basic SDN comprising of one controller, two nodes, and four hosts, connected via the OpenFlow protocol. The controller VM was allocated four processors, and the node VMs were allocated two each. The full system specs are as follows:

- **Host PC Specs:** AMD Ryzen 5 2600, 6 Cores 12 CPUs, 3400 MHz, 16GB RAM.
- **Controller VM (1):** Floodlight controller run from the Eclipse IDE. 4 CPUs, 8GB RAM, located in SSD storage.
- **Node VM (2):** OVS switch simulated in Mininet with two hosts, plus WireGuard integrated as a port. 2 CPUs, 2GB RAM, located in HDD storage.

5.5.2 Aims and Metrics

The main aims were to establish if the system is functioning as expected and calculate the average time needed to complete the procedures presented in 5.4. Eight different tests were conducted, running procedures repeatedly for 20 to 800 cycles – depending on the CPU intensity. Time was measured in seconds and converted to milliseconds, calculating both the individual average and combined average for the nodes. Any issues were logged automatically by the controller and reviewed after completion. The tests can be summarised as follows:

1. **Configure WireGuard:** Average time between sending a `dpkm_set_key` message and receiving a `dpkm_status` response. Run 400 times, 200 for each node. (5.4.1).
2. **Unconfigure WireGuard:** Average time between sending a `dpkm_delete_key` message and receiving a `dpkm_status` response. Only run 40 times, 20 for each node, due to the switch disconnect issue discussed in 5.4.2.
3. **Add Peer (Single Interface):** Average time between sending a `dpkm_add_peer` message and receiving a `dpkm_status` response. Run 800 times, 400 for each node. (5.4.3).
4. **Add Peer Procedure:** Average time to complete the entire add peer procedure, measuring difference between sending the first `dpkm_add_peer` message and receiving the last `dpkm_status` response. Run 400 times with both nodes. (5.4.3).
5. **Delete Peer (Single Interface):** Average time between sending a `dpkm_delete_peer` message and receiving a `dpkm_status` response. Run 800 times, 400 for each node. (5.4.4).
6. **Delete Peer Procedure:** Average time to complete the entire delete peer procedure, measuring difference between sending the first `dpkm_delete_peer` message and receiving the last `dpkm_status` response. Run 400 times with both nodes. (5.4.4).
7. **Full Setup:** Average time to set up a WireGuard tunnel between two unconfigured nodes and begin encrypted communication. Includes the procedures for configuring nodes (5.4.1), adding peers (5.4.3), and starting communication (5.4.5). CPU intensive if done repeatedly so run 20 times with both nodes.
8. **Full Teardown:** Average time to end encrypted communication and tear down the WireGuard tunnel. Includes the procedures for ending communication (5.4.6), deleting peers (5.4.4), and unconfiguring nodes (5.4.2). Run 20 times with both nodes.

5.5.3 Methodology

1. The DPKM message listener class (5.3.1) in Floodlight was modified to repeatedly perform the procedures necessary for each test. A count value was incremented, stopping execution after a set number of cycles. A pause of 20 seconds was enforced between each `dpkm_delete_key` response and sending the next message to overcome the disconnect issue in the unconfigure procedure. All other procedures were performed successively with no issue.

2. The network environment was set up by executing a custom Mininet topology script in both node VMs to launch two OVS switches, two WireGuard interfaces, and four hosts.
3. Wireshark was initiated with a filter that only captures the message sequences relevant to the running test.
4. Floodlight was launched and the test was initiated from the UI by performing the procedure as usual, causing a request/response loop to trigger until the count is reached.
5. Once the test has run for the full iterations the Wireshark session is stopped. The packet data containing the timings, source/destination IP, and message type is exported to a file in CSV format.
6. The data is pre-processed by the Power Query Excel tool and imported into a new spread sheet, removing any unwanted data types or packets.
7. For single request/response tests (1,2,3,5), the message request timings were moved into a column adjacent to the respective status response. The difference of each (in seconds) and the total average was calculated. Figure 5.40 pictures three of the excel formulas that accomplished this.
8. For tests involving the complete procedures (4,6), or an aggregation of multiple procedures (7,8), the difference between the first message request and the final message response was measured. For example, in the setup test the difference between the first `dpkm_set_key` message and the final `flow_add` message was calculated, followed by the total average.
9. After each test the system was reset, clearing the database, cleaning Mininet, and rebuilding the topology.
10. All results were compiled into a single table and plotted into multiple bar charts for better data visualisation (6.3).

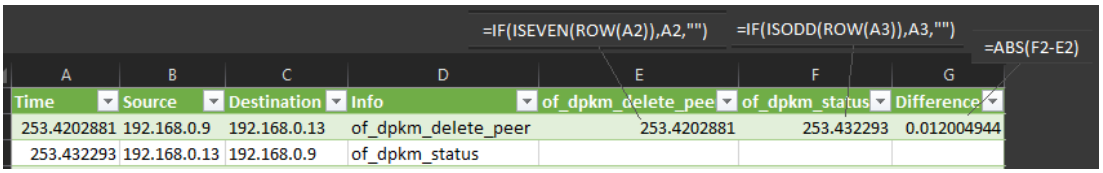

Figure 5.40: Screenshot captured from the excel spreadsheet, showcasing three of the excel formulas used to organise the data and calculate the time difference.

Chapter 6

Evaluation and Discussion

This chapter conducts a critical evaluation of the delivered Data Plane Key Management (DPKM) system, with reference to material from all previous chapters. Transparency is maintained by conducting an unbiased analysis of the project in satisfying or failing to satisfy the requirements, specifications, and designs. This chapter begins with section 6.1 where the implementation is analysed against the cryptographic and functional requirements presented in section 3.1. Next, section 6.2 reflects on the risks encountered during the development process, the effect on progress, and the employed mitigation strategies. Then, section 6.3 examines the results obtained from the testing of the DPKM procedures. Following this, section 6.4 outlines the achievements and enhancements within the field of study. Thereafter, section 6.5 provides a balanced evaluation by discussing the limitations and outstanding issues. Finally, section 6.6 proposes future directions to guide any researchers wishing to extend the content of this thesis.

6.1 Requirement Analysis

This section evaluates the completeness and correctness of the implementation documented in the previous chapter (5) against the cryptographic and functional requirements proposed in section 3.1. The cryptographic requirements (3.1.1) described the security related properties that DPKM should deliver and the functional requirements (3.1.2) outlined the operational aspects of the system. This section labels requirements that are fully satisfied with a tick (✓), mostly satisfied but lacking in areas with a tilde ($\sim$), and unsatisfied with a cross ($\times$). Overall, most requirements were fully satisfied, with five mostly satisfied areas and one unsatisfied area. These will be used for identifying limitations (6.5) and planning future development (6.6).

6.1.1 Cryptographic Requirements

- ✓ **Authenticity of Entities:** Entities in DPKM are mutually authenticated through existing security features that are assumed to be setup in advance. OpenFlow connections between Floodlight and OVS are authenticated by TLS using key stores containing asymmetric keys and certificates [Azo13]. REST connections between Floodlight and the application are authenticated through HTTPS using the same approach but with a different key store for isolation. Node-to-node connections are authenticated through the asymmetric cryptography and pre-shared keys used in WireGuard.
- ✓ **Authenticity of Keys:** Public keys are stored by the trusted controller alongside the respective identity of the node meaning authenticity of keys is protected by the authenticity of the entities.
- ✓ **Integrity:** Tamper-resistance and key integrity is achieved by only transporting public keys and using secure channels, storing private keys in protected files in the node, and only forming connections between configured and authenticated entities.
- ✓ **Confidentiality:** Communication is encrypted by TLS/SSL in REST/OpenFlow and ChaCha20Poly1305 in WireGuard (2.3.2).
- ✓ **Synchronicity:** All procedures operate in a request/response pattern and the controller maintains a global state by using status or error responses to update the connection/node status (5.4). Controller automatically sends messages based on the state, such as adding the second peer after the first or updating the interfaces after rekeying.
- ✓ **Freshness:** Session keys are automatically rekeyed by WireGuard after a message limit is reached or the session is 120 seconds old [Don17]. Other keys are periodically rekeyed by the controller after expiry of the cryptoperiod set during the configuration procedure (5.4.1). Key material is kept fresh and expired keys are removed from the system.
- ✓ **Context Bound Keys:** All keys are bound to the WireGuard interface during configuration and the IP address of the node. Session keys are generated and bound to specific peer connections. Public keys are not bound to connections but always correspond to the identity of a node.
- ∼ **Access Control:** Password protection is in place to prevent access to the node, key material, controller, and database. However, database passwords are currently stored in a plaintext file within Floodlight and there are no login mechanisms for the application UI. Additional measures are needed to fully satisfy this requirement such as file encryption

and better access authentication mechanisms.

- ✓ **Strength of Key:** DPKM uses 32-byte Curve25519 keys and the cryptographically secure hashing/encryption algorithms provided by WireGuard [Ber06; Don17]. It is not feasible to use brute force approaches to deduce the key with the technology available today.
- ✓ **Compromise causes minimal damage:** WireGuard provides perfect forward secrecy in its handshake by generating a new set of symmetric keys that cannot be broken by compromising old keys [DP18]. DPKM minimises the damage of a detected compromise by terminating all communication with the node and running the key revocation procedure (5.4.8).

6.1.2 Functional Requirements

- ✓ **32-byte key size:** DPKM generates 32-byte asymmetric keys in the node and successfully sends 32-byte public keys in messages over OpenFlow.
- ∼ **Maintain secrecy of private key:** Only public keys are transported by DPKM and private keys are stored in the node with strict access conditions. Requirement satisfied but could be enhanced further by encrypting the file.
- ✓ **Secure agreement of session keys:** WireGuard agrees on session keys in its 1-RT key exchange handshake based on the Noise protocol (2.3.2), with algorithms deemed cryptographically secure at the time of writing this thesis [Per16; DP18].
- ✓ **Rekey after expiry:** The rekeying procedure (5.4.7) successfully reconfigures the node after expiry of the cryptoperiod, restoring any connections or communication flows without interrupting other processes. Session keys are automatically rekeyed by WireGuard.
- ✓ **Randomness of key:** As described in section 2.3.2, WireGuard generates keys with the Curve25519 function which uses a Random Bit Generator (RBG) and ensures true randomness of the keys [Ber06].
- ✓ **Handle compromise:** A mechanism for detecting compromise was beyond the scope of this thesis (1.2) but when simulated the compromise is handled quickly by isolating the node, alerting the administrator with a message on the UI, and performing revocation (5.4.8).
- ∼ **Handle errors:** Fifteen new error messages were implemented that are returned by OVS and handled by the error listener class in Floodlight (5.3.1). Errors are logged in the

ErrorLog database table and the request is attempted a further two times before blocking the node. Further work is needed to add an error review/display tool to the DPKM UI.

- ∼ **Handle packet loss/delay:** OpenFlow uses TCP which ensures packets sent between the controller and node are delivered or restored if lost [Spe13]. Packet loss/delay does not impact key management processes since DPKM only updates the system state after successful completion and return of an operation. However, packets lost during node-to-node communication cannot be recovered due to WireGuard using UDP meaning this requirement can never be fully satisfied.
- × **Handle crash/restart:** No procedures were implemented for handling crashes or system restarts. Floodlight handles crashed nodes by disconnecting the OpenFlow connection and adjusting packet routing to use active nodes. This works for standard SDN but causes packet loss during encrypted node-to-node communication since WireGuard will continue sending packets to a crashed node. Future development is needed to introduce backup controllers for controller crashes, functionality for rerouting packets, and new procedures for setting up identical conditions on restart.
- ✓ **Intuitive UI:** The implemented DPKM UI labels and structures its visual components logically and could be understood by a new operator with little difficulty (5.3.3).
- ✓ **REST support:** Floodlight supports REST functionality for all DPKM procedures and provides APIs for external initiation from the UI (5.4).
- ✓ **OpenFlow support:** The OpenFlow protocol has been extended with the Loxigen tool for all planned extensions (5.2). Support has been added to both the Floodlight and OVS codebases, allowing DPKM-related controller-to-node communication.
- ∼ **Efficiency:** The DPKM system is highly efficient and feels instantaneous, constructing/destructing communication session and performing procedures in a fraction of a second. However, the system was only tested with two nodes and must therefore be retested with a larger network setup (5.5).
- ✓ **Responsiveness and Reliability:** All procedures trigger a status or error response to ensure reliability and responses are received within a second of the request.

6.2 Risk Analysis

Project risk is an event or condition that could have a negative impact on one or more of the project's objectives [Ins00]. Risks are an inevitable factor in every project, meaning the strength of the risk management strategies directly influence the probability of success. The project specification document preceding this thesis conducted a thorough risk analysis and formulated mitigation strategies for all risks. Table 6.1 presents the ten risks identified during the planning phase, with the likelihood of occurrence, severity of impact, and respective mitigation strategy. This section discusses these risks in order to reflect on the challenges faced during development of DPKM.

Risk	**Likelihood**	**Severity**	**Mitigation**
Instability of requirements	High	Low	Use of iterative and incremental SDLC anticipates changing requirements
Misunderstanding requirements	Low	Medium	Create extensive documentation and validation from project supervisor
Scale and Complexity	Medium	High	Build system in stages using tools, gradually increasing scale and complexity
Lack of support and documentation	Medium	Medium	Planning, Compilation of learning resources, Experimentation with tools, Replacement with well documented alternatives
Incompatible technology choices	Low	High	Attempting functionality with questionable compatibility early on, switch to alternatives
Inadequacy and availability of software and hardware	Low	High	Acquire access to tools early, find accessible alternatives
Delay from poor planning and organisation of schedule	Medium	Medium	Adjust timeline with progress, evaluation and planning in each iteration, Update estimates, Consult supervisor each week
Inefficient progress from unsuitable SDLC	Low	Medium	Deviation from SDLC to modified version or new approach
Loss of data	Low	High	Frequent backups on multiple devices
Hardware failure	Low	High	Borrow equipment from university until replaced

Table 6.1: Project risks, likelihood of occurrence, severity of impact, and mitigation strategy, taken from the preliminary specification document.

Instability of requirements

Changing requirements was a risk that had a high likelihood of occurring due to novelty of the project and need to incorporate new functionality during the development process. This risk

came into fruition but had a low impact severity because a SDLC was chosen that can accommodate changes in each iteration (3.3). Overall, this risk may have delayed the development timeline but did not jeopardise the project's success.

Misunderstanding requirements

Failing to understand the requirements was a risk that did not occur due to the requirements being created by the single person developing DPKM. Extensive documentation was produced in the planning and specification of the project, providing a clear guide for implementation.

Scale and Complexity

The scale and complexity of building DPKM was one of the biggest challenges and had a substantial impact on the delivered product. The quantity of software extensions and expected functionality was particularly daunting in the early stages of development. The sheer amount of work depending on a single developer combined with personal factors meant delay of the timeline and failure to meet the original targets. However, once the basic features were implemented the risk was mitigated by gradually increasing the functionality of the product. All procedures were added incrementally leading to a well-formed product that achieved most of its objectives and requirements (6.1). In retrospect this project may have been more suited to a team of developers since there is still room for improvement and new directions to be explored. These points will be highlighted in the future work section (6.6).

Lack of support and documentation

The novelty of this project meant limited resources were available for implementing the software extensions, requiring self-teaching and extensive study of raw code. This risk jeopardised successful delivery of the project and led to months of work without substantial progress. Adequate documentation was found for extending Floodlight, but learning resources for integrating OpenFlow extensions, extending OVS, and combining OVS and WireGuard were insufficient or inaccessible. Switching to alternative tools or a new project entirely became a serious consideration after studying all the relevant documentation, reaching out to the scientific community and contacts in industry, and still failing to make progress. A fresh approach was taken by attempting to integrate two new OpenFlow messages `dpkm_test_request` and `dpkm_test_response` in Floodlight and OVS. The test messages served no practical purpose but helped build knowledge in how to implement, process, and send new OpenFlow messages between the switch and controller. Through arduous compiler error debugging and manual

stack tracing the necessary class extensions and inner workings of OVS became apparent. This created momentum, leading to implementation of the `set_key` message and configuration procedure, followed by the subsequent procedures.

Incompatible technology choices

During the background research it became clear that the OVS and WireGuard technologies may have compatibility issues. As discussed in section 2.3.2, one of the limitations of WireGuard is the protocol only handles layer three traffic. For this reason, the scope of the project encompassed the handling of IP packets, with non-IP traffic left as an optional beyond scope feature (1.2). Encapsulation of non-IP packets as IP was attempted but could not be delivered due to the lack of learning resources and developer knowledge in extending OVS for this purpose. Overall, compatibility issue did not jeopardise success but impacted the full capabilities of the final product.

Inadequacy and availability of software and hardware

The software and hardware components for implementing DPKM were acquired at its inception meaning this risk had no impact on the deliverable. However, testing was limited by the number of VMs the host PC could support and the lack of suitable benchmarking tools. New benchmarking tools or extensions to existing tools are needed to accurately evaluate the performance of DPKM (6.5).

Delay from poor planning and organisation of schedule

Although the project faced significant delay, it is the author's opinion that this was not the result of poor planning/organisation but rather the fruition of several previous risks. Bi-weekly meetings were held with the supervisor (*appx* B), and the schedule was frequently evaluated and updated based on the progress.

Inefficient progress from unsuitable SDLC

No major issues were caused by the iterative and incremental SDLC and the phases helped guide development.

Loss of data

Only a small amount of data from one file was lost, utilising four GitHub repositories to backup the full source code, and cloud storage facilities to store multiple copies of supporting text documentation and resources.

Hardware failure

Environmental issues like hardware failure were not encountered and no equipment needed replacement.

6.3 Results Discussion

As described in section 5.5, the DPKM system underwent eight different tests. The primary aim was to measure the average time (milliseconds) needed to complete various procedures in a network comprising of one controller, two nodes, and four hosts. Table 6.2 presents the results obtained from calculating the time difference between the request and response packets in Wireshark. Cycles refers to the total number of iterations performed for each test. Average completion time is separated into two columns for nodes A and B, and then used to calculate the combined average. The final column documents any issues encountered by the controller during a test. Overall, the results show promise for the performance of DPKM since all operations could be completed in under half a second with minimal issues.

Test	**Cycles**	**Avg Time A (ms)**	**Avg Time B (ms)**	**Combined Avg (ms)**	**Issues**
Configure	400	4.61	5.98	5.29	No
Unconfigure	40	0.46	0.52	0.49	Disconn, Early Resp
Add Peer (S)	800	2.55	3.32	2.94	No
Add Peer (B)	400	72.78	72.78	72.78	No
Delete Peer (S)	800	17.78	20.77	19.28	No
Delete Peer (B)	400	100.65	100.65	100.65	No
Full Setup	20	376.7	376.7	376.7	No
Full Teardown	20	178.39	178.39	178.39	See Unconf

Table 6.2: Results from the eight tests performed with one controller and two nodes, measuring the average time to complete various DPKM procedures in milliseconds.

The tested system has proven to be efficient and makes changing the WireGuard setup feel instantaneous in the controller. It was discovered that node A performed marginally better than B in all tests, with a total difference of 5.19ms. Comparing the single message request tests (1,2,3,5), shows that `dpkm_delete_key` messages (2) return a status response the quickest, and `dpkm_delete_peer` messages (5) return the slowest (figure 6.1). However, the unconfigure test data has questionable viability for comparison due to the inherent issues of the *Unconfigure WireGuard* procedure (5.4.2). In the node a status response is sent before executing the unconfigure sequence. This means the time only includes the request/response whereas the other

tests account for the changes made to WireGuard. Unfortunately, this meant obtaining an accurate result was not possible due to the disconnect issue caused by rebuilding the WireGuard interface. If measured correctly, the unconfigure test could prove to be the slowest of the four.

One surprising outcome is that deleting a peer performed worse than configuring WireGuard, with an average difference of 13.99ms. It was predicted that configuration would be the slowest operation due to the intensive process of generating and binding new 32-byte WireGuard keys. The tester was particularly satisfied with the `dpkm_add_peer` response, requiring only 2.94ms for a node to append new peer information to WireGuard and return a `dpkm_status` message. It is speculated that this would be the fastest procedure in the DPKM system if unconfiguring WireGuard was measured accurately.

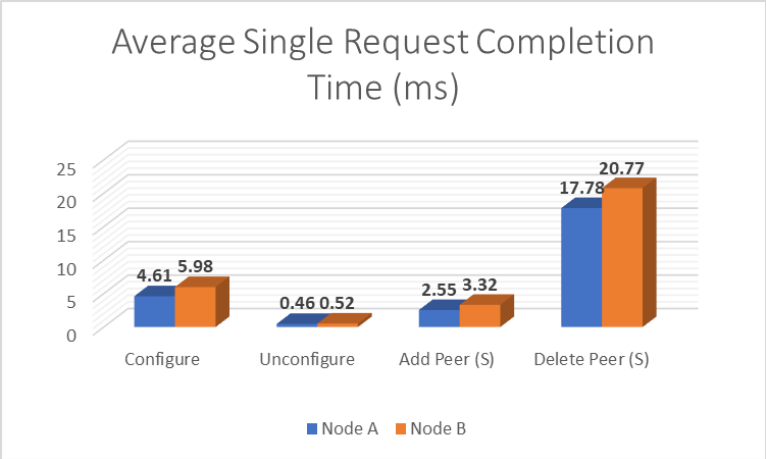

Figure 6.1: Bar Chart plotting the individual average completion time (ms) for the four single request/response tests with nodes A and B.

Performing tests with multiple requests (4,6,7,8) enables speculation over the performance of full procedures and controller operations (figure 6.2). Examining the results against the single request tests shows that most of the computation occurs within the controller. The add peer procedure required a total of 72.78ms. Removing the total time of two single message sequences (5.88ms) shows that 66.9ms was spent in the controller. Applying the same logic to the delete peer procedure results in a slightly lower time of 62.09ms. One limitation of the testing methodology is the time calculations do not include the subsequent controller operations (after the second status response). It can be estimated that the real time of the full add peer and delete peer procedures is approximately 139.68ms and 167.55ms respectively.

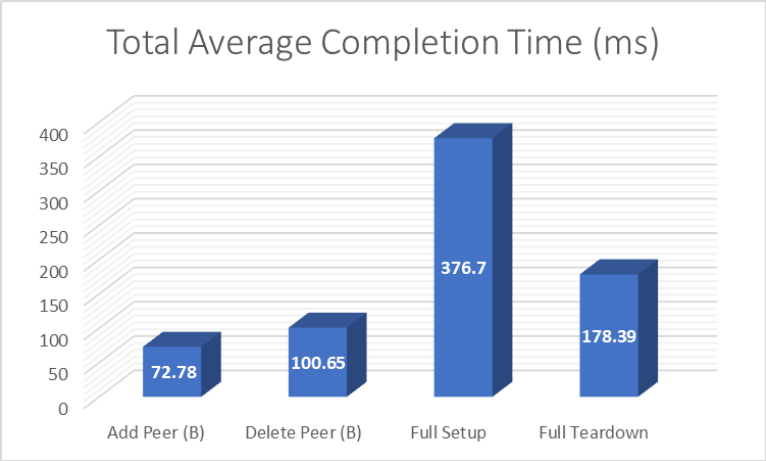

Figure 6.2: Bar Chart plotting the combined average completion time (ms) for the four multi-part request/response tests with nodes A and B.

Measuring the full setup involved performing the *Configure WireGuard* procedure twice (5.4.1), followed by the *Add Peer* procedure (5.4.3), and finally the *Start Communication* procedure (5.4.5). The full setup required a total of 376.7ms. Using the previous measurements, it can be estimated that the controller spends 226.44ms processing the configure WireGuard responses and preparing to start communication. This value was determined by removing the add peer procedure (139.68ms) and two set key requests/responses (10.58ms) from the total.

Measuring the full teardown was done by performing the *End Communication* procedure (5.4.6), followed by the *Delete Peer* procedure (5.4.4), and finally the *Unconfigure WireGuard* procedure twice (5.4.2). The full teardown required a total of 178.39ms but suffered from the inaccuracies of the delete key data. Using previous measurements, it can be estimated that the controller only spends 9.86ms preparing to end communication. This value was obtained by removing the delete peer procedure (167.55ms) and two delete key requests/response (0.98ms) from the total time.

Overall, it is no surprise that the controller occupies most procedural time since the database must be accessed and multiple conditionals/loops are used in the code. It would be beneficial to repeat the same experiments after making improvements to reduce message size and streamline larger methods. In future, more thorough testing is needed to encompass other parameters such as latency and throughput, analyse performance with a larger network, and evaluate DPKM against alternative solutions (6.6).

6.4 Achievements

This thesis aimed to deliver an SDN key management scheme named DPKM that facilitates data plane encryption through automated configuration of WireGuard tunnels. The primary motivations were achieved, successfully enhancing the security and capabilities of SDN, providing an improved proof of concept of Braadland's thesis (2017) [Bra17], and enabling dynamic establishment of encrypted flows. Development was a challenging but rewarding process that greatly expanded the developer's skill set and led to the production of a well-formed software solution. This section summarises the overall achievements from both a research and practical standpoint. Firstly, enhancements to the field of study will be identified in relation to the related work. Secondly, success will be measured by evaluating the full body of work against the proposed specification and design criteria. Lastly, several noteworthy deliverables will be highlighted to outline specific achievements. To provide an unbiased evaluation the limitations and lacking areas will be contrasted to the achievements in the succeeding section (6.5).

6.4.1 Enhancements to Field of Study

The preliminary background research made it evident that automated node-to-node encryption in OpenFlow-based SDN has been conceptualised but never delivered (2.4). Birkeland (2016) investigated and tested manual approaches to IPsec-based data plane encryption in SDN, expressing the need for easier configuration mechanisms and native support in software components [Bir16]. This thesis fulfilled the proposed future directions by extending Floodlight, OpenFlow, and OVS to support centralized management of WireGuard-based encryption channels. Braadland (2017) designed a key management scheme called DPKMS and formulated extensions to the OpenFlow protocol, requiring a proof of concept to determine the practical potential [Bra17]. This thesis transformed the concept of DPKMS into reality and redesigned most aspects to address flaws and misconceptions. The following enhancements are identified when comparing the content of this thesis to DPKMS [Bra17]:

- DPKMS generates private keys in the controller and transports them over OpenFlow, potentially compromising key integrity. This vulnerability is not present in DPKM since all keys are generated in the node and only public keys are transported.
- DPKMS makes changes to the WireGuard interface through new actions. This is not fea-

sible in OpenFlow since actions make changes to packets not interfaces. DPKM utilises multiple scripts to successfully support interactions between OVS and WireGuard.

- Messages in DPKMS rely on TLV structures for storing additional information. This is a feature of multipart messages which are difficult to extend and typically used for statistics. DPKM uses single-part OpenFlow messages to reduce message size and minimise complexity.
- DPKM successfully sets the flow conditions for encrypted communication, whereas DPKMS would be unable to facilitate encrypted communication if implemented to the exact specification.
- Rekeying in DPKMS relies on a non-existent queue action to pause packet flow. DPKM uses the practical approach of temporarily rerouting packets.
- DPKMS does not consider how the controller handles errors. The DPKM system includes full error handling with an ErrorLog database table.
- Most of the OpenFlow extensions proposed for DPKMS would not translate to a practical product, whereas this thesis delivers working examples.
- Wherever possible DPKM improves performance by reducing the number of request/response message sequences in its procedures.
- This thesis designs and implements an application UI with REST support. Moreover, practical extensions to the Floodlight and OVS codebases are documented.
- DPKM implements a compromise detection simulator and has undergone testing of the system.

Evaluating DPKM against the P4-IPsec system proposed by Hauser et al. (2020) is challenging due to the differences in technology choices [Hau+20]. Development on P4-IPsec and DPKM was undertaken in parallel and produced similar enhancements to the SDN field. DPKM enables dynamic WireGuard-based node-to-node encryption for OpenFlow switches whereas P4-IPsec delivers dynamic IPsec encryption for P4 switches. The modern technology choices used by Hauser et al. (2020) are an advantage over DPKM and give the impression that P4-IPsec will have a bigger overall impact on the scientific field [Hau+20]. Nevertheless, this thesis has helped progress different areas by addressing the lack of documentation surrounding OpenFlow extensions, Floodlight, and OVS. In future, comparative testing and cryptanalysis is needed to evaluate the two schemes under the same experimental conditions (6.6).

6.4.2 Measuring Success

Project success can be measured by analysing the thesis and implemented solution against the objectives, specification, and design. This thesis achieved the ten objectives set in section 1.3, producing a publication that reviews and enhances the state of the art, contains a feature rich set of designs and deliverables, and critically evaluates the functional system. Despite the scale, complexity, and lack of documentation jeopardising success (6.2), all designed procedures were implemented with limited missing components. Most requirements were fully satisfied (6.1), reinforcing the correctness and completeness of the software solution. Extending Floodlight has allowed the controller to administer the full key management life cycle (2.2.2) and perform all its expected duties (4.1.1). Extending OpenFlow has introduced new objects to the protocol and added key management capabilities to southbound communication (5.2). Extending OVS has enabled the switch to support new DPKM functionality and interact with the integrated WireGuard interface (5.3.2). Extending Floodlight web-ui has transformed the standard UI into a tool for initiating DPKM procedures and facilitating encryption (5.3.3). Overall, the author feels that the project has been a success and is satisfied with the work produced, aside from several limitations that will be addressed in future development.

6.4.3 Noteworthy Deliverables

- Extensions to four codebases – Floodlight, Floodlight-webui, Loxigen, Open vSwitch, with approximately 7500 lines of code spread across four GitHub repositories (5.1).
- Three virtual machines supporting a functional SDN capable of key management and dynamic node-to-node encryption with DPKM.
- New MySQL database tracking the system state and topology, with three database tables ConfiguredPeers, ConnectedPeers, and ErrorLog (4.1.1).
- Support for WireGuard in OVS and novel usage as a tunnelling protocol in SDN.
- Six new OpenFlow messages, one new OXM condition, and fifteen new error messages (5.2).
- New documentation for extending OpenFlow, Floodlight, OVS, and UI.
- Results from eight different performance tests proving the speed of the procedures.
- Full setup instructions for replicating the development environment and testing or continuing work on DPKM (*appx* B).

6.5 Limitations

Providing a well-rounded, unbiased evaluation involves presentation of both the strengths and weaknesses of a project. Although the delivered DPKM system met most of its requirements and objectives there are still areas that could be improved. This section hopes to provide transparency by outlining the known limitations of DPKM. Moreover, this section serves as a guide for improving DPKM and identifying future directions for development, testing, and analysis (6.6).

Empty Bytes Increase Message Size

Difficulties were encountered whilst trying to transport keys and ipv4 addresses in add peer, delete peer, and status OpenFlow experimenter messages (5.2.2). After failing to use standard integer fields, a working solution was achieved by utilising the char array description fields `desc_str` and `serial_num`. Although this method allows the procedures to operate correctly, these fields store up to 256-bytes and 32-bytes of statistical information respectively. Packet analysis with Wireshark shows that the key is 46 bytes and the ipv4 addresses are 12 bytes (switch) and 9 bytes (WireGuard). This means a total of 253 empty bytes are included in add peer messages, 210 bytes in delete peer messages, and 273 bytes in status messages. This is an inefficient use of space, making DPKM message extensions appear unprofessional. On the other hand, the negative effect on performance is likely to be marginal and the extra space means the scheme can accommodate increases in WireGuard key size. Nevertheless, a cleaner, more standardized approach needs to be developed for DPKM to have a place in industry.

Missing State Extraction Message

In the design of DPKM a `get_status` controller-to-switch message was planned that extracts the state of the node by triggering a status response (4.3). This message was defined in Loxigen, assigned a subtype (4) and respective error code, but never implemented or used. The developer was unwilling to spend the limited time available on integrating a message that's non-essential to core functionality and unused by the procedures. Implementing `get_status` requires new methods for the UI or Floodlight to initiate and send the message, and OVS functionality for extracting information from the WireGuard interface. Retrieving the status is beneficial to the synchronicity of the system and will be implemented when time constraints are no longer an issue.

Unconfiguring WireGuard throws Error

The *Unconfigure WireGuard* procedure (5.4.2) must remove keys by shutting down OVS, removing and recreating a blank WireGuard interface, and restarting the node. This is due to WireGuard not including functionality for removing keys or resetting the interface. Moreover, failing to temporarily shut down OVS causes the node to freeze and full deletion of WireGuard prevents future procedures being possible. A status response must be sent before running the unconfigure script, meaning for a short period the controller may think the procedure succeeded when execution failed. This is slightly remedied by returning an error after the status, but the solution is still unideal.

Furthermore, this approach causes an error to be thrown in the controller because the switch unexpectedly disconnects and reconnects. The developer suspects the error is caused by Floodlight continuing to send link layer discovery packets (LLDP) when the switch is shutdown or unprepared but could not find a solution. On one hand, this error appears to have no detrimental effect to running processes or future operations and the procedure fulfils its intended purpose. On the other hand, any errors demerit DPKM and could prevent the system from meeting the standards expected of a marketable product.

Encrypts Layer Three Traffic Only

WireGuard is a purely layer three protocol meaning only IP packets can be encrypted and sent between the nodes (2.3.2). Using WireGuard has restricted the capabilities of DPKM since OVS receives both IP and non-IP traffic. In its current state the *Start Communication* procedure (5.4.5) sets up flows for routing IP packets through WireGuard, sending all other packets unencrypted via the controller. Despite being beyond the scope of this thesis (1.2), mechanisms for handling non-IP packets were attempted but could not be achieved. DPKM has not reached its full potential, requiring additional OVS extensions for encapsulation of non-IP packets as IP. Such extensions were too complex for the developer to formulate within the project timeline and will be considered in future developments.

Lack of Handling for Crashes/Restarts

DPKM failed to satisfy the functional requirement of handling crashes or system restarts (6.1.2). This means if a communicating node crashes its peers will continue to send packets over WireGuard, causing packet loss. The system state is partially maintained by Floodlight and WireGuard after restarts due to the data stored in the controller and node. However, the

previous flow conditions are lost and must be reinstated. Floodlight provides fault tolerance features meaning DPKM can be configured to use backup controller instances during crashes. Nevertheless, additional mechanisms are needed to reroute any traffic directed to crashed nodes and automatically restore the system state on restart.

Built and Tested with Two Nodes

The scope of this thesis encompassed the construction of a basic SDN with one centralized controller and two nodes (1.2). For DPKM to be a marketable product it needs to be proven to function with a multitude of nodes since real networks are expected to support many. The underlying functionality has been implemented with the intention of supporting any number of nodes and shows no obvious signs of incompatibility. However, without detailed code reviews and extensive testing DPKM cannot be accredited for industry use. Moreover, it may be discovered that the performance of DPKM rapidly deteriorates as the network scale increases. Additional experiments and analyses are needed to enhance this research and take DPKM from the scientific field to industry.

Inherited Security Concerns

This thesis uses the technologies of Floodlight web-ui, Floodlight, OpenFlow, OVS, and WireGuard meaning any existing security issues are inherited by DPKM. One concern is the lack of access control and authentication when connecting to the UI. Currently the UI can be accessed through the IP address and port running Floodlight, allowing anyone with the right details to take full control of the network. Strict login and authentication mechanisms are needed before DPKM can be deployed on a publicly accessible network. A similar concern is the centralized nature of the Floodlight controller. Centralized SDN has received criticism throughout the scientific community due to the controller being a potential single point of failure. Much like the UI, compromise of the controller grants an attacker power to cause irreversible damage to the entire network. This can be overcome by adopting distributed architectures with multiple active controllers. In future, DPKM or a close variation should be developed for distributed SDN.

Plaintext Storage

Despite the overall security of DPKM achieving most requirements (6.1) there is still room for improvement. Floodlight accesses the MySQL database through the credentials stored in the `db.properties` file (5.1). The VM hosting Floodlight has password protection and the file

has restricted access permissions but the credentials are stored in plaintext format. Additional encryption mechanisms are needed to convert the credentials to ciphertext. This situation applies to the storage and protection of private keys in the node. Currently OVS generates private keys to a file which has restricted access permissions but contains the plaintext 32-byte key. Encryption would protect the private key even if the file contents were compromised.

Outdated Technology Choices

The SDN field is progressing at a rapid pace meaning every year technologies are replaced by the next innovative and promising solution. Reviewing the state of the art makes it evident that Floodlight peaked around 2016, and has gradually declined in popularity as more powerful, feature rich alternatives emerge (2.3.4). Focus has also shifted from OpenFlow-based switch technologies like OVS to more programmable data plane solutions such as P4 [Bos+14]. It could be argued that due to its supported technologies DPKM has no place as a key management solution in 2020. However, the developer stands by the technology choices (2.3) and feels the project would not have been possible without them. Floodlight provided good documentation and a strong basis for building DPKM. OpenFlow supported the flexibility for extensions and is still considered the standard southbound protocol. OVS is the most widely used and popular OpenFlow switch. Moreover, WireGuard only emerged in 2017, is becoming an increasingly popular tunnelling solution, and has not been used previously in SDN.

Overall, the achievements of DPKM far outweigh the limitations and give credit to the technology choices. DPKM as a standalone product may not receive industry usage but the documented approach and delivered code can now be applied to more complex prolific technologies such as the ONOS controller.

6.6 Future Work

The content of this thesis has created many new avenues for a future researcher or team of researchers to explore. Although a strong contribution to the scientific field has been made, the scale and complexity of development has meant the DPKM system is far from complete. This section outlines the work needed to enhance the research and extend the software solution. The reader should be able to use this section, and the set-up instructions in appendix B, as a guide for future testing and development.

6.6.1 Testing and Analysis

For DPKM to be accurately evaluated and accredited for industry use; extensive testing and analysis is needed. The DPKM system has been built and analysed for a single controller with two connected nodes. As described in section 6.5, this is a limitation of the product since SDN networks are usually expected to support a multitude of nodes. Enough unfulfilled research directions exist to produce an entire publication on analysing the security and performance of DPKM. However, the field is lacking in tools for benchmarking the system due to the novelty of the software extensions. Customisations to the Mininet [Min20] network virtualization tool are needed to simulate larger numbers of DPKM OVS switches and support experimentations with the new OpenFlow messages (5.2.2). New benchmarking tools or extensions to existing tools such as CBench [CBe20] and iPerf3 [iPe20], should also be explored to accurately measure the performance of DPKM.

The following experiments, metrics, and analyses could be conducted to evaluate DPKM:

- Measuring latency of the procedures and setup/teardown times for different numbers of nodes. For example, benchmarking the delay between Floodlight sending a `set_key` message to configure a node and receiving a status response, with increasing numbers of nodes – 2, 4, 8, 12, 16, and more.
- Benchmarking throughput of encrypted node-to-node communication, maximum traffic capacity, and packet loss.
- Evaluating performance of DPKM against alternatives such as P4-IPsec [Hau+20], using consistent metrics and testing under identical experimental conditions.
- Conducting a thorough security analysis of both individual software components and the entire system. This could be done conceptually or by using penetration testing and ethical hacking techniques to probe for vulnerabilities. Results can be used to contrast technology choices to alternatives and make subsequent improvements.
- In depth code review to find internal issues or inefficient code.
- Automated tests such as unit tests and integration tests.

6.6.2 Continuing Development

This thesis purposefully confined DPKM to a scope (1.2) and set of objectives (1.3) that were achievable by a single developer. Moreover, the limitations have highlighted areas that can be improved through additional work. These factors have resulted in a multitude of future directions which will be discussed within the context of the SDN planes.

Data Plane

Possible improvements to the data plane involve modification of WireGuard and OVS to better integrate the two technologies. In its current state, DPKM is constrained by WireGuard to only handling layer three, IP traffic. New mechanisms should be developed to enable OVS to encapsulate non-IP traffic as IP before passing it to WireGuard. Outstanding issues remain in the *Unconfigure WireGuard* procedure (5.4.2) that cause an error to be thrown in the controller. A solution may be discovered through alternative approaches to the unconfigure script, modifications to the WireGuard software, or introduction of better native support to OVS.

Interactions between OVS and WireGuard are done through the execution of bash script or commands. Improvements can be made by developing kernel modules or OVS daemons that integrate the inner node functionality in a cleaner, containerized fashion. The process of setting up a DPKM node involves manually adjusting the IP addresses in the script (5.1) and making several tweaks to the system environment. A more user-friendly approach could be formulated with a new command-line tool or data plane GUI.

OpenFlow

The OpenFlow messages proposed in this thesis use the unconventional method of description fields to store key and ipv4 addresses. It would be beneficial to achieve a working solution that adheres to the OpenFlow standards and reduces message size. Moreover, the `get_status` message was defined in OpenFlow but never used (4.3). Modifications to the UI, Floodlight and OVS should be made to support the message and extract the state of the node. Additional OpenFlow extensions for new match, action, and instruction objects could also be formulated to increase the granularity of packet matching and encrypted flows in DPKM.

Application Plane

The primary focus of this thesis has been on key management in the data plane meaning a basic UI was created (5.3.3). Further styling and attention to aesthetics would polish DPKM

into a professional product. For the UI to be used on a publicly accessible network login and access control mechanisms need to be developed. This would require extension of the controller database or an additional database for storing credentials, design of login forms, and a permission or role system for administrators. Another beneficial UI extension would be inclusion of an error review/logging tool connected to the ErrorLog database. Such a tool would provide stronger monitoring of the system state and quicker resolution of issues. This can be realised by adding a new REST API to Floodlight.

Control Plane

The bulk of all DPKM functionality and logic is implemented in the Floodlight controller. Floodlight suffers from the security risks associated with centralized SDN and has significantly diminished in popularity. Translation of DPKM and the associated code to a popular, distributed controller like ONOS would strengthen its position in the SDN field. ONOS is built in java and based on Floodlight [Ber+14], meaning much of the produced work is applicable. This is an exciting prospect that would make DPKM valuable to industry. Moreover, this would allow branching into P4 and other modern technologies supported by ONOS.

Finally, several additional extensions could be made to extend overall security and data protection. These include but are not limited to – implementing fault tolerance for crashes and restarts, encrypting stored information, intrusion/compromise detection systems, and integrated use of digital certificates.

Chapter 7

Conclusion

This thesis documented the development of an automated key management scheme called DPKM that facilitates WireGuard-based data plane encryption in SDN. Manual establishment of node-to-node encryption defeats the dynamic nature of SDN, meaning an automated configuration process is a requisite to delivering flexible network management. Surveying the state of the art showed that key management in OpenFlow-based SDN had been conceptualised, but never implemented. This thesis was motivated to produce a proof of concept of Braadland's scheme DPKMS [Bra17] but required a complete redesign of the proposed extensions. The delivered DPKM system has maintained several core concepts from DPKMS but is a vastly novel product in the research field.

Development required approximately 7500 lines of code and extension of four codebases – Floodlight, Floodlight-webui, Loxigen, and Open vSwitch. Three virtual machines were used to build a functional SDN with one controller and two nodes. The OpenFlow protocol was extended by using the experimenter structure to define six new messages, one flow matching condition, and fifteen error messages. These extensions were integrated into the Floodlight controller by generating a new OpenFlow library with the Loxigen tool. A DPKM module was added to Floodlight with an OpenFlow message handler, REST support, MySQL database, and error listener. OVS software was modified to support the key management extensions, return status or error messages, and adjust the WireGuard setup. A new UI window was included to support the initiation of key management procedures through REST connections to the controller, and display information about the nodes, active tunnels, and system state.

Critical evaluation showed that the thesis achieved all its objectives and DPKM satisfied most requirements. The scale, complexity, and lack of documentation was challenging for a single developer, but the produced work is of a high standard and exceeded expectations. DPKM successfully administers the key management lifecycle and supports dynamic establishment of WireGuard tunnels for encrypted IP communication. DPKM is far from industry ready due to several limitations and needs additional work to handle all communication, fix outstanding issues, and streamline design aspects. Extensive testing and analysis with a multitude of nodes is necessary to accurately evaluate performance and address undetected issues. The contributions of this thesis have made further development a possibility for any researcher interested in extending or analysing DPKM. This thesis has progressed the field of study and is a stepping-stone toward integrating key management in SDN. The author hopes to continue development and bring DPKM into industry as a marketable product.

Bibliography

[AB15] Adel Alshamrani and Abdullah Bahattab. "A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model". In: *International Journal of Computer Science Issues (IJCSI)* 12.1 (2015), p. 106.

[AL99] Carlisle Adams and Steve Lloyd. *Understanding public-key infrastructure: concepts, standards, and deployment considerations*. Sams Publishing, 1999.

[Ark+08] D Arkhipkin et al. "An SSH key management system: easing the pain of managing key/user/account associations". In: *Journal of Physics: Conference Series*. Vol. 119. 7. IOP Publishing. 2008, p. 072005.

[AS10] Gaurav Agarwal and Saurabh Singh. "A Comparison Between Public Key Authority and Certification Authority for Distribution of Public Key". In: *International Journal of Computer Science and Information Technologies* 1 (Jan. 2010).

[Ato20] Atom. *Atom text editor Home Page*. 2020. URL: `https://atom.io` (visited on 11/01/2020).

[Aum+13] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.

[AY17] Hakan Akcay and Derya Yiltas-Kaplan. "Web-Based User Interface for the Floodlight SDN Controller". In: *International Journal of Advanced Networking and Applications* 8.05 (2017), pp. 3175–3180.

[Azo13] Siamak Azodolmolky. *Software defined networking with OpenFlow / Siamak Azodolmolky.* eng. 2013. ISBN: 1-84969-873-2.

[Bar+01] Daniel J Barrett et al. *SSH, the Secure Shell: the definitive guide*. " O'Reilly Media, Inc.", 2001.

[Bar+09] Elaine Barker et al. "Recommendation for key management part 3: Application-specific key management guidance". In: *NIST special publication* 800 (2009), p. 57.

[Bar+13] Elaine Barker et al. "NIST Special Publication 800-130: A framework for designing cryptographic key management systems". In: *National Institute of Standards and Technology Report* (2013).

[Bar20] Elaine Barker. "NIST Special Publication 800-57: Recommendation for Key Management: Part 1 - General". In: *NIST special publication* (2020). DOI: 10.6028/NIST.SP.800-57pt1r5.

[Ber+14] Pankaj Berde et al. "ONOS: towards an open, distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 1–6.

[Ber06] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography - PKC 2006*. Ed. by Moti Yung et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9.

[BIK02] Matt Blaze, John Ioannidis, and Angelos D Keromytis. "Trust management for IPsec". In: (2002).

[Bir16] Steffen Fredrik Birkeland. "Software Defined Data Flow Isolation by Virtualization and Cryptographic Key Distribution". MA thesis. NTNU, 2016.

[BM12] S Balaji and M Sundararajan Murugaiyan. "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC". In: *International Journal of Information Technology and Business Management* 2.1 (2012), pp. 26–30.

[Bos+14] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[BR12] Elaine Barker and Allen Roginsky. *Recommendation for cryptographic key generation*. US Department of Commerce, National Institute of Standards and Technology, 2012.

[Bra17] Anna Selvåg Braadland. "Key Management for Data Plane Encryption in SDN Using WireGuard". In: 2017.

[BS03] Kurt Bittner and Ian Spence. *Use case modeling*. Addison-Wesley Professional, 2003.

[CBe20] CBench. *CBench GitHub Repository*. 2020. URL: https://github.com/trema/cbench (visited on 12/01/2020).

[Cha+18] Paul Chaignon et al. "Oko: Extending Open VSwitch with Stateful Filters". In: SOSR '18. Los Angeles, CA, USA: Association for Computing Machinery, 2018. ISBN: 9781450356640. DOI: 10.1145/3185467.3185496. URL: https://doi.org/10.1145/3185467.3185496.

[Cho+19] Ankur Chowdhary et al. "TRUFL: Distributed Trust Management framework in SDN". In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–6.

[Cho02] W. Chou. "Inside SSL: the secure sockets layer protocol". In: *IT Professional* 4.4 (2002), pp. 47–52. DOI: 10.1109/MITP.2002.1046644.

[CK15] Eric F Crist and Jan Just Keijser. *Mastering OpenVPN*. Packt Publishing Ltd, 2015.

[Cre11] Cas Cremers. "Key Exchange in IPsec Revisited: Formal Analysis of IKEv1 and IKEv2". In: *Computer Security – ESORICS 2011*. Ed. by Vijay Atluri and Claudia Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 315–334. ISBN: 978-3-642-23822-2.

[Dar+17] T. Dargahi et al. "A Survey on the Security of Stateful SDN Data Planes". In: *IEEE Communications Surveys Tutorials* 19.3 (2017), pp. 1701–1725. DOI: 10.1109/COMST.2017.2689819.

[DAr01] Paolo D'Arco. "On the Distribution of a Key Distribution Center". In: *Theoretical Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 357–369. ISBN: 978-3-540-45446-5.

[Dav01] Carlton R. Davis. *Ipsec: Securing Vpns*. McGraw-Hill Professional, 2001. ISBN: 0072127570.

[DBC88] A. M. Davis, E. H. Bersoff, and E. R. Comer. "A strategy for comparing alternative software development life cycle models". In: *IEEE Transactions on Software Engineering* 14.10 (1988), pp. 1453–1461. DOI: 10.1109/32.6190.

[DH76] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[Di +99] Giovanni Di Crescenzo et al. "How To Forget a Secret". In: *STACS 99*. Ed. by Christoph Meinel and Sophie Tison. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 500–509.

[Doc20a] Floodlight Documentation. *net.floodlightcontroller.core Interface IOFMessageListener*. 2020. URL: http://floodlight.github.io/floodlight/javadoc/floodlight/net/floodlightcontroller/core/IOFMessageListener.html (visited on 11/02/2020).

[Doc20b] Floodlight Documentation. *net.floodlightcontroller.core Interface IOFSwitch*. 2020. URL: http://floodlight.github.io/floodlight/javadoc/floodlight/net/floodlightcontroller/core/IOFSwitch.html (visited on 11/02/2020).

[Don17] Jason A Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *NDSS*. 2017.

[DP18] Benjamin Dowling and Kenneth G. Paterson. "A Cryptographic Analysis of the WireGuard Protocol". In: *Applied Cryptography and Network Security*. Ed. by Bart Preneel and Frederik Vercauteren. Cham: Springer International Publishing, 2018, pp. 3–21. ISBN: 978-3-319-93387-0.

[Elk+02] O. Elkeelany et al. "Performance analysis of IPSec protocol: encryption and authentication". In: *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No.02CH37333)*. Vol. 2. 2002, 1164–1168 vol.2. DOI: 10.1109/ICC.2002.997033.

[Fei06] Markus Feilner. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd, 2006.

[Fel+18] Dennis Felsch et al. "The dangers of key reuse: practical attacks on IPsec IKE". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 567–583.

[FKK11] Alan Freier, Philip Karlton, and Paul Kocher. "The secure sockets layer (SSL) protocol version 3.0". In: *IETF* 3 (2011), pp. 1–67.

[Flo20] Floodlight. *Floodlight SDN Controller Home Page*. 2020. URL: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview (visited on 10/30/2020).

[Fou12] Open Networking Foundation. "Software-defined networking: The new norm for networks". In: *ONF White Paper* 2.2-6 (2012), p. 11.

[Fou20] Eclipse Foundation. *Eclipse Home Page*. 2020. URL: https://www.eclipse.org (visited on 11/01/2020).

[FS99] Niels Ferguson and Bruce Schneier. *A cryptographic evaluation of IPsec*. 1999.

[Fum98] Walter Fumy. "Key Management Techniques". In: *State of the Art in Applied Cryptography: Course on Computer Security and Industrial Cryptography Leuven, Belgium, June 3–6, 1997 Revised Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 142–162. ISBN: 978-3-540-49248-1. DOI: 10.1007/3-540-49248-8_6. URL: https://doi.org/10.1007/3-540-49248-8_6.

[Gen+99] Rosario Gennaro et al. *Two-phase cryptographic key recovery system*. US Patent 5,937,066. Aug. 1999.

[Gir20] Damien Giry. *Keylength-NIST report on cryptographic key length and cryptoperiod (2020)*. 2020. URL: URL%20https://www.%20keylength.%20com/en/4/ (visited on 10/20/2020).

[Git20] GitHub. *GitHub Home Page*. 2020. URL: https://github.com (visited on 11/01/2020).

[Gor14] Paul Goransson. *Software defined networks : a comprehensive approach / Paul Goransson, Chuck Black*. eng. 2014. ISBN: 0-12-416684-9.

[Gos17] Bhargavi Goswami. "Revolution in Existing Network under the Influence of Software Defined Networks (SDN)". In: Mar. 2017.

[HAP18] Yotam Harchol, Ittai Abraham, and Benny Pinkas. "Distributed SSH Key Management with Proactive RSA Threshold Signatures". In: *Applied Cryptography and Network Security*. Ed. by Bart Preneel and Frederik Vercauteren. Cham: Springer International Publishing, 2018, pp. 22–43. ISBN: 978-3-319-93387-0.

[Hau+20] F. Hauser et al. "P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN". In: *IEEE Access* 8 (2020), pp. 139567–139586. DOI: 10.1109/ACCESS.2020.3012738.

[Hey00] Rus Heywood. *UML use case diagrams: Tips and FAQ*. 2000. URL: https://www.andrew.cmu.edu/course/90-754/umlucdfaq.html (visited on 08/01/2020).

[Hol20a] Arne Holst. *Amount of traffic from software-defined networks (SDNs) and network function virtualization (NFV) in data centers worldwide, from 2015 to 2021*. 2020. URL: https://www.statista.com/statistics/637966/worldwide-sdn-nfv-data-center-traffic/ (visited on 04/30/2020).

[Hol20b] Arne Holst. *Software-defined networking (SDN) market size worldwide from 2013 to 2021*. 2020. URL: https://www.statista.com/statistics/468636/global-sdn-market-size/ (visited on 04/30/2020).

[Hu+15] Z. Hu et al. "A comprehensive security architecture for SDN". In: *2015 18th International Conference on Intelligence in Next Generation Networks*. 2015, pp. 30–37. DOI: 10.1109/ICIN.2015.7073803.

[Ins00] Project Management Institute. *A guide to the project management body of knowledge (PMBOK guide).* Vol. 2. Project Management Inst, 2000.

[iPe20] iPerf3. *iPerf3 Home Page*. 2020. URL: https://iperf.fr (visited on 12/01/2020).

[Iza16] Ryan Izard. *Floodlight Web GUI*. 2016. URL: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/40403023/Web+GUI (visited on 05/01/2020).

[Iza18] Ryan Izard. *How to Use OpenFlow Queues*. 2018. URL: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/15040526/How+to+Use+OpenFlow+Queues (visited on 10/30/2020).

[Jac+16] Ethan J Jackson et al. "Softflow: A middlebox architecture for open vswitch". In: *2016 USENIX Annual Technical Conference*. 2016, pp. 15–28.

[Jai+13] Sushant Jain et al. "B4: Experience with a Globally-Deployed Software Defined Wan". In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), pp. 3–14. ISSN: 0146-4833. DOI: 10.1145/2534169.2486019. URL: https://doi.org/10.1145/2534169.2486019.

[Jar+14] M. Jarschel et al. "Interfaces, attributes, and use cases: A compass for SDN". In: *IEEE Communications Magazine* 52.6 (2014), pp. 210–217.

[Jus05] Mike Just. "Key agreement". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 325–325. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_218. URL: https://doi.org/10.1007/0-387-23483-7_218.

[Kau+05] Charlie Kaufman et al. *Internet key exchange (IKEv2) protocol*. Tech. rep. RFC 4306, December, 2005.

[KF13] H. Kim and N. Feamster. "Improving network management with software defined networking". In: *IEEE Communications Magazine* 51.2 (2013), pp. 114–119.

[KKS13] R. Klöti, V. Kotronis, and P. Smith. "OpenFlow: A security analysis". In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 2013, pp. 1–6. DOI: 10.1109/ICNP.2013.6733671.

[Kre+15] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

[KRT11] I. Kotuliak, P. Rybár, and P. Trúchly. "Performance comparison of IPsec and TLS based VPN technologies". In: *2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2011, pp. 217–221. DOI: 10.1109/ICETA.2011.6112567.

[Lan+16] Adam Langley et al. "ChaCha20-Poly1305 cipher suites for transport layer security (TLS)". In: *RFC 7905* 10 (2016).

[Li+16] J. Li et al. "Rekeying for Encrypted Deduplication Storage". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016, pp. 618–629. DOI: 10.1109/DSN.2016.62.

[Lip18] Benjamin Lipp. *A Mechanised Computational Analysis of the WireGuard Virtual Private Network Protocol*. 2018.

[LKR14] A. Lara, A. Kolasani, and B. Ramamurthy. "Network Innovation using OpenFlow: A Survey". In: *IEEE Communications Surveys Tutorials* 16.1 (2014), pp. 493–512. DOI: 10.1109/SURV.2013.081313.00105.

[LLP17] R Lopez, G Lopez-Millan, and F Pereniguez-Garcia. "Software-defined networking (sdn)-based ipsec flow protection". In: *Internet Engineering Task Force, Internet-Draftdraft-ietf-i2nsf-sdn-ipsec-flow-protection-00*. 2017.

[Lox20] Floodlight LoxiGen. *Floodlight Loxigen Github Repository*. 2020. URL: https://github.com/floodlight/loxigen (visited on 10/30/2020).

[LR17] T. Lins and R. A. Rabelo Oliveira. "Energy efficiency in industry 4.0 using SDN". In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. 2017, pp. 609–614. DOI: 10.1109/INDIN.2017.8104841.

[LT17] D. Lacković and M. Tomić. "Performance analysis of virtualized VPN endpoints". In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2017, pp. 466–471. DOI: 10.23919/MIPRO.2017.7973470.

[Mac+20] Steven Mackey et al. "A Performance Comparison of WireGuard and OpenVPN". In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. CODASPY '20. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 162–164. ISBN: 9781450371070. DOI: 10.1145/3374664.3379532. URL: https://doi.org/10.1145/3374664.3379532.

[Mau92] Ueli M Maurer. "A universal statistical test for random bit generators". In: *Journal of cryptology* 5.2 (1992), pp. 89–105.

[MCC17] Y. Ma, Y. Chen, and J. Chen. "SDN-enabled network virtualization for industry 4.0 based on IoTs and cloud computing". In: *2017 19th International Conference on Advanced Communication Technology (ICACT)*. 2017, pp. 199–202. DOI: 10.23919/ICACT.2017.7890083.

[McK+08] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[MD13] Apoorva Mishra and Deepty Dubey. "A comparative study of different software development life cycle models in different scenarios". In: *International Journal of Advance research in computer science and management studies* 1.5 (2013).

[Mea99] Catherine Meadows. "Analysis of the Internet Key Exchange protocol using the NRL protocol analyzer". In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*. IEEE. 1999, pp. 216–231.

[Med+14] J. Medved et al. "OpenDaylight: Towards a Model-Driven SDN Controller architecture". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. 2014, pp. 1–6. DOI: 10.1109/WoWMoM.2014.6918985.

[MI88] Tsutomu Matsumoto and Hideki Imai. "On the KEY PREDISTRIBUTION SYSTEM: A Practical Solution to the Key Distribution Problem". In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 185–193. ISBN: 978-3-540-48184-3.

[Mil+88] S. P. Miller et al. "Kerberos Authentication and Authorization System". In: *In Project Athena Technical Plan*. 1988.

[Min20] Mininet. *Mininet Home Page*. 2020. URL: http://mininet.org (visited on 12/01/2020).

[MKK11] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. "Revisiting traffic anomaly detection using software defined networking". In: *International workshop on recent advances in intrusion detection*. Springer. 2011, pp. 161–180.

[MLD18] L. Mamushiane, A. Lysko, and S. Dlamini. "A comparative evaluation of the performance of popular SDN controllers". In: *2018 Wireless Days (WD)*. 2018, pp. 54–59. DOI: 10.1109/WD.2018.8361694.

[MM13] Tal Mizrahi and Yoram Moses. "Time-based updates in OpenFlow: A proposed extension to the OpenFlow protocol". In: *CCIT Report# 835* 1792 (2013).

[MMR15] L. V. Morales, A. F. Murillo, and S. J. Rueda. "Extending the Floodlight Controller". In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. 2015, pp. 126–133. DOI: 10.1109/NCA.2015.11.

[Mut+19] Ammar Muthanna et al. "Secure and Reliable IoT Networks Using Fog Computing with Software-Defined Networking and Blockchain". In: *Journal of Sensor and Actuator Networks* 8.1 (Feb. 2019), p. 15. ISSN: 2224-2708. DOI: 10.3390/jsan8010015. URL: http://dx.doi.org/10.3390/jsan8010015.

[NHA17] Musa Ndiaye, Gerhard P Hancke, and Adnan M Abu-Mahfouz. "Software defined networking for improved wireless sensor network management: A survey". In: *Sensors* 17.5 (2017), p. 1031.

[OCF15] F. Olivier, G. Carlos, and N. Florent. "SDN Based Architecture for Clustered WSN". In: *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. 2015, pp. 342–347.

[ONF13] Open Networking Foundation (ONF). "OpenFlow Switch Specification Version 1.4.0 (Wire Protocol 0x05)". In: (2013).

[ONF20a] ONF. *Open Networking Foundation*. 2020. URL: https://www.opennetworking.org (visited on 04/30/2020).

[ONF20b] ONF. *Open Networking Foundation - ONOS*. 2020. URL: https://opennetworking.org/onos/ (visited on 05/01/2020).

[Opp16] Rolf Oppliger. *SSL and TLS: Theory and Practice*. Artech House, 2016.

[Opp98] R. Oppliger. "Security at the Internet layer". In: *Computer* 31.9 (1998), pp. 43–47. DOI: 10.1109/2.708449.

[OVS20] OVS. *What is Open vSwitch?* 2020. URL: https://docs.openvswitch.org/en/latest/intro/what-is-ovs/ (visited on 10/30/2020).

[Pan+17] B. Pandya et al. "Framework for securing SDN southbound communication". In: *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*. 2017, pp. 1–5. DOI: 10.1109/ICIIECS.2017.8275912.

[Par20] Visual Paradigm. *UML Class Diagram Tutorial*. 2020. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/ (visited on 11/02/2020).

[PD13] Ben Pfaff and Bruce Davie. "The open vswitch database management protocol". In: *Internet Requests for Comments, RFC Editor, RFC* 7047 (2013).

[Per16] Trevor Perrin. “The noise protocol framework”. In: *PowerPoint Presentation* (2016).

[Pfa+15] Ben Pfaff et al. “The design and implementation of open vswitch”. In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 117–130.

[PK00] R. Perlman and C. Kaufman. “Key exchange in IPSec: analysis of IKE”. In: *IEEE Internet Computing* 4.6 (2000), pp. 50–56. DOI: 10.1109/4236.895016.

[PM18] Miteshkumar Shaileshbhai Parmar and Arvind D Meniya. “Imperatives and Issues of IPSEC Based VPN”. In: *International Journal of Science and Modern Engineering (IJISME)* 1.2 (2018), pp. 38–41.

[Por+15] Phillip A Porras et al. “Securing the software defined network control layer.” In: *NDSS*. 2015.

[Pri20] Dan Price. *WireGuard vs. OpenVPN: Which One Should You Use?* 2020. URL: https://www.makeuseof.com/tag/wireguard-vs-openvpn/ (visited on 10/28/2020).

[PS17] Frederic Pohl and Hans Dieter Schotten. “Secure and Scalable Remote Access Tunnels for the IIoT: An Assessment of openVPN and IPsec Performance”. In: *Service-Oriented and Cloud Computing*. Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen. Cham: Springer International Publishing, 2017, pp. 83–90. ISBN: 978-3-319-67262-5.

[Qin+14] Z. Qin et al. “A Software Defined Networking architecture for the Internet-of-Things”. In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. 2014, pp. 1–9.

[Ran17] Sahil Barjtya Ankur Sharma Usha Rani. “A detailed study of Software Development Life Cycle (SDLC) models”. In: *International Journal Of Engineering And Computer Science* 6.7 (2017).

[Ree00] George Reese. *Database Programming with JDBC and JAVA*. " O’Reilly Media, Inc.", 2000.

[RM20] Research and Markets. *Software-Defined Networking Market by Component (SDN Infrastructure, Software, and Services), SDN Type (Open SDN, SDN via Overlay, and SDN via API), End User, Organization Size, Enterprise Vertical, and Region - Global Forecast to 2025*. 2020. URL: https://www.researchandmarkets.com/reports/5136178/software-defined-networking-market-by-component (visited on 10/30/2020).

[Rup10] Nayan B. Ruparelia. "Software Development Lifecycle Models". In: 35.3 (May 2010), pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814. URL: https://doi.org/10.1145/1764810.1764814.

[Sak+06] Shoichi Sakane et al. "Kerberized internet negotiation of keys (KINK)". In: *RFC4430, March* (2006).

[SDG18] N. A. Snow, V. R. Dasari, and B. E. Geerhart. "OpenFlow Experimenter Labels for Encoding Adaptive Network Functions". In: *2018 IEEE 39th Sarnoff Symposium*. 2018, pp. 1–5. DOI: 10.1109/SARNOF.2018.8720450.

[Sez+13] S. Sezer et al. "Are we ready for SDN? Implementation challenges for software-defined networks". In: *IEEE Communications Magazine* 51.7 (2013), pp. 36–43.

[Sha+20] Arash Shaghaghi et al. "Software-Defined Network (SDN) Data Plane Security: Issues, Solutions, and Future Directions". In: *Handbook of Computer Networks and Cyber Security: Principles and Paradigms*. Ed. by Brij B. Gupta et al. Cham: Springer International Publishing, 2020, pp. 341–387. ISBN: 978-3-030-22277-2. DOI: 10.1007/978-3-030-22277-2_14. URL: https://doi.org/10.1007/978-3-030-22277-2_14.

[SM14] N. A. B. Sulaiman and H. Masuda. "Evaluation of a Secure Live Migration of Virtual Machines Using Ipsec Implementation". In: *2014 IIAI 3rd International Conference on Advanced Applied Informatics*. 2014, pp. 687–693. DOI: 10.1109/IIAI-AAI.2014.142.

[SMG13] Marian Stoica, Marinela Mircea, and Bogdan Ghilic-Micu. "Software Development: Agile vs. Traditional." In: *Informatica Economica* 17.4 (2013).

[SNK12] M. Shin, K. Nam, and H. Kim. "Software-defined networking (SDN): A reference architecture and open APIs". In: *2012 International Conference on ICT Convergence (ICTC)*. 2012, pp. 360–361.

[Spe13] OpenFlow Switch Specification. "Version 1.3. 2 (Wire Protocol 0x04)". In: *Open Networking Foundation* (2013).

[Tim20] Microsoft Office Timeline. *Microsoft Office Timeline Home Page*. 2020. URL: https://www.officetimeline.com (visited on 05/05/2020).

[TJ01] Frank A Tycksen Jr and Charles W Jennings. *Digital Certificate*. US Patent 6,189,097. Feb. 2001.

[TLG15] N. B. Truong, G. M. Lee, and Y. Ghamri-Doudane. "Software defined networking-based vehicular Adhoc Network with Fog Computing". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015, pp. 1202–1207.

[Tom13] FUJITA Tomonori. "Introduction to ryu sdn framework". In: *Open Networking Summit* (2013), pp. 1–14.

[Tos+14] Umar Toseef et al. "C-BAS: Certificate-based AAA for SDN experimental facilities". In: *2014 Third European Workshop on Software Defined Networks*. IEEE. 2014, pp. 91–96.

[TSP17] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. "Building an Extensible Open VSwitch Datapath". In: *SIGOPS Oper. Syst. Rev.* 51.1 (Sept. 2017), pp. 72–77. ISSN: 0163-5980. DOI: 10.1145/3139645.3139657. URL: https://doi.org/10.1145/3139645.3139657.

[Tut20] TutorialsPoint. *SDLC - Iterative Incremental Model*. 2020. URL: https://www.tutorialspoint.com/adaptive_software_development/sdlc_iterative_incremental_model.htm (visited on 05/03/2020).

[TV16] Poonam Tijare and Deepika Vasudevan. "The Northbound APIs of Software Defined Networks". In: (Oct. 2016). DOI: 10.5281/zenodo.160891.

[Vir20] VirtualBox. *VirtualBox Home Page*. 2020. URL: https://www.virtualbox.org (visited on 11/01/2020).

[Vod20] Vodafone. *SDN Research Report 2019*. 2020. URL: https://www.vodafone.com/business/news-and-insights/white-paper/SDN-research-report-2019 (visited on 10/30/2020).

[Wei01] Joel Weise. "Public key infrastructure overview". In: *Sun BluePrints OnLine, August* (2001), pp. 1–27.

[Wir20] Wireshark. *Wireshark Home Page*. 2020. URL: https://www.wireshark.org (visited on 11/01/2020).

[YL+06] Tatu Ylonen, Chris Lonvick, et al. *The secure shell (SSH) protocol architecture*. 2006.

[You+17] F. Z. Yousaf et al. "NFV and SDN—Key Technology Enablers for 5G Networks". In: *IEEE Journal on Selected Areas in Communications* 35.11 (2017), pp. 2468–2478. DOI: 10.1109/JSAC.2017.2760418.

[Zbi+98] H Zbinden et al. "Quantum cryptography." In: *Applied Physics B: Lasers & Optics* 67.6 (1998).

[ZR00] Pedro Zubeldia and Gordon W Romney. *Method and apparatus for managing key revocation*. US Patent 6,044,462. Mar. 2000.

[Zur+17] Piotr Zuraniewski et al. "Facilitating ICN Deployment with an Extended Openflow Protocol". In: *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ICN '17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 123–133. ISBN: 9781450351225. DOI: 10.1145/3125719.3125729. URL: https://doi.org/10.1145/3125719.3125729.

Appendix A

Record of Supervision

Student Name: Luke Hengstenberg.
Student Number: 878876.
Dissertation Title: *"DPKM: A Key Management Scheme Facilitating WireGuard-based Data Plane Encryption in SDN"*.
Supervisor: Dr Pardeep Kumar.

Full record of the meeting minutes and content is provided in tables A.1 and A.2.

Statement of originality

I certify that this dissertation is my own work and that where the work of others has been used in support of arguments or discussion, full and appropriate acknowledgement has been made. I am aware of and understand the University's regulations on plagiarism and unfair practice.

Signed .....

Date .....14/12/2020.....

Supervision	Date, Duration	Notes	Initials Supervisor	Initials Student
1: Brief outline of research question and prelimary title.	13/03/2020, 1 hour	Discussed findings from the background research into SDN security and possible directions for the project. Identified the potential approach of implementing a proof of concept of another masters thesis.	P.K	L.H
2: Discussion of detailed plan and bibliography.	04/05/2020, 1 hour	Question and answer session regarding the project specification and design document. Went through the plan for implementation and how best to structure the document.	P.K	L.H
3: Progress update.	26/06/2020, 45 mins	Setup basic SDN system and started writing extensions for OpenFlow and Loxigen. Concern over lack of documentation and discussed plan B scenarios.	P.K	L.H
4: Progress update.	03/07/2020, 45 mins	Finished a number of OpenFlow extensions but feels they are incorrect. Attempting extensions to OVS/Floodlight but failing to make progress due to documentation. Struggling to understand how to integrate the new OpenFlow objects. Expressed several external factors that negatively impacted mental health/project. Looked at finding additional resources, alternative approaches and extending the deadline due to no progress.	P.K	L.H
5: Progress update.	10/07/2020, 30 mins	Stepped away from work to refresh knowledge on basic SDN concepts before continuing. Contacted a professional in SDN industry for advice via zoom call. Submitted extension application. Adjusted progress timeline.	P.K	L.H
6: Progress update.	31/07/2020, 30 mins	Attempting to send and process set_key message but receiving bad_experimenter errors. Decision made to focus on implementing a simple test request/response message sequence before attempting more complex extensions.	P.K	L.H

Table A.1: Meeting minutes part A taken as record of supervision.

Supervision	Date, Duration	Notes	Initials Supervisor	Initials Student
7: Progress update.	21/08/2020, 30 mins	Successfully implemented test request/response messages in Floodlight/OpenFlow/OVS. Wrote script for setup/teardown. Still concerned over lack of beneficial progress.	P.K	L.H
8: Progress update.	28/08/2020, 1 hour	Changed design of set key message to have no payload. Successfully implemented configure wireguard procedure with status response. Implemented add peer and delete peer procedures but some issues. Created MySQL database. Gave a brief demo, capturing packets with Wireshark.	P.K	L.H
9: Progress update.	25/09/2020, 1 hour	Implemented start communication procedure and new OXM but cannot encapsulate non-IP packets as IP. Lots of debugging and discussion over how to route other traffic as layer 3. Implemented rekeying procedure. Had to modify all other procedures due to synchronisation issues from keys changing.	P.K	L.H
10: Progress update.	16/10/2020, 30 mins	Implemented all remaining procedures - end communication, compromise and revocation. Added Error handling. Attempting to write unit tests but having difficulty. Dissertation becomes focus from next week.	P.K	L.H
11: Progress update.	30/10/2020, 30 mins	Discussed dissertation layout and progress so far. Student finally in a good position and happy with the product.	P.K	L.H
12: Progress update.	13/11/2020, 15 mins	Brief meeting regarding dissertation. Should be on track to finish by deadline.	P.K	L.H
13: Progress update.	27/11/2020, 30 mins	Question and answer session regarding best way to submit code and the chapters left to write.	P.K	L.H
14: Submission.	11/12/2020, 30 mins	Final meeting discussing submission, the future of DPKM, the project as a whole, and general reflection on the work produced.	P.K	L.H

Table A.2: Meeting minutes part B taken as record of supervision.

Appendix B

Setup instructions for DPKM system

B.1 Prerequisites

Multiple Ubuntu 18.04 virtual machines setup as “Bridged Adapters” or another method to host machines with unique IP addresses. In DPKM, VirtualBox was used to create three VM’s – one for the controller, two for the nodes. Follow the guide in the footnote for help setting up Ubuntu VM’s. [1].

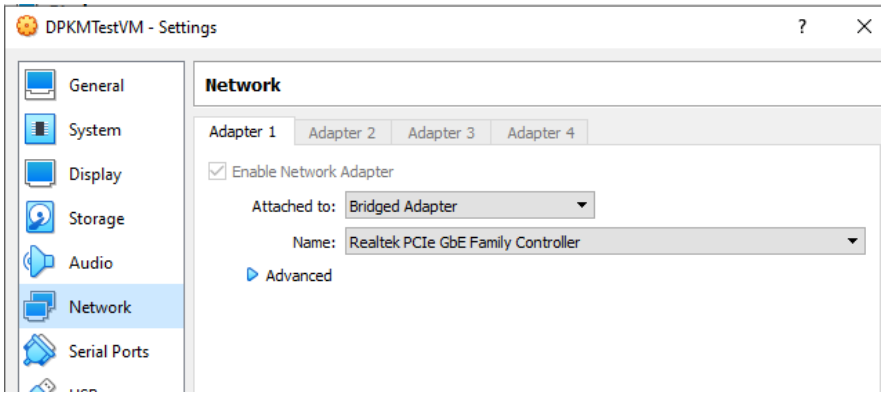

Figure B.1: Example of Bridged Adapter setting in VirtualBox.

[1] `https://brb.nci.nih.gov/seqtools/installUbuntu.html`

B.2 DPKM Floodlight Controller/UI Setup

1. Install Java Development Kit 8 and change current jdk version to 8. Follow guide in the footnote for help installing java/changing version[2]. The java version can be changed with the command: `update-alternatives --config java`.
2. Install the Floodlight dependencies by running the command:
   `apt-get install git build-essential ant maven python-dev`.
3. CD into the desired directory and download the source code from github:
   `git clone git://github.com/lukehengstenberg/DPKM_Floodlight.git`.
4. From this point onwards the downloaded `DPKM_Floodlight` folder will be referred to as `floodlight`. Rename the folder for convenience.
5. Before Floodlight can be used the `cntrldb` database needs to be created. Install MySQL server with the command:
   `apt-get install mysql-server`.
6. Check MySQL is running with `sudo service mysql status`. If down start with `service mysql start`.
7. Login to MySQL with `mysql -u root -p`, entering root password when prompted.
8. Run the command `CREATE DATABASE cntrldb;` to recreate the database.
9. Create the floodlight user with the command:
   `CREATE USER 'floodlight'@'localhost' IDENTIFIED BY 'FLPass878876#';`.
   If new credentials are to be used, modify the following file to store the new url, user, and password `floodlight/src/main/resources/db.properties`.
10. Select the database with `USE cntrldb;` and grant database access permissions to floodlight with `GRANT ALL ON cntrldb.* TO 'floodlight'@'localhost';`.
11. Use `exit` to return to command line. CD into floodlight and run the command `mysql -u floodlight -p cntrldb < cntrldb.sql` to create the three tables CommunicatingPeers, ConfiguredPeers, and ErrorLog.
12. Log back into mysql as floodlight with `mysql -u floodlight -p`, run `USE cntrldb;`, followed by `SHOW TABLES;` to confirm the three tables have been created. Use `DESCRIBE` to see table schema.

[2] `https://docs.datastax.com/en/jdk-install/doc/jdk-install/installOpenJdkDeb.html`

13. CD back into floodlight and run `git submodule init`, `git submodule update`, `ant` to build the floodlight.jar file and automatically pull the UI from `git://github.com/lukehengstenberg/DPKM_Floodlight-webui`.
14. If an error is thrown at this point remove the `/floodlight/src/main/resource/web` directory and run `git submodule sync` followed by `git submodule update --init`.
15. Make a new local directory with `mkdir /var/lib/floodlight` and set access permissions with `chmod 777 /var/lib/floodlight`.
16. Running the command `java -jar target/floodlight.jar` can now launch the controller. **HOWEVER**, in its current prototype state this method means the DPKM UI cannot reach the cntrldb database, throwing an error. To use the UI for DPKM procedures the project **MUST** be setup and run from eclipse.
17. Download the eclipse IDE with `snap install --classic eclipse`.
18. Follow the instructions on the official floodlight website[a] to create the eclipse project setup files, import floodlight as an existing project into a new workspace, and run the controller.

```
$ ant eclipse
```

This creates several files: Floodlight.launch, Floodlight_junit.launch, .classpath, and .project. From these you can setup a new Eclipse project.

- Open eclipse and create a new workspace
- File -> Import -> General -> Existing Projects into Workspace. Then click "Next".
- From "Select root directory" click "Browse". Select the parent directory where you placed floodlight earlier.
- Check the box for "Floodlight". No other Projects should be present and none should be selected.
- Click Finish.

Running Floodlight in Eclipse

Create the FloodlightLaunch target:

- Click **Run->Run Configurations**
- Right Click **Java Application->New**
- For **Name** use **FloodlightLaunch**
- For **Project** use **Floodlight**
- For **Main** use **net.floodlightcontroller.core.Main**
- Click **Apply**

To then run Floodlight click on the drop-down arrow next to the Play button and select the proper target to run. These also work with debug targets.

[a] `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343544/Installation+Guide#InstallationGuide-EclipseIDE`

19. **Troubleshooting:** The first attempt at building the project in eclipse may produce the error "The project was not built due to X being marked read-only". This can be solved by changing the workspace folder to have write access permissions or selecting another build path.
20. Now that the controller is running the GUI should be deployed. Use a web browser

to navigate to the URL `http://<controller-ip>:8080/ui/pages/index.html`. The DPKM page can then be reached through the navbar or `/dpkm.html`.

21. The Floodlight/UI setup is now complete! The dpkm source code can be viewed and extended by navigating to the `src/main/java` folder and exploring the `net.floodlightcontroller.dpkmconfigurewg` package. Extending the UI will require the repository to be cloned into a separate folder and opened in a text editor/IDE suited to web development.

B.3 DPKM OVS Node Setup

1. Login to the VM hosting the DPKM node and clone the git repo with `git clone https://github.com/lukehengstenberg/DPKM_OVS.git`. Rename the directory to `ovs` for convenience.
2. Add the file names `ConfigureWG.sh`, `UnconfigureWG.sh`, `SetupTestNetworkWG.sh`, `dpkm_mn_custom.py` to the file `distfiles`. This will stop a compiler error when initially building OVS. In future these files need to be explicitly included in the distribution.
3. Modify the contents of `SetupTestNetworkWG.sh` to set the desired WireGuard IP address (line 22), change the switch IP address to the VM IP (line 27), and the controller IP address to match the Floodlight VM setup previously (line 31). Make sure to add the WireGuard IP to `UnconfigureWG.sh`.
4. Move `SetupTestNetworkWG.sh` to preferred location and set as executable with `chmod 111 SetupTestNetworkWG.sh`.
5. Move `ConfigureWG.sh` and `UnconfigureWG.sh` to the root directory with `cp ConfigureWG.sh /`. As a root user change the file permissions to executable with `chmod 111 ConfigureWG.sh`.
6. Install the full list of requirements on the official OVS website. [3]
7. Python 3 can be installed from the link in the footnote [4], and running the following commands should satisfy all dependencies. Get any missing packages based on compiler errors.

[3] `https://docs.openvswitch.org/en/latest/intro/install/general/#build-requirements`

[4] `https://phoenixnap.com/kb/how-to-install-python-3-ubuntu`

```
sudo apt-get install -y build-essential
sudo apt-get install -y fakeroot
sudo apt-get install -y debhelper
sudo apt-get install -y autoconf
sudo apt-get install -y automake
sudo apt-get install -y libssl-dev
sudo apt-get install -y bzip2
sudo apt-get install -y openssl
sudo apt-get install -y graphviz
sudo apt-get install -y python-all
sudo apt-get install -y procps
sudo apt-get install -y python-qt4
sudo apt-get install -y python-zopeinterface
sudo apt-get install -y python-twisted-conch
sudo apt-get install -y libtool
sudo apt-get install -y dh-autoreconf
sudo apt-get install -y python3-sphinx
sudo apt-get install -y libelf-dev
```

Listing B.1: Dependencies needed to build and use Open vSwitch.

8. In the ovs directory run `./boot.sh` followed by `./configure` to setup the OVS package. Consult the different configure options on the OVS website if extending development. It is recommended to enable compiler errors with `--enable-Werror` to assist with debugging.
9. Follow the building[5] and starting[6] instructions on the OVS website to build and start the daemons/scripts.
10. Install WireGuard by running `sudo apt update`, followed by
    `sudo apt install wireguard`.
    WireGuard is supported out of the box on Ubuntu 20.4 but DPKM was built for Ubuntu 18.04.
11. Install mininet by running `sudo apt update`, followed by
    `sudo apt install mininet`.
    This is for testing the switch as a virtual topology with hosts.
12. The switch can either be launched directly with `./SetupTestNetworkWG.sh` or through mininet with `./dpkm_mn_custom.py`. The `SetupTestNetworkWG` script will create a new switch with an integrated WireGuard interface but no hosts. The `dpkm_mn_custom` script will create a custom mininet topology with one switch, two hosts but no WireGuard

[5] https://docs.openvswitch.org/en/latest/intro/install/general/#building
[6] https://docs.openvswitch.org/en/latest/intro/install/general/#starting

interface. WireGuard will have to be created and included separately by running the following commands in order:

`sudo ip link add dev wg0 type wireguard`,

`sudo ip addr add <IP>/24 dev wg0`,

`sudo ip link set wg0 up`,

`sudo ovs-vsctl add-port s1 wg0`.

13. Confirm by running `sudo ovs-vsctl` to display the switch, controller connection, and ports. Running `sudo wg` should display the blank interface with a random listening-port.
14. **WARNING:** The `dpkm_mn_custom.py` file needs to be modified to include the controller IP and the dpid/mac values may need to be changed to avoid conflict with other switches. In Floodlight the out port in the `constructFlowAdd` method of the `DpkmFlows.java` class may need to be modified to the correct port number of WireGuard. The code currently assumes WireGuard is port 1, which is only the case in the `SetupTestNetworkWG` launch approach. Run the command `sudo ovs-vsctl get Interface wg0 ofport` to check the port number.
15. The basic DPKM node setup is now complete! The source code can be modified by opening `ovs` as a project in a text editor such as Atom. Any changes must be built by running `sudo make` and `sudo make install`, followed by `./SetupTestNetworkWG.sh` or `./dpkm_mn_custom.py` to rebuild the network topology.

B.4 Capturing Packets in Wireshark

1. Install Wireshark on the Floodlight VM with

   `sudo add-apt-repository ppa:wireshark-dev/stable`,

   `sudo apt update`,

   `sudo apt -y install wireshark`.
2. CD into the floodlight folder and copy the Wireshark lua script into the correct location with

   `sudo cp openflow.lua /usr/lib/x86_64-linux-gnu/wireshark/plugins`.

   This script was generated by loxigen and will enable capturing and analysis of DPKM packets.
3. Run Wireshark from root or with `sudo wireshark`, select an interface, and type `of` into

the packet filter to capture OpenFlow packets.

4. Start the controller in eclipse and launch any nodes with `./SetupTestNetworkWG.sh` or `./dpkm_mn_custom.py` (and the commands to add WireGuard).
5. The GUI should now be accessible! Try initiating the configure WireGuard procedure for each node, then add the interfaces as peers to build connections and start communication. In the node, use `sudo wg` to observe the changes to the interface configuration and `sudo ovs-ofctl dump-flows s1` to view the flow.

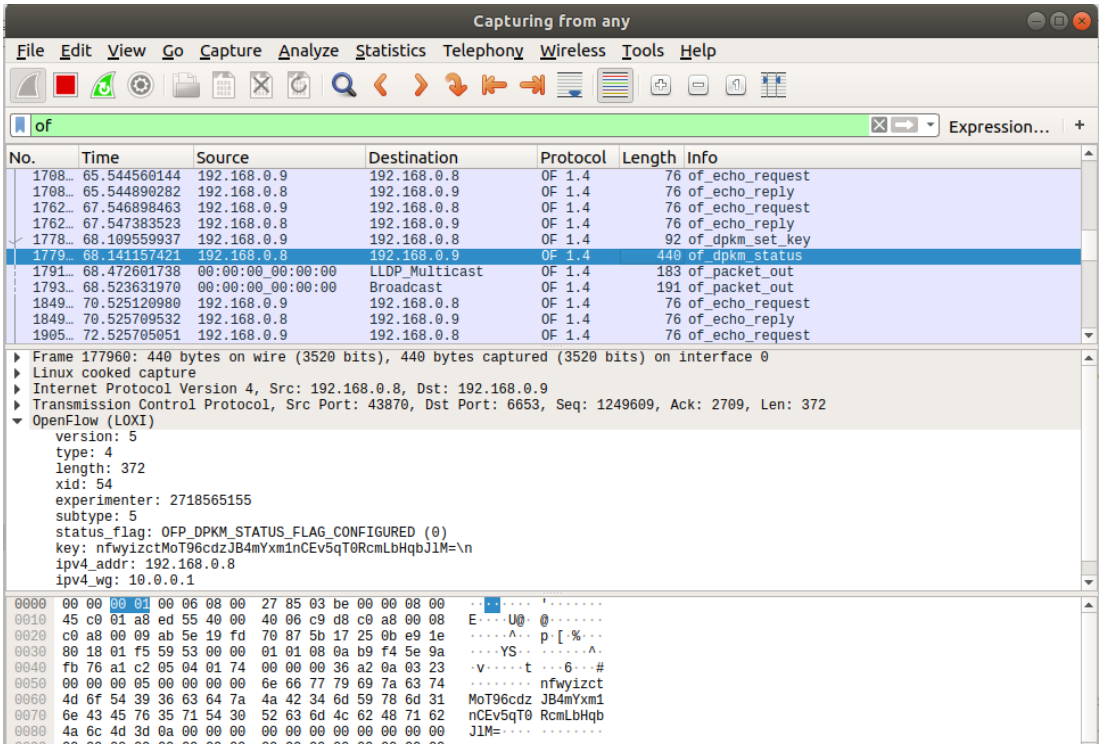

Figure B.2: Example of Wireshark setup, with OpenFlow filter and lua script for capturing DPKM packets.

Appendix C

Full Code Description

This appendix describes the classes and methods added to Floodlight, OVS, and Floodlight-webui. For brevity, only the main definitions and a short description of each is given, with the full code contained in the GitHub repositories. Loxigen was not included since only a smaller number of extensions were made, meaning most code was presented within the main body of this thesis (5.2). The Loxigen GitHub repository is not public due to it being a fork of the original instead of a clone. However, the generated .jar file can be accessed through the Floodlight repository[1], containing all the generated DPKM builders used by the controller. The input extension definitions will be bundled in the code submission.

C.1 Floodlight

This section provides a description of each class and method in the `net.floodlightcontroller.dpkmconfigurewg` Floodlight package. Visit or download the GitHub repository to see the source code[2].

C.1.1 ConnectionProvider.java

Provides a connection to the underlying DPKM database using the credentials stored in the db.properties file.[3]

[1] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/lib/openflowj-3.6.0-SNAPSHOT-sources.jar`

[2] `https://github.com/lukehengstenberg/DPKM_Floodlight`

[3] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/ConnectionProvider.java`

`public class ConnectionProvider`

`public static Connection getConn()`
@return Connection object from DriverManager or null.

`private static Properties getCredentials()`
@return Properties object with credentials stored in db.properties file.

C.1.2 Dpkm.java

Core internal functionality of the Data Plane Key Management protocol. Main utility is querying underlying db and sending messages to switch without direct interaction from administrator.[4]

`public class Dpkm`

`protected void constructAddPeerMessage(IOFSwitch sw, String ipv4Addr)`
Writes a `DPKM_ADD_PEER` message to the given switch sw for any configured switches in db. This triggers the switch to add the peer info to its WireGuard interface, returning a `DPKM_STATUS` or error response message. Used internally to automatically add configured switches as peers.

`protected void sendAddPeerMessageInternal(IOFSwitch sw, String peerPubKey, String peerIPv4, String peerIPv4WG)`
Builds and writes a `DPKM_ADD_PEER` message to switch sw with params. This triggers the switch to add the peer info to its WireGuard interface, returning a `DPKM_STATUS` or error response message. Used internally by sendAddPeerMessage and constructAddPeerMessage.

`protected void sendDeletePeerMessageInternal(IOFSwitch sw, String peerPubKey, String peerIPv4, String peerIPv4WG)`
Builds and writes a `DPKM_DELETE_PEER` message to switch sw with params. This triggers the switch to remove the peer info from its WireGuard interface, returning a `DPKM_STATUS` or error response message. Used internally by sendDeletePeerMessage and constructDeletePeerMessage.

`protected String getDpId(String ipv4Addr, int id, boolean isId)`
Returns the DatapathId of switch with address ipv4Addr or matching id from db or Error.

`protected String getIp(String dpid, boolean wg)`
Returns the Ipv4 Address or WireGuard address of switch sw from db or Error.

[4] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/Dpkm.java`

`protected int getCryptoperiod(String dpid)`
Returns the cryptoperiod of switch dpid from db or -1.

`protected DpkmSwitch statusToNode(IOFSwitch sw, OFDpkmStatus msg)`
Converts an IOFSwitch and OFDpkmStatus message into a DpkmSwitch object.

`protected DpkmPeers statusToPeer(IOFSwitch sw, OFDpkmStatus msg)`
Converts an IOFSwitch and OFDpkmStatus message into a DpkmPeers object.

`protected String checkIPExists(String ipv4Addr)`
Returns count of records with matching ipv4Addr in db.

`protected int checkConfigured()`
@return int Number of switches with status 'CONFIGURED' or -1 if error.

`public boolean checkCompromised(String dpid)`
Returns boolean compromised for the switch with dpid.

`public int checkConnected(String ipv4AddrA, String ipv4AddrB, String statusType)`
Returns count of peer connections using SQL queries based on statusType or -1 if error. Default: count of connections between ipv4AddrA and ipv4AddrB. Used internally for a number of conditional statements.

`protected int checkConnectedAny(String ipv4Addr)`
Returns count of any connections with ipv4Addr as a peer. Used internally for a number of conditional statements.

`public int checkError(String dpid)`
Returns unresolved error count for switch with DatapathId dpid. Used internally for a number of conditional statements.

`protected void writeSwitchToDB(DpkmSwitch node)`
Updates the database by inserting a new record with the information contained in the DpkmSwitch object. Used internally to add a record for a new WireGuard configured switch.

`protected void updateSwitchInDB(DpkmSwitch node)`
Updates the database by modifying an existing record with the information contained in the DpkmSwitch object. Used internally to update an existing record for a new WireGuard key.

`protected void updateSwitchCompromised(int id)`
Updates the database by modifying an existing record with the matching id to compromised = true.

Used internally to simulate compromise of a switch.

`protected void addPeerConnection(String ipv4Addr, String ipv4AddrPeer)`
Updates the database by inserting a new peer connection between ipv4Addr and ipv4AddrPeer. Used internally to add a record for a new WireGuard peer connection.

`protected void updatePeerInfo(String ipv4Addr, String ipv4AddrPeer, String statusChange)`
Updates the database by modifying an existing connection with matching IP addresses ipv4Addr and ipv4AddrPeer. Chooses an SQL Query based on integer statusChange. Default: set connection status as 'CONNECTED'. Used internally to update an existing peer connection to reflect state of switch.

`protected void removePeerConnection(String ipv4Addr, String ipv4Peer)`
Updates the database by removing the record matching the IP addresses ipv4Addr and ipv4Peer. Used internally to remove an existing peer connection.

`protected void removeAllPeerConnections(String dpid)`
Updates the database by removing any records matching the switch dpid.

`protected void removeSwitch(String ipv4Addr)`
Updates the database by removing the record matching the address ipv4Addr. Used internally to remove a WireGuard configured switch.

C.1.3 DpkmCompromiseNodeResource.java

REST API for compromising a switch to simulate the behaviour of some security system and begin the revocation procedure. Takes json data from UI and deserializes, executing corresponding function.[5]

`public class DpkmCompromiseNodeResource extends ServerResource`

`@Post public String compromise(String fmJson)`
Compromise the switch with matching info in the given fmJson. Calls compromiseNode with the id which ends all communication with the switch and sets it as "COMPROMISED".

[5] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmCompromiseNodeResource.java

C.1.4 DpkmConfigureWG.java

Facilitator and handler of Data Plane Key Management protocol communication between the controller and switch. Provides functionality to REST APIs used by the administrator. Adjusts WireGuard configuration by sending particular DPKM messages based on the status response received from the switch and desired logic of administrator.[6]

`public class DpkmConfigureWG extends DpkmFlows implements IFloodlightModule, IDpkmConfigureWGService, IOFMessageListener`

`public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx)`
Implements a message listener for messages of type experimenter. Executes appropriate functions based on received message subtype.

`private void processStatusMessage(IOFSwitch sw, OFDpkmStatus msg)`
Processes the status message sent by the switch based on the status flag. Calls one of several functions to process based on type of status response.

`private synchronized void processConfigured(IOFSwitch sw, DpkmSwitch node)`
Processes the `SET_KEY` status response sent by the switch. Uses the content of the node object to query the database and respond accordingly.

`private synchronized void processPeerAdded(IOFSwitch sw, DpkmPeers peer)`
Processes the `ADD_PEER` status response sent by the switch. Uses the content of the peers object to query the database and respond accordingly.

`private synchronized void processPeerRemoved(IOFSwitch sw, DpkmPeers peer)`
Processes the `DELETE_PEER` status response sent by the switch. Uses the content of the peers object to query the database and respond accordingly.

`private synchronized void processRevoked(IOFSwitch sw, DpkmSwitch node)`
Processes the `DELETE_KEY` status response sent by the switch. Uses the content of the node object to query the database and respond accordingly.

`public void sendSetKeyMessage(DatapathId dpid, int cryptoperiod)`
Writes a `DPKM_SET_KEY` message to the switch with the given dpid, and sets the cryptoperiod. This triggers the switch to generate the keys, configuring its WireGuard interface, returning a status or error response message.

[6] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmConfigureWG.java`

`public void sendDeleteKeyMessage(DatapathId dpid)`
Writes a `DPKM_DELETE_KEY` message to the switch with the given dpid. This triggers the switch to delete its keys, unconfiguring WireGuard, returning a status or error response message.

`public List<DpkmSwitch> getSwitches()`
Returns a full list of configured switches from the db, mapping fields to DpkmSwitch objects.

`public List<DpkmPeers> getPeers()`
Returns a full list of peers from the db, mapping fields to DpkmPeers objects.

`public void sendAddPeerMessage(String sourceDpid, String targetDpid)`
Writes a `DPKM_ADD_PEER` message to the switch with the given source dpid, adding the switch with the given target dpid as a peer. This triggers the switch to add the peer info to its WireGuard interface, returning a status or error response message.

`public void sendDeletePeerMessage(String sourceDpid, String targetDpid, boolean keyChange`
Writes a `DPKM_DELETE_PEER message` to the switch with the given source dpid, removing the switch with the given target dpid as a peer. This triggers the switch to remove the peer info from its WireGuard interface, returning a status or error response message.

`public void startCommunication(String dpidA, String dpidB)`
Writes a `FLOW_MOD` message to both peer switches to push traffic through WG interface in port in accordance to the created flow table entry. Enables packets to be encrypted/decrypted and sent between the interfaces.

`public void endCommunication(String dpidA, String dpidB, String endType)`
Writes a `FLOW_MOD` message to both peer switches to terminate communication entirely or continue communication unencrypted (not through WG). This is decided by the administrator using the top level UI to set endType.

`public synchronized void rekey(String dpid, int cryptoperiod)`
Rekey's the switch after expiry of the given cryptoperiod. This consists of reconfiguring the switch using a `DPKM_SET_KEY` message.

`public void compromiseNode(int id)`
Compromises switch with id to trigger the revocation procedure. This updates the db record and ends all communication with the switch.

`public void revoke(String dpid, String revType)`
Carries out the revocation procedure for a compromised switch dpid. Based on the top-level policy this means reconfiguring the switch with a `SET_KEY` message or terminating the switch with a `DELETE_KEY` message.

C.1.5 DpkmConfigureWGResource.java

REST APIs for getting all configured switches, adding/configuring a new switch, and deleting a configured switch. Takes json data from UI and deserializes, executing corresponding function.[7]

`public class DpkmConfigureWGResource extends ServerResource`

`@Get("json") public List<DpkmSwitch> retrieve()`
Returns a full list of configured switches from the db in json format.

`@Post public String configure(String fmJson)`
Configures WG interface in the given switch and sets the cryptoperiod. Deserializes to get dpid and cryptoperiod, sending `SET_KEY` message on success.

`@Delete public String delete(String fmJson)`
Deletes/unconfigures WG interface for switch matching the given id. Deserializes to get id, finds db record, sending `DELETE_KEY` message on success.

`public static DpkmSwitch jsonToDpkmSwitch(String fmJson)`
Converts switch information given in json format to a DpkmSwitch object. Maps each json value to a field in DpkmSwitch.

C.1.6 DpkmConfigureWGWebRoutable.java

Registers the DPKM REST resources with the controller and assigns paths that can be used to externally call the APIs. [8]

`public class DpkmConfigureWGWebRoutable implements RestletRoutable`

[7] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmConfigureWGResource.java

[8] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmConfigureWGWebRoutable.java

```
public Restlet getRestlet(Context context)
```

```
public String basePath()
```

C.1.7 DpkmEndCommunicationResource.java

REST API for ending communication between two connected peers. Takes json data from UI and deserializes, executing endCommunication to terminate all or WG only communication depending on the top level preference.[9]

```
public class DpkmEndCommunicationResource extends ServerResource
```

```
@Post public String end(String fmJson)
```

Ends communication for connection with matching peer information in json. Deserializes to get both peer switch information and end type. Either removes flow entirely or modifies flow to continue unencrypted.

C.1.8 DpkmError.java

Internal functionality for the Dpkm error handler. Used to interact with the database after an OpenFlow error is reported.[10]

```
public class DpkmError extends Dpkm
```

```
protected void logError(String dpid, String type, String errCode,
String note)
```

Updates the database by inserting a new record into table ErrorLog for an error that occurred at a switch dpid.

```
protected int checkErrorCount(String dpid, String errCode)
```

Counts the number of errors matching code errCode for switch dpid that have not been resolved.

```
protected int checkAttempt(String dpid, String errCode)
```

Gets the number of attempts switch dpid has had at sending a message causing error code errCode. Used to implement a limit to the number of times the controller can resend a request.

[9] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmEndCommunicationResource.java

[10] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmError.java

`protected void updateAttempt(String dpid, String errCode)`
Updates number of attempts for error errCode reported by switch dpid. Increments Attempt by 1, used to record the number of retries.

`protected void updateMaxAttempt(String dpid, String errCode, String note)`
Called when the max number of attempts has been reached, updating the note of a record to provide information to the administrator.

`protected void resolveError(String dpid, String type)`
Called when an error has been resolved and some DPKM message is working as expected. Updates resolved to true for all records of matching type.

`protected String extractIp(OFDpkmBaseError inError)`
Extracts the ipv4 address from an error message as a substring of the toString(). Used to recreate the same add/delete peer message and retry.

`protected String getIpFromKey(String key)`
Gets the IP address of a switch matching a specific public key. Used in circumstances where an error is caused by a missing IP address.

C.1.9 DpkmErrorHandler.java

Module responsible for handling all errors related to the Data Plane Key Management protocol. Version 1.0 of DPKM has a total of 15 error messages which are handled here. When an error is sent by a switch in response to a DPKM message this class logs the error and resends the same message several times in case the error occurred due to some unique circumstance. If the maximum number of retries is reached this class disconnects the switch until the error has been addressed externally by the administrator.[11]

`public class DpkmErrorHandler extends DpkmError implements IFloodlightModule, IOFMessageListener`

`public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx)`
Implements a message listener for messages of type EXPERIMENTER and ERROR. Only interested in `DPKM_STATUS` (subtype 5) experimenter messages as an indicator that the message is now returning as expected. Listens for all DPKM error messages and passes on to the processing method.

[11] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmErrorHandler.java`

`private void processResponseMessage(String dpid, OFDpkmStatus msg)`
Processes the status message sent by the switch based on the status flag. Resolves errors based on type of status response indicating that the message is returning as expected.

`private synchronized void processErrorMessage(IOFSwitch sw, OFDpkmBaseError inError)`
Processes the error messages sent by the switch based on the subtype. Has a total of 15 cases for the 15 DPKM error types. Calls a handler method based on the type of error and desired strategy.

`private synchronized void handleErrorSetKey(IOFSwitch sw, String errCode, String note)`
Handles errors caused by a `DPKM_SET_KEY` message. Resends the message twice so the controller has three attempts. If all three attempts fail mark the error as `ACTION_NEEDED`. The switch is now blocked from performing DPKM related functions until administrator marks the error as resolved.

`private synchronized void handleErrorDeleteKey(IOFSwitch sw, String errCode, String note)`
Handles errors caused by a `DPKM_DELETE_KEY` message. Resends the message twice so the controller has three attempts. If all three attempts fail mark the error as `ACTION_NEEDED`. The switch is now blocked from performing DPKM related functions until administrator marks the error as resolved.

`private synchronized void handleErrorAddPeer(IOFSwitch sw, String errCode, String note, String ipv4Peer)`
Handles errors caused by a `DPKM_ADD_PEER` message. Resends the message twice so the controller has three attempts. If all three attempts fail mark the error as `ACTION_NEEDED`. The switch is now blocked from performing DPKM related functions until administrator marks the error as resolved.

`private synchronized void handleErrorDeletePeer(IOFSwitch sw, String errCode, String note, String ipv4Peer)`
Handles errors caused by a `DPKM_DELETE_PEER` message. Resends the message twice so the controller has three attempts. If all three attempts fail mark the error as `ACTION_NEEDED`. The switch is now blocked from performing DPKM related functions until administrator marks the error as resolved.

`private synchronized void handleErrorOther(IOFSwitch sw, OFDpkmBaseError inError, String errCode, String note)`
Handles get and missing errors caused by `SET_KEY`, `ADD_PEER`, or `DELETE_PEER` messages. Identifies the message that caused the error by extracting the msgType from the returned error message string. Resends the message twice so the controller has three attempts. If all three attempts fail mark the error as `ACTION_NEEDED`. The switch is now blocked from performing DPKM related functions until

administrator marks the error as resolved.

C.1.10 DpkmFlows.java

Flow configuration functionality of the Data Plane Key Management Protocol. Used for routing packets through WG interface, stopping packets from being sent between switches, or redirecting packets through a different port. Class utilised by start communication, end communication, rekeying and revocation procedures. Work in progress.[12]

`public class DpkmFlows extends Dpkm`

`public OFFlowAdd constructFlowAdd(IOFSwitch peerA, IOFSwitch peerB)`
Constructs a `FLOW_ADD` message for directing any packets received by peerA destined to peerB through the port containing the WG interface. Sets flow match conditions and actions/instructions to apply to packet. Currently a work in progress and only matches on IP packets because Wireguard operates on L3.

`protected OFFlowModify constructFlowDrop(IOFSwitch peerA, IOFSwitch peerB)`
Constructs a `FLOW_MOD` message for modifying an existing flow to drop any packets on peerA destined to peerB.

`protected OFFlowModify constructFlowModify(IOFSwitch peerA, IOFSwitch peerB)`
Constructs a `FLOW_MOD` message for changing an existing flow and routing any packets on peerA destined to peerB via the controller. This means packets will not be encrypted and sent through WG.

`protected void handleFlowRekeying(IOFSwitch peerA, IOFSwitch peerB, boolean before)`
Handler for rekeying procedure. Before procedure packets are redirected to standard port so are temporarily unencrypted. After procedure the flow is restored.

C.1.11 DpkmManagePeerResource.java

REST APIs for retrieving peers and adding/deleting a peer connection. Takes json data from UI and deserializes, executing corresponding function.[13]

[12] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmFlows.java`
[13] `https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmManagePeerResource.java`

```
public class DpkmManagePeerResource extends ServerResource
```

```
@Get("json") public List<DpkmPeers> retrieve()
```
Returns a full list of peer connections from the db in json format.

```
@Post public String add(String fmJson)
```
Adds a new peer connection for the switch information in the given json. Deserializes to get both peer switch information, sending `ADD_PEER` message on success.

```
@Delete public String delete(String fmJson)
```
Deletes a peer connection for db record matching the id in the json. Deserializes to get id, finds db record, sending `DELETE_PEER` message on success.

```
public static DpkmPeers jsonToDpkmPeer(String fmJson)
```
Converts peer information given in json format to a DpkmPeers object. Maps each json value to a field in DpkmPeers.

C.1.12 DpkmPeers.java

Object representing a peer connection between switch A and switch B. Stores useful information during a number of peer related procedures.[14]

```
public class DpkmPeers
```

```
public DpkmPeers()
```

C.1.13 DpkmRekeyResource.java

REST API for rekeying a switch upon expiry of its cryptoperiod. Takes json data from UI and deserializes, executing rekeying function.[15]

```
public class DpkmRekeyResource extends ServerResource
```

```
@Post public synchronized String rekey(String fmJson)
```
Rekey's switch matching the given json by running the configuration procedure. Deserializes to get switch information, sending `SET_KEY` message on success.

[14] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmPeers.java

[15] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmRekeyResource.java

C.1.14 DpkmRevocationResource.java

REST API for carrying out the revocation procedure with a compromised switch. Utility is decided by the administrator, either reconfiguring the switch with new public/private keys or terminating the switch entirely. Takes json data from UI and deserializes, executing corresponding function.[16]

```
public class DpkmRevocationResource extends ServerResource
```

```
@Post public String revoke(String fmJson)
```

Revoke the keys of the switch with matching info in the given fmJson. Calls revoke with the dpid and type of revocation to carry out.

C.1.15 DpkmStartCommunicationResource.java

REST API for starting communication between two connected peers. Takes json data from UI and deserializes, executing startCommunication function.[17]

```
public class DpkmStartCommunicationResource extends ServerResource
```

```
@Post public String start(String fmJson)
```

Starts communication for connection matching peer information in json. Deserializes to get both peer switch information, sending `FLOW_ADD` messages on success.

C.1.16 DpkmSwitch.java

Object representing a single switch in the DPKM protocol. Stores useful information during a number of switch related procedures.[18]

```
public class DpkmSwitch
```

```
public DpkmSwitch()
```

[16] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmRevocationResource.java

[17] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmStartCommunicationResource.java

[18] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/DpkmSwitch.java

C.1.17 IDpkmConfigureWGService.java

Defines the abstract functions for configuring WireGuard and provides a service for linking the REST APIs to methods in the module dpkmconfigurewg. Important interface to add services to the underlying module through extending the base IFloodlightService class.[19]

```
public interface IDpkmConfigureWGService extends IFloodlightService
```

Method definitions not included, given previously in C.1.4 and C.1.2.

C.2 Open vSwitch

This section provides a list of classes and methods added or modified in the OVS codebase. Visit or download the GitHub repository to see the source code[20].

- `/build-aux/extract-ofp-msgs:` lines 19, 197-198.
- `/build-aux/extract-ofp-fields:` lines 81-83.
- `/include/openflow/automake.mk:` line 3.
- `/include/openflow/dpkm-ext.h:`
  Openflow extensions for the Data Plane Key Management Protocol. This class defines the enumerators and structs representing messages and flags for carrying out specific functionality. Designed to match the format of messages in the floodlight controller.
  `enum ofp_dpkm_status_flag`
  `DPKM_STATUS` message flags indicating the state of WG and response type.
  `struct ofp_dpkm_status`
  Defines structure of `DPKM_STATUS` message.
  `struct ofp_dpkm_set_key`
  Defines structure of `DPKM_SET_KEY` message.
  `struct ofp_dpkm_delete_key`
  Defines structure of `DPKM_DELETE_KEY` message.
  `struct ofp_dpkm_add_peer`
  Defines structure of `DPKM_ADD_PEER` message.
  `struct ofp_dpkm_delete_peer`

[19] https://github.com/lukehengstenberg/DPKM_Floodlight/blob/master/src/main/java/net/floodlightcontroller/dpkmconfigurewg/IDpkmConfigureWGService.java

[20] https://github.com/lukehengstenberg/DPKM_OVS

Defines structure of `DPKM_DELETE_PEER` message.
`struct ofp_dpkm_test_request` `ofp_dpkm_test_reply`
Defines structure of `DPKM_TEST` messages (Only used for testing purpose).

- `/include/openvswitch/match.h:` line 142.
- `/include/openvswitch/meta-flow.h:` Added `dpkm_method` OXM type to `enum OVS_PACKED_ENUM mf_field_id`, lines 1086-1104.
- `/include/openvswitch/ofp-errors.h:` Added fifteen error messages to `enum ofperr`, lines 895-929.
- `/include/openvswitch/ofp-msgs.h:` Added message definitions to `enum ofpraw` and `enum ofptype`, lines 521-540, 592-598.
- `/include/openvswitch/ofp-util.h:` Added abstracts of message structs and decode functions.
- `/lib/flow.c:` Added support for `dpkm_method` OXM, lines 798-800, 1280-1282, 1727-1729, 1836-1838, 1973-1975.
- `/lib/flow.h:` Added support for `dpkm_method` OXM, lines 31, 120-121, 174-175.
- `/lib/match.c:` Added support for `dpkm_method` OXM, lines 490-496, 1472, 1535-1539.
- `/lib/meta-flow.c:` Added support for `dpkm_method` OXM, lines 286-287, 558, 810-813, 1158-1161, 1577-1580, 1852, 2117-2121, 2358.
- `/lib/meta-flow.xml:` Added support for `dpkm_method` OXM, lines 2982-2989.
- `/lib/nx-match.c:` Added support for `dpkm_method` OXM, lines 1054, 1093-1096.
- `/lib/odp-util.c:` Added support for `dpkm_method` OXM, lines 6131, 7235, 8585.
- `/lib/odp-util.h:` Added support for `dpkm_method` OXM, line 150.
- `/lib/ofp-match.c:` Added support for `dpkm_method` OXM, line 68.
- `/lib/packets.h:` Added support for `dpkm_method` OXM, line 121.
- `/lib/ofp-bundle.c:` Added message definitions to `static bool ofputil_is_bundlable(enum ofptype type)`, lines 295-301.
- `/lib/ofp-print.c:` Added message definitions, lines 1201-1209.
- `/lib/ofp-msgs.c:` lines 238-240.

- `/lib/ofp-util.c:` OpenFlow extensions to decode DPKM messages.
  `enum ofperr ofputil_decode_dpkm_set_key(const struct ofp_header *oh, struct ofputil_dpkm_set_key *kin)`
  Decodes an `OFPT_DPKM_SET_KEY` message. Maps the message fields to an abstract structure which is used to configure WG and format the status response. If subtype is not `SET_KEY` an error will be thrown.
  `ofputil_decode_dpkm_delete_key(const struct ofp_header *oh, struct ofputil_dpkm_delete_key *kin)`
  Decodes an `OFPT_DPKM_DELETE_KEY` message. Maps the message fields to an abstract structure which is used to unconfigure WG and format the status response. If subtype is not `DELETE_KEY` an error will be thrown.
  `ofputil_decode_dpkm_add_peer(const struct ofp_header *oh, struct ofputil_dpkm_add_peer *pin)`
  Decodes an `OFPT_DPKM_ADD_PEER` message. Maps the message fields to an abstract structure which is used to add a peer and format the status response. If subtype is not `ADD_PEER` an error will be thrown.
  `ofputil_decode_dpkm_delete_peer(const struct ofp_header *oh, struct ofputil_dpkm_delete_peer *din)`
  Decodes an `OFPT_DPKM_DELETE_PEER` message. Maps the message fields to an abstract structure which is used to delete a peer and format the status response. If subtype is not `DELETE_PEER` an error will be thrown.
- `/ofproto/ofproto-dpif-rid.h:` Added support for `dpkm_method` OXM, line 103.
- `/ofproto/ofproto-dpif-xlate.c:` Added support for `dpkm_method` OXM, line 4127.
- `/ofproto/ofproto.c:`
  Openflow extensions for the Data Plane Key Management Protocol. The functions below (lines 6270 to 6691) are used to modify the WireGuard configuration using script, handle messages received from the controller, and create status responses based on the state. This is where the main DPKM functionality takes place.
  `enum ofperr configure_wg(void)`
  Configures the WireGuard interface by executing the script in executable file ConfigureWG.sh which is positioned in root during setup. This script file generates the keys for the WG interface and sets up other aspects such as the listening port and IP routing. Returns 0 on success or error.
  `enum ofperr unconfigure_wg(void)`
  Unconfigures the WireGuard interface by executing the script in executable file UnconfigureWG.sh which is positioned in root during setup. This script file removes the current WireGuard configuration by deleting the WG interface and its keys, recreating it as a blank unconfigured interface. Returns 0 on success or error.

`enum ofperr add_peer_wg(struct ofputil_dpkm_add_peer pin)`
Adds a new peer to the WireGuard interface using the information stored in the given `add_peer` struct created prior from a received `DPKM_ADD_PEER` message. Sets the peer using its public key, WG ipv4 and local ip address space as the allowed-ips (accepts messages from these addresses), the ipv4 address of the switch as the endpoint (where to send packets), and keep-alive interval of 30s. Returns 0 on success or error.
`enum ofperr delete_peer_wg(struct ofputil_dpkm_delete_peer din)`
Deletes an existing peer using the public key stored in the given `delete_peer` struct created prior from a received `DELETE_PEER` message. Returns 0 on success or error.
`enum ofperr get_pubkey(char *publickey)`
Reads the public key from its storage file into char publickey. Returns 0 on success or error.
`enum ofperr get_wg_addr(char *wg_addr)`
Gets the ipv4 address of the wireguard interface wg0 directly from the running system using the interface name and copies it into char `wg_addr`. Returns 0 on success or error.
`static enum ofperr handle_dpkm_set_key(struct ofconn *ofconn,`
`const struct ofp_header *oh)`
Handler for any `DPKM_SET_KEY` messages received from the controller. Decodes the message, runs the necessary script to configure WG, encodes a `DPKM_STATUS` response and sends it to the controller. Returns 0 on success or error.
`static enum ofperr handle_dpkm_delete_key(struct ofconn *ofconn,`
`const struct ofp_header *oh)`
Handler for any `DPKM_DELETE_KEY` messages received from the controller. Decodes the message, runs the necessary script to unconfigure WG, encodes a `DPKM_STATUS` response and sends it to the controller. Returns 0 on success or error.
`static enum ofperr handle_dpkm_add_peer(struct ofconn *ofconn,`
`const struct ofp_header *oh)`
Handler for any `DPKM_ADD_PEER` messages received from the controller. Decodes the message, runs the necessary script to add a new peer, encodes a `DPKM_STATUS` response and sends it to the controller. Returns 0 on success or error.
`static enum ofperr handle_dpkm_delete_peer(struct ofconn *ofconn,`
`const struct ofp_header *oh)`
Handler for any `DPKM_DELETE_PEER` messages received from the controller. Decodes the message, runs the necessary script to remove a peer, encodes a `DPKM_STATUS` response and sends it to the controller. Returns 0 on success or error.
`static enum ofperr handle_single_part_openflow(struct ofconn *ofconn,`
`const struct ofp_header *oh,`
Modified to handle DPKM messages, lines 8963-8973.

- `/ofproto/ofproto.h`: Added prototypes of previous methods, lines 55-56, 567-573.
- `ConfigureWG.sh`: Script for configuring WireGuard.
- `SetupTestNetworkWG.sh`: Script for quick setup of network topology.
- `UnconfigureWG.sh`: Script for unconfiguring WireGuard.
- `dpkm_mn_custom.py`: Script for custom mininet topology.

C.3 Application UI

This section provides a list of all script functions behind the Application UI. Visit or download the GitHub repository to see the full source code[21].

`function startRefresh()`
Refreshes all of the values on the page every fifteen seconds.

`function LoadSwitches()`
Gets all of the connected switches from the controller.
GET call to `/wm/core/controller/switches/json`.

`$("#btnConfigureWG").click(function ()`
Checks the data in the configureWG modal form and calls the ConfigureWGSwitch function.

`function ConfigureWGSwitch(switchid, cryptoperiod)`
Post method for sending a `DPKM_SET_KEY` message and setting the cryptoperiod on success.
POST call to `/wm/dpkm/configure/json`.

`function GetWGSwitches()`
Gets all of the WireGuard Configured switches from the database.
GET call to `/wm/dpkm/retrieve/json`.

`function cryptoLog(switchid)`
Stops multiple rekey requests by temporarily pushing/popping switch id's to an array.

`function Rekey(switchid, cryptoperiod)`
Rekey method for generating a new key upon expiry of the cryptoperiod.
POST call to `/wm/dpkm/rekey/json`.

[21] `https://github.com/lukehengstenberg/DPKM_Floodlight-webui`

`function DeleteSwitch(id)`
Delete method for sending a `DPKM_DELETE_KEY` message on success to the specified switch.
DELETE call to `/wm/dpkm/delete/key/json`.

`$switchesSource.change(function()`
Removes the selected source switch as an option in the target switch dropdown. Populates the target switch dropdown with the remaining options.

`function updateField(field)`
Updates the IP address field based on the selected switch id.
GET call to `/wm/dpkm/retrieve/json`.

`function GetWGSwitchesModal()`
Gets all configured switches from the database populating drop-downs.
GET call to `/wm/dpkm/retrieve/json`.

`$("#btnAddPeer").click(function ()`
Checks the data in the addPeer modal form and calls the AddPeer function.

`function AddPeer(switchIdSource, ipv4Source, switchIdTarget, ipv4Target)`
Post method for sending a `DPKM_ADD_PEER` message to add a new peer connection.
POST call to `/wm/dpkm/add/peer/json`.

`function GetWGPeers()`
Gets all of the WireGuard Peer connections from the database.
GET call to `/wm/dpkm/retrieve/peers/json`.

`function DeletePeer(cid)`
Delete method for sending a `DPKM_DELETE_PEER` message on success to the specified switch.
DELETE call to `/wm/dpkm/delete/peer/json`.

`function StartCommunication(cid)`
Start communication method for sending a `FLOW_MOD` message on success to the specified peers.
POST call to `/wm/dpkm/start/json`.

`function EndCommunicationModal(cid)`
Toggles the end communication modal and maps the cid of the table row.

`$("#btnEndCommunication").click(function ()`
Checks the data in the end communication modal form and calls the EndCommunication function.

`function EndCommunication(cid, endType)`
End communication method for terminating all or WG p2p communication.
POST call to `/wm/dpkm/end/json`.

`$("#btnCompromiseNode").click(function ()`
Checks the data in the simulate compromise modal form and calls the CompromiseNode function.

`function CompromiseNode(id)`
Compromise node method for simulating compromise of a switch. Used to initialise the conditions for the revocation procedure.
POST call to `/wm/dpkm/compromise/json`.

`function HandleCompromiseModal(dpid)`
Displays the handle compromise modal for the administrator to take action. Modal fixed on display until some action is taken.

`$("#btnHandleCompromise").click(function ()`
Checks the data in the handle compromise modal form and calls the HandleCompromise function.

`function HandleCompromise(dpid, revType)`
Carries out the revocation procedure based on the administrators chosen action. Either completely unconfigures the node by removing from DPKM system or reconfigures by generating new keys.
POST call to `/wm/dpkm/revoke/json`.