NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

**SOFTWARE-DEFINED NETWORKS:
PROTOCOL DIALECTS**

by

Michael Sjoholmsierchio

December 2019

Thesis Advisor:	Geoffrey G. Xie
Co-Advisor:	Britta Hale

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2019	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE SOFTWARE-DEFINED NETWORKS: PROTOCOL DIALECTS	**5. FUNDING NUMBERS** RCNEF
6. AUTHOR(S) Michael Sjoholmsierchio	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research , Arlington, VA 22203	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Software-defined networks (SDNs) are attractive to businesses and the military because they enable centralized and policy-based control at per flow level. However, current SDN standards by the Open Networking Foundation do not require the use of encryption or authentication for communication between controllers and switches. We propose a novel method to add message authentication to SDN control plane traffic via the use of a protocol dialect. A protocol dialect is a variation of an existing implementation of an open-source protocol such as OpenFlow, achieved by either adding proxies or directly modifying the binary code to incorporate new security measures or remove unused features. This research provides a framework for systematic creation and evaluation of a protocol dialect, and presents a novel design of a protocol dialect for OpenFlow. The protocol dialect includes three derivatives and provides authentication that not only is independent of Transport Layer Security (TLS) but also may mitigate some attacks, e.g., cipher-suite downgrade attacks, against TLS. Performance measurements from a Mininet experiment show that the derivatives did not significantly impact the communication latency of OpenFlow, adding less than 1% overhead when TLS is not enabled and less than 22% with TLS enabled.

14. SUBJECT TERMS network security, software-defined networks, protocol dialect	**15. NUMBER OF PAGES** 151
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

SOFTWARE-DEFINED NETWORKS: PROTOCOL DIALECTS

Michael Sjoholmsierchio
Lieutenant, United States Navy
BS, Norwich University, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN CYBER SYSTEMS AND OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
December 2019

Approved by: Geoffrey G. Xie
Advisor

Britta Hale
Co-Advisor

Dan C. Boger
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software-defined networks (SDNs) are attractive to businesses and the military because they enable centralized and policy-based control at per flow level. However, current SDN standards by the Open Networking Foundation do not require the use of encryption or authentication for communication between controllers and switches. We propose a novel method to add message authentication to SDN control plane traffic via the use of a protocol dialect. A protocol dialect is a variation of an existing implementation of an open-source protocol such as OpenFlow, achieved by either adding proxies or directly modifying the binary code to incorporate new security measures or remove unused features. This research provides a framework for systematic creation and evaluation of a protocol dialect, and presents a novel design of a protocol dialect for OpenFlow. The protocol dialect includes three derivatives and provides authentication that not only is independent of Transport Layer Security (TLS) but also may mitigate some attacks, e.g., cipher-suite downgrade attacks, against TLS. Performance measurements from a Mininet experiment show that the derivatives did not significantly impact the communication latency of OpenFlow, adding less than 1% overhead when TLS is not enabled and less than 22% with TLS enabled.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API	application program interface
ABC	Attribute Based Cryptography
DiD	Defense in Depth
D1	Derivative 1
D2	Derivative 2
D3	Derivative 3
FS	future secrecy
HMAC	Hash Message Authentication Code
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
IBC	Identity-Based Cryptography
IKM	Initial Keying Material
IP	Internet Protocol
IDS	Intrusion Detection System
IPSec	IP Security
MITM	man-in-the-middle
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
NPS	Naval Postgraduate School
OVS	Open vSwitch

OOB	out-of-band
PKG	private-key generator
PCS	Post-Compromise Security
PKI	Public Key Infrastructure
RFC	Request for Comments
RMF	Risk Management Framework
SSH	Secure Shell
SNMP	Simple Network Management Protocol
SDN	software-defined network
SP	Special Publication
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TFTP	Trivial File Transfer Protocol
TTP	trusted third party
VM	virtual machine

Acknowledgments

To my wife, thank you for supporting me through every step of this research. Your support at home, on the road, and at work was essential to me completing this thesis.

To my parents, thank you for reading and listening to me brainstorm at each stage of development.

To Dr. Geoffrey Xie, thank you for your mentorship. It has been an honor to work with you and learn how to improve in class and on this thesis. Your belief in me inspired me to be a better student and researcher.

To Dr. Britta Hale, thank you for always challenging me to learn more and improve each part of this thesis. I greatly appreciate your patience with me in understanding new topics.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

A software-defined network (SDN), compared to traditional routing, provides commercial businesses and the military centralized control and management of communication networks. SDNs are also well suited to the growing field of cloud services and data centers due to their adaptability, scalability, security, and logical control, and are capable of enhanced centralized control due to the separation of the control-plane and data-plane [1]. This separation of traffic types provides opportunities but creates security vulnerabilities [2]. Consequently, we have chosen SDNs as the first platform for implementing a new security architecture. SDNs are suitable for this new security method because of their desirable characteristics such as centralized network control, traffic monitoring, logical control, threat detection, and security policy enforcement [2]. The OpenFlow protocol governs the control-plane traffic of a typical SDN [3] and will be our focus for security improvements.

The components and communication protocol for SDNs are open-source, therefore, allowing the modification of system components and the protocol within the boundaries of the standard to improve network security. We define a protocol dialect as a variation of an existing protocol at the binary level to incorporate additional security measures while maintaining the core functionality of the protocol. Protocol dialects are unique, plug-and-play, and customizable for individual organizations. This research aims to identify and demonstrate the process of dialecting a protocol with security controls to improve authentication and integrity. It is anticipated that protocol dialecting could be applied to other protocols and customized to enhance information security according to an organization's needs.

The OpenFlow protocol manages the control-plane traffic of SDNs between a controller and its switches and is governed by the OpenFlow Foundation [3]. OpenFlow Switch Specification Version 1.5.1 does not require encryption, but Transport Layer Security (TLS) is the recommended default security mechanism [3]. The lack of a required encryption mechanism is one of the first indicators that a protocol dialect may be a candidate for improved security. Besides a lack of encryption, other indicators that a protocol dialect may be a candidate are a lack of authentication or integrity checks inherent to a protocol. Transmission Control Protocol (TCP) is provided as the default protocol to ensure the transfer of data between SDN controllers and SDN switches [3]. TLS is the only optional security suite recommended by the standard and is not fully implemented by all vendors [4].

SDNs require additional security controls to ensure information security [4]. The weak separation of control plane and data plane traffic commanded by a single controller presents a single-point of failure and new attack vectors against an SDN controller's logic [5]. Due to a lack of existing security support and non-perfect security, protocol dialects can increase the time and effort spent by an attacker and become a method of Defense in Depth (DiD). DiD is a security defense model that relies upon more than one layer of security systems or measures to increase the difficulty of an attack. This research presents a method by which a custom-tailored security suite may be chosen, implemented, and tested to improve security based upon requirements of the system manager.

1.1 Problem Statement

OpenFlow, like many other communication protocols such as Telnet, SNMPv2, and Trivial File Transfer Protocol (TFTP), lack inherent information security controls without physical security [4]. For example, OpenFlow Switch Specification Version 1.5.1 does not require the use of encryption or authentication for communication between controllers and switches in an SDN [3]. However, being an open-source protocol, it is a candidate for modification into a protocol dialect. We propose protocol dialects as a method to improve information security. Protocol dialects increase the effort expended by an attacker against an SDN via the OpenFlow protocol with or without TLS enabled. Protocol dialecting may be performed in addition to a recommended security mechanism, such as TLS for OpenFlow, and we propose it as a method to apply DiD. The system of protocol dialects will not be secret nor depend upon the same secrets of an existing security mechanism such as TLS. OpenFlow is currently vulnerable in the aspects of confidentiality, integrity, and authentication, particularly in the case of a man-in-the-middle (MITM) attack [4]. A MITM scenario consists of a case where attackers are able to insert themselves between systems or operations. A MITM may present new or unique attack vectors that are not typically possible through external interface attacks only. For the case of a MITM, the development of a security system for DiD included in a communication protocol that causes an attacker to spend more resources and time to perform similar network attacks.

1.2 Research Questions

The introduction of a protocol dialect into an existing communication protocol adds complexity to a protocol that is not readily noticeable. An increase in complexity requires an attacker to spend more time and effort to perform an attack on a network. This thesis addresses the following questions:

- Can a protocol dialect be effectively implemented into an existing protocol to improve information security?

Additional questions inherent to the addition of a protocol dialect to OpenFlow in SDNs are the following:

- How effective is the security provided by protocol dialects?
- Which attacks on TLS, if any, can protocol dialects inhibit?

1.3 Thesis Organization

Chapter 2 contains background on information security requirements and systems, the OpenFlow protocol, and related security systems. Chapter 3 defines the design process to create a protocol dialect for a given base communication protocol. Chapter 4 presents one method to implement the process from Chapter 3 using the OpenFlow protocol based on our design case and assumptions. The completed experiment and results are provided in Chapter 5 to include problems encountered during implementation and any abnormal results. Lastly, Chapter 6 contains conclusions based upon our results to determine the accuracy of the hypothesis, efficiency of the dialect, and future work recommendations.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Background

The development of a protocol dialect builds upon the implementation of information security principles, the Risk Management Framework (RMF) developed by National Institute of Standards and Technology (NIST), and similar security systems. This research relies upon the principles and processes developed by NIST. NIST was chosen due to the applicability of this research to both government and civilian organizations. One way to frame and characterize security gains is through NIST's RMF [6]. Yet, the inspiration for the methodology for protocol dialects takes its origin from side-channel authentication and new research in the field of application program interface (API) specialization [7]. Combining these new fields with heavily relied upon principles and terms presents the opportunity for the similar field of protocol modification within the bounds of a protocol standard.

2.1 Information Security Principles

Terms and principles of the dialect design process are inspired by NIST Special Publication (SP) 800-12 Rev. 1 [8]. To determine how information security can be improved requires the classification of the types of security characteristics and what security controls assist in providing greater security. The major information security characteristics presented by NIST include confidentiality, authenticity, and availability [8]. Confidentiality protects data from unauthorized disclosure to anyone besides the sender and authorized viewer. An example of confidentiality ensures that an attacker sniffing a network is unable to determine the contents of packets traversing a network [8]. Integrity is the ability to verify that data is from only a specific sender and has not been modified during transit [8]. Lastly, availability is the performance of a system to meet the needs of an organization without being prevented normal operation by an attacker [8]. An organization's security policy will denote which information security objectives a dialect should be designed to achieve.

2.1.1 Defense-in-Depth

An emphasis on DiD motivates the improvement of security by layering various types of security mechanisms and protocols [6]. DiD has especially become more important in the field of network security to protect vital systems [9]. The use of multiple layers of protection provides a more complex strategy and system that an attacker must first identify and solve to access a system or network. Specifically in the year 2011, Jajodia et al. developed a DiD situation awareness tool to increase the ability for personnel and systems to self-identify vulnerabilities [9]. Jajodia et al. created a new tool called Cauldron to analyze maps of networks and different paths for attacks. While their tool aids an operator by providing visual aids, the underlying objective of both protocol dialects and Cauldron is to improve security through the use of DiD.

2.1.2 Risk Management Framework Design

We design the dialect process to improve the information security of communication protocols through the addition of security mechanisms. The dialect process consists of the identification of fields that may be modified in a protocol and the addition of a derivative in the identified fields. A derivative is a security mechanism that meets the security requirements of an organization. A related concept and process is the RMF provided by NIST in SP 800-37 Rev 2. The purpose of NIST's RMF is to provide guidance and a process for improving security and privacy by self-assessment and management decisions [6]. The RMF is similar to the protocol dialects process in the stages of preparation, categorization, selection, implementation, and assessment [6]. The following list from NIST and their RMF provides a summary of each of the above steps, which are similar to the protocol dialects process and complementary to those discussed in Chapter 3.

- Prepare: Establish and determine the organization's priorities and system that will be selected.
- Categorize: Determine the data at risk, potential impacts, and security vulnerabilities of concern.
- Select: Select the security mechanisms and control systems that will increase security of the data at risk determined in the categorize step.
- Implement: Choose an implementation method appropriate for the analyzed system and implement it.
- Assess: Determine if the controls implementation and selection meet the security needs determined during categorization [6].

The RMF should be used in conjunction with the protocol dialect process to provide a holistic approach to information security. Each organization should tailor its use of the RMF and protocol dialect process to meet its security needs and concerns. In the case of SDN, the priorities can be different than typical information systems due to the requirement for availability to ensure that the system can still function normally with added security features.

2.1.3 Post-Compromise Security

Post-Compromise Security (PCS) is an additional highly desirable security quality post-compromise and, as such, it has become a recent focus of newer security mechanisms by researchers such as Cohn-Gordon et al. [10]. They focus on recently developed and tested techniques to combat the loss of confidentiality to include ratcheting. Specifically, they present recent developments in the field by focusing on definitions, models, and operations of authenticated key exchange. One aspect of evaluating PCS includes the assessment of data confidentiality loss in the case of long-term key compromise [10]. A "ratcheting" system is one recent example of a method to improve PCS by the use of one-way functions to generate a chain of keys that are different and deleted after use [10]. This method is important and adds another aspect of security because the past keys cannot be calculated or created from present keys; therefore, the previous data cannot be decoded by an attacker even if they gain access to the present keys [10]. The benefits of post-compromise security should be included into a protocol dialect if possible.

2.2 The OpenFlow Protocol

Hu provides an insight into definitions and benefits of The OpenFlow protocol [11]. For example, he states that the OpenFlow protocol has enabled dynamic networks in response to growing requirements for bandwidth and management. There are two primary components of an SDN. Those two components consist of a single controller and as many switches as are required to complete a network. Hu describes the OpenFlow protocol as the management method for interaction and data transfer between an SDN controller and its switches to manage new flow requirements, topology, and setup [11]. He states that this network setup and topology allows for a manager or system operator to configure a whole network by interacting with one device instead of each traditional router individually. An overview of a typical SDN may be found in Figure 2.1.

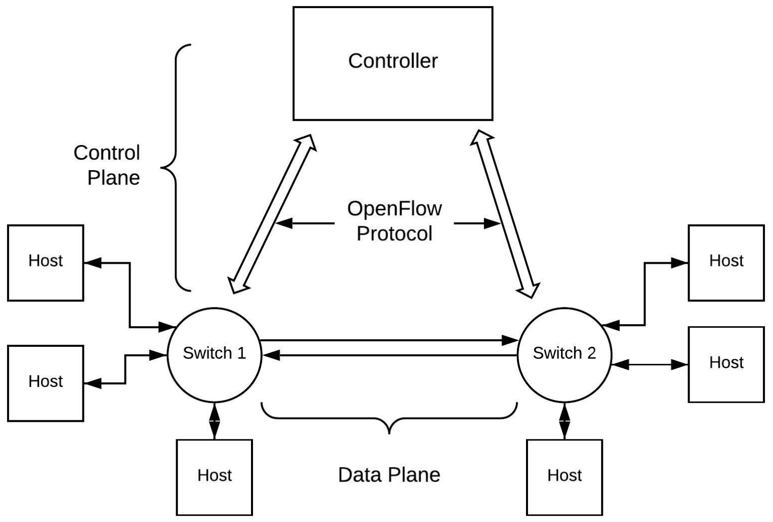


Figure 2.1. Example SDN Topology.

The example topology provided in Figure 2.1 provides a small network with only two switches and six hosts. The control plane is only between the switches and controller in the network, while the data plane exists between switches to pass data back and forth between hosts. Due to the event-driven nature of an SDN, each switch determines how to route and process packets from hosts using flow tables [11]. This flow is determined by the controller, and it can also be checked by the controller to determine how often the different forwarding rules have been applied [11].

For the transition from traditional networks, it is important that OpenFlow extensions can be added to commercial routers via software updates to enable them for OpenFlow [11]. However, adhering to the switch specification requirements established by the OpenFlow Foundation will ensure successful operation of controllers and switches using the OpenFlow protocol [3]. The ports for the data plane of a switch do not communicate directly with the OpenFlow channels [3]. The OpenFlow protocol is used to modify the flow entries in flow tables as determined by the controller [3]. A typical timing diagram of an SDN setup and operation can be found in Figure 2.2.

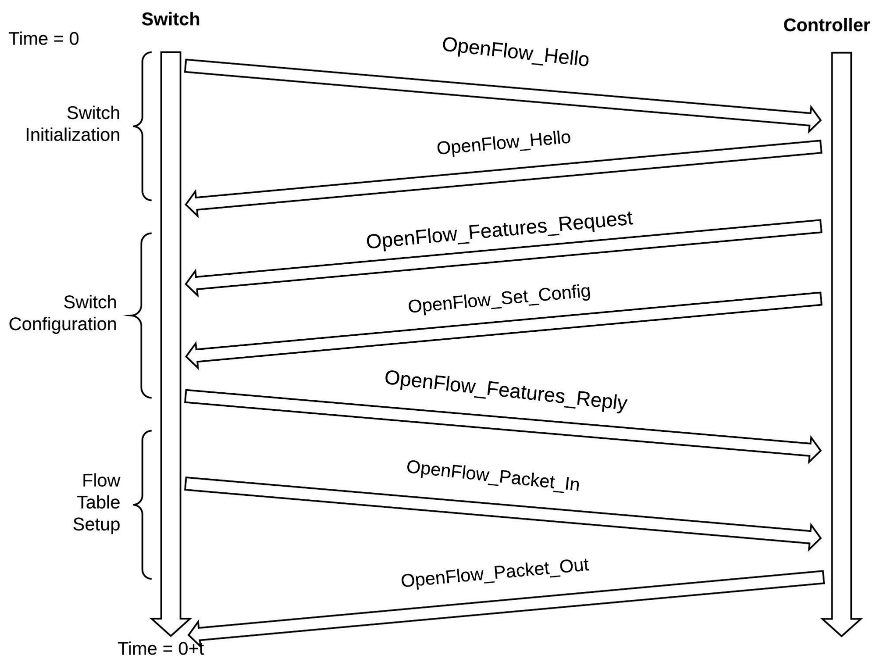


Figure 2.2. SDN Timing Diagram.

The timing diagram in Figure 2.2 was constructed using Mininet packet captures and analysis with Wireshark [12], [13]. In this timing diagram, it may be observed that a switch notifies its connected controller via an OpenFlow-Hello type OpenFlow packet. The initial detection and configuration can be observed through both the Hello type OpenFlow messages as well as the Features-Request, Set-Config, and Features-Reply. These packets transfer the necessary information to the controller to determine the capabilities of the switch and prepare it for operation [3]. Lastly, when a new packet arrives at the switch and it does not have a matching entry in a flow table, then it queries the controller for the correct action as found in the OpenFlow Packet-in and Packet-out messages [3].

2.2.1 OpenFlow Vulnerabilities

The OpenFlow standard only provides TLS as a recommended default security mechanism; [3] however, there are many other security mechanisms that should be considered for their use as a potential dialect. If OpenFlow does not have TLS enabled, it would be just as insecure as Telnet, Simple Network Management Protocol (SNMP)v2, and TFTP. Therefore, the following background is provided on recommended security systems as options to improve security via a dialect.

Even with TLS enabled, there are many different types of attacks on TLS as described by Sheffer et al. [14]. Some of those attack methods include the following:

- SSL Stripping
- STARTTLS Command Injection Attack
- BEAST
- Padding Oracle Attacks
- Compression Attacks
- Certificate and RSA-Related Attacks
- Theft of RSA Private Keys
- Diffie-Hellman Parameters
- Renegotiation
- Triple Handshake
- Virtual Host Confusion
- Denial of Service

A variety of attacks on TLS are well published and readily available to an attacker. OpenFlow and TLS present opportunities for an attacker to gain access to data, especially control-plane traffic, if TLS is not implemented correctly [4]. These attack methods include the ability to modify a network without requiring direct physical access to network components. This research does not aim to replace or modify TLS, but rather adds another defense layer for DiD to protect OpenFlow's control-plane traffic.

One attack against TLS that is particularly relevant to this research is DROWN [15]. DROWN is an example of a cross-protocol attack in the general category of a cipher-suite downgrade attack. In this type of attack, discovered by Aviram et al., TLS is potentially vulnerable if a previous version of TLS is still supported by the device using TLS to encrypt traffic [15]. The attacker is able to utilize the previous version of TLS to target a device such as a server. This type of attack can cause the decryption of traffic that is utilizing the TLS connection. See section 4.4 for more details on using Protocol Dialects to counter cipher-suite downgrade attacks through another authentication system.

The attack surface for an SDN is different from traditional networks. All decisions in an SDN are assigned to one controller for a network, and Xu et al. identified the possibility of new types of attacks on the decision center of the network. Xu et al. evaluated three SDN controllers and found 15 previously unknown vulnerabilities [5]. The attacks evaluated consisted of race conditions, which can occur in an SDN's controller [5]. Attacks that disable or harm the controller negatively affect or disable the rest of the network. The primary concerns for these types of attacks are availability and confidentiality [5].

The attacks discovered by Xu et al. are possible due to the asynchronous nature of the SDN control plane [5]. The new attack is called a state manipulation attack SDN and is evaluated using Xu et al.'s software ConGuard to determine if the race condition exists [5]. This type of attack can cause data to be stolen or availability to be threatened [5]. Xu's research demonstrates that with this new type of technology (SDN), variants of existing types of attacks are still being discovered.

2.3 Related Work

Specialization of binary source code to reduce vulnerabilities to code-reuse attacks has been a relatively new field of study in the past decade. API specialization and side-channel authentication are two defense methods related to protocol dialects. API specialization is a defense system designed against code-reuse attacks. An example of a code-reuse attack is the targeting of a particular function in a process to bypass non-executable memory protection to transfer control of that function to an attacker [7]. Previous defense methods in the field of code-reuse attacks include control-flow integrity and pointer controls to protect programs from being manipulated by attackers [16]. A recent implementation of code specialization to reduce generic attacks has been demonstrated through an API specialization system titled Shredder [7]. An API is the predetermined interface method for a program. The Shredder system analyzes a given API and reduces the opportunities for an attacker to perform code-reuse attacks [7].

Protocol dialecting is also related to the more historic applications of side-channel authentication, such as electromagnetic side-channel authentication. Both methods are similar in nature to the concept of protocol dialects because they both modify original code and system operation within the bounds of a system to add security. Side-channel authentication uses modification of electrical signals without influencing the data passed along the medium [17]. This identification of side-channel exploitation as a method of defense was identified by Sakiyama et al. Exploitation of keys for cryptographic systems can be a vulnerability to physical systems through power analysis [18]. Instead, side-channel authentication uses signals to add authentication data for verification using physical information leakage [18].

2.3.1 API Specialization

API specialization has been researched and formalized by Mishra and Polychronakis in their system Shredder [7]. To start their work, they identified shell code vulnerable to code-reuse attacks [7]. A code-reuse attack occurs when malware is able to exploit a system but utilizes functions that a specific interface did not require to perform its purpose [7]. The Shredder system provides a defender the ability to enforce customized policy on memory calls for a process to block excess functions not required. The use of system calls that are not required for the original program to function are also blocked [7]. The enforced control of program operation and function calls reduces those available to an attacker.

This shredder system also removes functionality that is not desired by the host organization. The removal of code-reuse opportunities is another type of DiD, because Shredder would analyze API code and remove those attack opportunities before public implementation or release [7]. While protocol dialects do not aim to remove capabilities or features inherent to the design of a standard, they do share the goal of modifying source code to increase difficulty of an attack on a network protocol.

2.3.2 Side-Channel Authentication

Electromagnetic side-channel authentication utilizes an existing electronic system to manipulate and verify the physical layer to provide additional layers of authentication security [17]. The manipulation of the physical layer with layering levels of noise and inserted additional data can provide a different form of authentication than a traditional Hash Message Authentication Code (HMAC) [17]. A HMAC is the product of a shared secret and message to create a Message Authentication Code (MAC). Side-channel authentication does not violate a protocol standard [17].

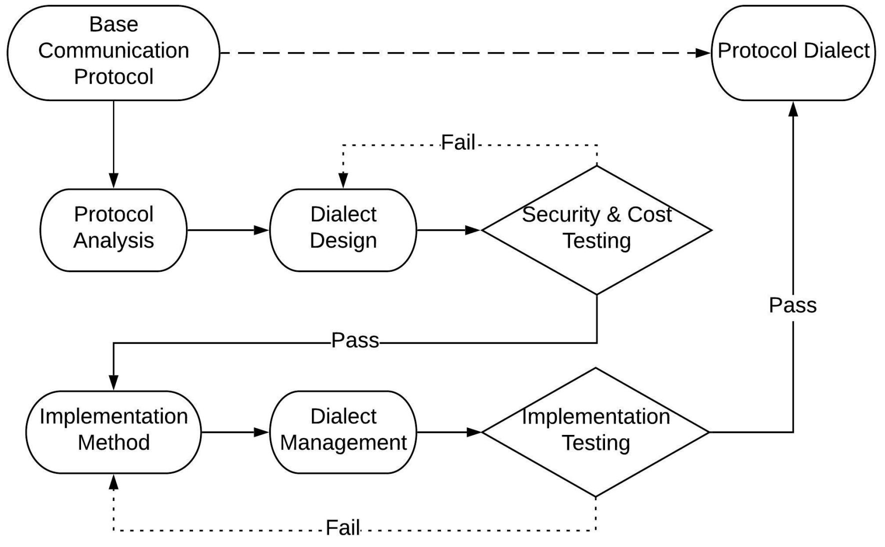
Side-channel authentication adds authentication data inside of an original transmission, while appearing like normal data when observed by an attacker. This extra information can be used to provide data transfer in methods not originally anticipated by a developer in a standard. A benefit of using side-channel authentication is the minimal cost and change to the original system, and it does not interact with the original software [17]. In an implementation by Perazzone et al., a side-channel authentication scheme consists of fingerprint-embedded authentication for additional security and to prevent sniffing attacks against the system [17]. The method defined by Perazzone expands upon the work by Sakiyama because it incorporates more security [17]. Security is enhanced by Perazzone because the side-channel authentication added includes a tag generated by secret keys and code books. This tag is then verified by the receiver before processing the data [17]. Side-channel authentication provides another security mechanism besides just HMAC [17].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: A Formulation of Protocol Dialect Design

This chapter describes the step-by-step process of creating a dialect for a protocol and integrating it into an existing base communication protocol. Dialecting is the process by which a dialect is created and applied to a base communication protocol. The dialect overview process summarizes the objectives, options, and actions at each stage of the process. The chosen dialect options and implementation for this research and the OpenFlow protocol is discussed in Chapter 4.

Each stage of the process requires independent analysis and decisions by an organization or a dialect designer; however, we present a typical or general option for each stage as well as our choice for each stage as an example. This process performs in linear order and involves intermediate analysis checks to ensure that the dialect will meet an organization's requirements before time is spent in implementation and management. If an intermediate analysis check does not pass an organization's requirements then repeat the previous stage is repeated. Figure 3.1 demonstrates the overall process to dialect a protocol.



The solid lines denote the dialect design process. The dotted lines denote self-assessment loops. The dashed line signifies that a modification has occured.

Figure 3.1. Protocol Dialect Process Overview.

Once a base communication protocol, such as OpenFlow, is chosen it must be analyzed in Stage 1: Protocol Analysis. This stage identifies the protocol standard and identifies potential locations where a dialect or modifications can be made that are within the bounds of the base protocol standard. In Stage 2: Dialect Design consists of selecting an existing security standard or scheme for the option(s) identified in a protocol. Next, in Stage 3: Security and Cost Analysis, both the security of the dialect chosen and overhead costs are analyzed. Security and cost results are then compared against the designing organization's security requirements. Examples of cost concerns in the case of an SDN or network are efficiency, speed, and availability. If the cost and security provided do not meet organizational requirements, then the dialect designer must return to the previous stage. If cost and security goals are met, then Stage 4: Implementation Method consists of determining the best method to include the dialect chosen into the existing protocol. In Stage 5: Dialect Management, the support mechanisms to support day-to-day dialect operation are determined. Lastly, in Stage 6: Implementation Testing, the dialect is tested to ensure that it correctly operates inside the base communication protocol.

The protocol dialect process can be performed on protocols that are already implemented in a system; however, it is preferably performed prior to a planned transition to a new network type such as an SDN. The concept and goals of this process are similar to NIST's RMF because of the similar objective to classify risks, select security controls to reduce risk, and implement those controls to improve information security [6]. Therefore, it is recommended that NIST's SP 800-37 is used in conjunction with the security system of protocol dialecting to provide a more comprehensive approach to network security. In summary, the protocol dialect process is versatile and designed to be chosen and implemented uniquely to an organization to fit their needs.

3.1 Definitions

The following terms are defined for this research:

- Base Communication Protocol: A selected communication network protocol that has an established standard such as Internet Protocol (IP), TCP, and OpenFlow.
- Dialect: A protocol dialect is a variation of an existing protocol at the binary level to incorporate additional security measures, while maintaining the core functionality of

the protocol.

- Derivative: A security mechanism incorporated into a dialect.
- Dialecting: The process of creating a dialect and including it into an existing base communication protocol.
- Dialect Designer: The administrator or team that performs the dialecting process.
- MS1: The dialect created for this research specific to OpenFlow and our implementation.

3.2 Base Communication Protocol

A base communication protocol is the original network protocol that exists in a currently employed system or will be installed in a future system. A base communication protocol should be selected with the objective of improving one or more information security characteristics. This process can be applied to open-source protocols and proprietary protocols if the standard is known and established. Authenticity, integrity, and/or confidentiality are potential qualities that will be improved by the addition of a dialect. The selection of a communication protocol may also require the adherence to any applicable Request for Comments (RFC)s to ensure that the protocol will still communicate with exterior systems if required. Overall, the most important aspect of the base communication protocol stage is the selection of the correct protocol, which can be modified and will improve the organization's information systems.

3.3 Protocol Analysis

Protocol analysis of a base protocol is the examination of the protocol standard selected as the base communication protocol. There are two main steps in this stage. The first step evaluates the protocol standard to determine the port numbers used, header layout, header content, header rules, message types, connection details, and included security options. Once evaluation of the protocol standard is complete, then identification of preferential option locations can occur. Understanding and analyzing the protocol standard to be modified aims to determine the best location for dialect modifications. Some locations for dialecting a protocol include fields in the base communication protocol's header and the experimenter type message.

Desirable characteristics of locations to place dialects are consistency, large bit-space, low requirements, and randomness. Consistency means that the dialect is included in every message or a header field used to transport every data message. However, a dialect may also be added in its own communication channel and message packets. For example, a normal packet may be sent in the command channel, but another logical layer of authentication can be added with separate communication packets containing a dialect. A large available bit-space with low requirements is an ideal location because it provides flexibility for modification and larger bit strength for digests. Overall, while there are desirable locations to include a dialect, the ability to modify a protocol or its operation is critical to performing the dialecting once an ideal location has been identified. The protocol analysis stage, as shown in Figure 3.2, is placed following the selection of a base protocol and before dialect design.

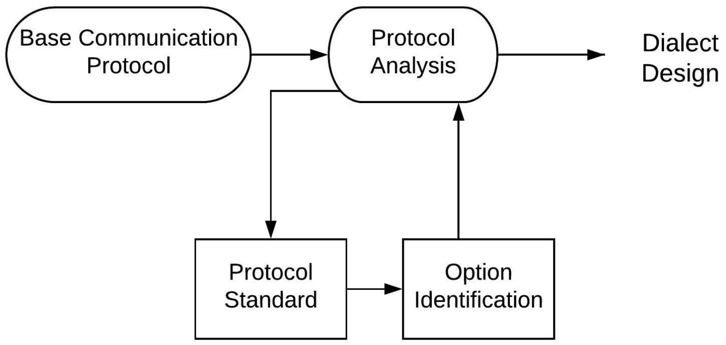


Figure 3.2. Protocol Analysis Stage.

3.3.1 Protocol Standard

The protocol standard for the base protocol is a required document for performing all dialect design and implementation. An example of a protocol standard for OpenFlow would be OpenFlow Switch Specification Version 1.5.1 (Protocol Version 0x06) [3]. An open-source protocol is an example of a protocol that is readily available for analysis and dialect option identification. The open-source characteristic is vital for the dialecting process because it allows for the transparency of use and requirements for each field in the chosen protocol and the protocol's overall operation. Observing the protocol in operation is beneficial to determine the potential system operation that may be used as a dialect. A standard is essential to this process, because it allows for the option analysis and identification of potential fields or methods of modification into a dialect. However, if a protocol is designed with a dialect in mind, then it can become part of the standard creation process. To determine standard operating procedures for a protocol, such as OpenFlow, the following standard requirement field types should be analyzed:

- Port numbers
- Header Layout
- Header Content
- Header Rules
- Message Types
- Message Handling
- Connection Setup
- Connection Maintenance
- Recommended Security Schemes

The above list of protocol standard characteristics provides the blueprint for the typical operation of a system that utilizes networking such as an SDN. A protocol dialect provides the opportunity to improve a different information security requirement compared to a recommended security system with a standard. For example, if a recommended security system already provides authenticity for the base communication protocol, it may be desirable for the organization to select a security scheme and derivative that improves confidentiality. The above list is only a general minimum requirement before proceeding to option identification. The analysis of each section of the above required fields for a protocol is then evaluated for its feasibility of modification within the bounds of the standard.

3.3.2 Option Identification

Once a protocol's standard has been reviewed, the next step in the evaluation process is option identification. Option identification consists of assessing areas of a standard that can support a dialect. The header of the protocol and experimenter message type are two examples of locations suitable for dialecting the OpenFlow protocol. Preferred fields of a standard are those that allow for change without violating the original standard. Some typical quality descriptors that are preferred for protocol dialecting include:

- Randomness

- Minimal Requirements
- Large Bit-size
- Employed Across Message Types
- Headers

Each listed quality influences the potential success of a dialect. A random field allows for modification because it typically requires fewer specific bits. Randomness also aligns with functions like a hash where randomness is also used towards a security principle. Random fields can be observed in a variety of protocols where there are typically counters that may be sequential over time, but start randomly. It should not be assumed that randomness will hide a security function; it can provide a small amount of obfuscation if a normally random field is used for side-channel communication or a dialect. When a standard has strict requirements, it may then require translation instead of direct modification to account for changes that are not allowed in the standard in a protocol. As long as translation can occur before devices that require a standard, then it is allowed to use a dialect.

Another desirable quality of a standard for dialection includes fields or message types that allow for large bit-size. A large bit-size is desirable because it allows for functions that require large digests. With the incorporation of larger digests, it is possible to include improved security mechanisms. For example, it is desirable to include a larger digest size such as 256 bits compared to 32 bits due to the possibility of forgeries.

For ease of implementation, it is beneficial to use the same dialect type across messages. Therefore, if there is a field or allowance for a dialect across messages this may reduce the memory required at a dialecting proxy or on a machine by only including one dialect.

Lastly, headers are a useful method of including a protocol dialect. Headers are useful because they are typically used across messages types, they have clear definitions on bit requirements, and they typically include some kind of randomness. A header provides an opportunity for a dialect to be used in a variety of conditions and messages.

3.4 Dialect Design

The dialect design stage consists of selecting a security scheme to be employed with the dialect as the security derivative. This stage includes the pairing of identified option locations that have limited requirements and a chosen security control. Once the combined locations and desired security qualities are known, then a list of security mechanisms, which ideally have previously been evaluated, can be created. Security concerns should be based upon previous organizational doctrine and the security manager's decision to ensure that priorities and controls reduce vulnerabilities or impacts of an attack. While this stage takes place after protocol analysis, it is set before security and cost analysis. The dialect design intermediate analysis for performance and security are performed after design to ensure that security, efficiency, and safety standards are met before implementing a dialect into the base communication protocol. If dialect design is not successful in meeting an organization's security or efficiency requirements, then this stage is repeated until both security and efficiency have met the organization's requirements. See Figure 3.3 for location of the dialect design stage inside the dialect creation process.

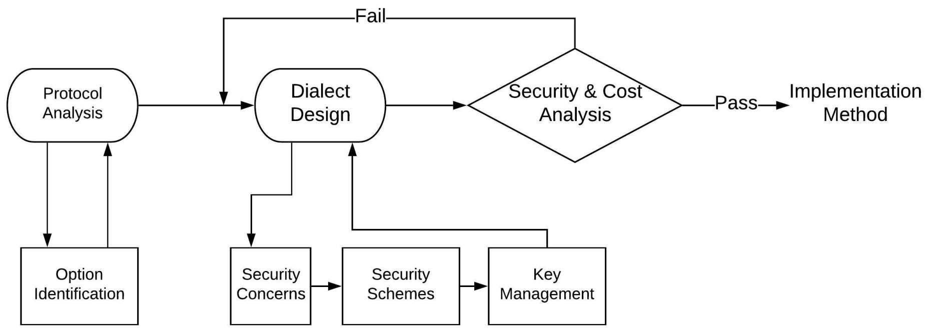


Figure 3.3. Dialect Design Stage.

3.4.1 Security Concerns

Information security risks and organizational impacts should be determined before a dialect is created. This requirement is set so that the dialect can adequately meet the needs of an organization based on its self-assessed vulnerabilities and impacts. A corresponding quality for impact is also required to determine the relative priorities between different types of threats [6]. This research assumes NIST's definitions of security threats to information systems [8]. Table 3.1 summarizes NIST's SP 800-12 Rev. 1 security concerns for information systems that apply to data in transit of the control-plane traffic of an SDN [8].

The above security qualities are at risk from various styles of attacks by an adversary. For example, if integrity of command packets in the control-plane to a switch could be modified

Table 3.1. Security Concerns Summary.

Security Concern	Objective
Confidentiality	Only the intended recipients are able to read the data in a transmission.
Integrity	The data in transit has not been modified intentionally or unintentionally.
Availability	The system is operating normally and not precluded from operation.
Authenticity	Data origin can be verified and entity authentication.

before reaching a switch, then an attacker could modify flows of an SDN switch or its configuration. Table 3.1 provides a list of security concepts that guide the dialect designer to determine what kind of derivatives should be chosen. For example, if the organization's security concern is confidentiality, then TLS may be one of the derivatives selected for a dialect. The chosen security objective of concern influences the derivative selected.

3.4.2 Security Schemes

A derivative is comprised of a security scheme, dependent upon the security goal. Table 3.2 shows example goals, e.g. confidentiality and authenticity, as well as example protocols and algorithms which may be used to achieve these. One or more derivatives are included inside of one dialect. The process to create a derivative starts with the selection of an information security principles as an objective. Once a security principle has been selected then a corresponding protocol choice and algorithm class should be selected. For cases when confidentiality is desired, encryption is selected, whereas signatures or a MAC algorithm are selected for authentication. The previous stage of option identification is then used to determine how much memory and processing resources are available to the security scheme. The protocol options for Table 3.2 are recommended by Samociuk [19].

The derivative design stack in Figure 3.2 shows example options of algorithms and types of security modules to be used inside of a derivative. A security algorithm should be chosen that supports the level of security required for an organization, as not all options provide equivalent levels or types of security. Other considerations for the security algorithm chosen include storage space and processing speed. The objective is to ensure that the equipment can support normal operations and traffic.

Table 3.2. Derivative Design Stack.

Dialect Derivative Security Layer	***Example Options***
Information Security Objective	Confidentiality, Authenticity
Protocol (e.g. Key Exchange, Key Distribution, and Authentication Protocols)	TLS, SSH, IPSec
Algorithm Class	MAC, Digitial Signature, Encryption
Algorithm	HMAC, SHA3, ECDSA, AES
Sub-Algorithms (If Required)	MD5, SHA-1, SHA3, BLAKE2

3.4.3 Key Management

Key management generates and manages the secret keys used in the protocol dialect. The secrecy and generation of keys is a critical concern for each of the above recommended security mechanisms in Table 3.2 [19]. The determination of a key management system and policy is required to ensure that the system can operate under normal conditions such as rollovers and in the case of a compromise. We considered the following key management mechanisms options for dialect MS1:

- Public Key Infrastructure
- Attribute Based Cryptography
- Identity-Based Cryptography
- out-of-band

Each of the above crypto systems and associated key distribution mechanisms presents different requirements, pros and cons, and complexity for incorporation into an existing communication system. As one example, Public Key Infrastructure (PKI) is already required for the TLS security scheme which is recommended by the OpenFlow Switch Specification standard [3]. Benefits of using a similar PKI system for both an inherent security mechanism such as TLS and protocol dialects would reduce the requirement to create another system for use when managing the dialect. Other benefits of PKI include the variety of deployment models such as up-cross-down, flexible bottom-up, and top-down [20] and extensive research that has been performed on the key management system. However, other systems provide benefits with regard to security issues that have been identified with PKI [21].

While there are many benefits to PKI, some of the PKI-based systems have proven insecure due to companies and governments being able to access keys in the infrastructure [21]. Another disadvantage of using the existing PKI includes the time and effort for setup of public and private key pairs [21]. These issues have prompted the necessity for identifying other options for this research. The flexibility needed for initial prototyping may require a different initial key management system or out-of-band (OOB) key distribution. An appropriate key management model must also be chosen to reflect the trust model incorporated into a system. In SDNs, a controller is inherently trusted; therefore, it may be used as the key manager in the system. The security objective of a derivative may be the same or different from already included security mechanisms; however, the use of different keys is preferred to ensure that the compromise of one set of keys does not compromise multiple security schemes. For example, if TLS is enabled in an SDN, separate keys should be used for the derivative.

3.5 Security and Cost Analysis

Security and cost analysis are required following dialect design. This stage ensures that the security and cost of the dialect meet that of the organization's needs. Security evaluation of the dialect is conducted using informal analysis and previous research. Previous research into the level of security provided by a scheme must be evaluated against any other security schemes used. The cost of the dialect is another important factor. Here we define cost as the resources and efficiency of the dialect in comparison to the original base communication protocol.

Each component of an SDN must be tested for the cost of processing, power, and storage to ensure that availability is not sacrificed due to the increased security. For example, the selection of a key management system will require memory for keys, algorithms, and working space. Key management may also influence the amount of extra traffic on the network not related to normal data flow. While either security or cost can be evaluated first, both must be checked to ensure that the dialect both improves security and does not sacrifice system availability. The Security and Cost Analysis stage is shown in Figure 3.4.

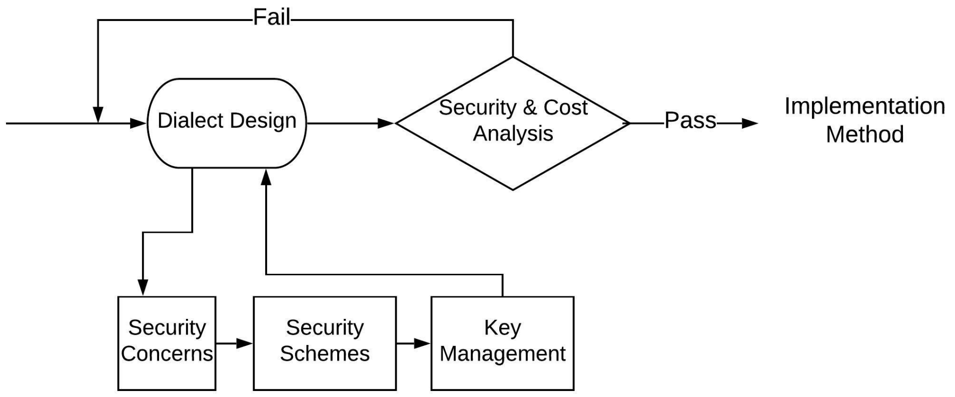


Figure 3.4. Security and Cost Analysis.

3.6 Implementation Method

The implementation method stage follows the analysis stage and confirms that security and cost requirements have been met. During this stage the method of implementation is selected that will determine how the dialect will be added to the system for long-term use. The implementation method supports the integration of the dialecting software with the existing system. Some security concerns and priorities of this stage include ensuring that the dialect cannot be bypassed and that it does not negatively affect system performance. See Figure 3.5 for the overall process diagram for this stage. The implementation method may also affect resource usage of the original system components. For example, a dialecting proxy would utilize its resources to perform the transformation of a protocol and, therefore, does not require the modification of original system components.

There are various methods by which a protocol dialect could be incorporated into a base communication protocol. Each implementation method provides unique benefits and requirements. Some examples of the various methods to implement a protocol include binary, proxy, and electromagnetic side-channel. We propose that one or more methods could be used to add a dialect. Chapter 4 details the one implementation method chosen for this research, which supported multiple derivatives.

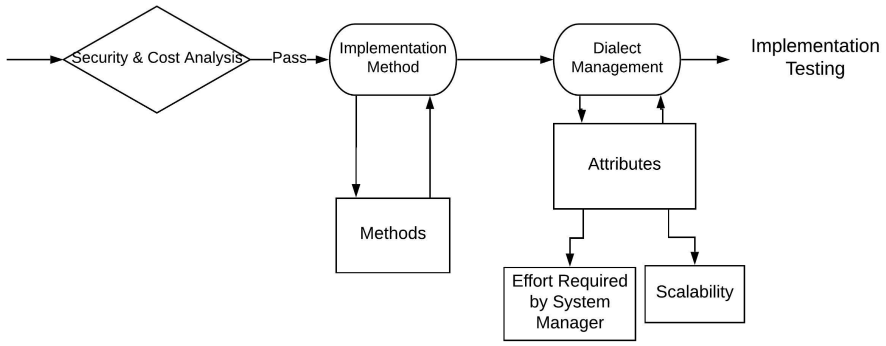


Figure 3.5. Implementation Method Stage.

3.6.1 Side-Channel Authentication Methods

Electromagnetic side-channel authentication is another method of implementing a dialect [17]. While not performed in this research, this method presents another way to add authentication to a protocol while still meeting the requirements of a protocol standard. Side-channel authentication has been performed through both electromagnetic methods and timing [17]. Background information on each of these two methods is provided in Chapter 2. The benefit of performing side-channel authentication is that the protocol does not need to be modified directly. Therefore, this method could be chosen if there are no preferred locations in which a dialect can be added directly. Side-channel authentication may or may not require the addition of other components in the network to verify signals as the dialect itself before the data stream is passed to the controller or switch. An example of an Electromagnetic side-channel authentication protocol dialect implementation method may be found in Figure 3.6.

Figure 3.8 demonstrates how a potential side-channel authentication system would work. For example, instead of the use of a proxy a signal encoder could be used to perform frequency, voltage, or other electrical signal modification. The direct modification of the electrical signal would be performed by an encoder at the after the original signal is generated by the controller. The authenticator would then authenticate the side-channel data before passing on the signal to the switch. The use of one implementation system does not limit the use of another type for additional layers of defense.

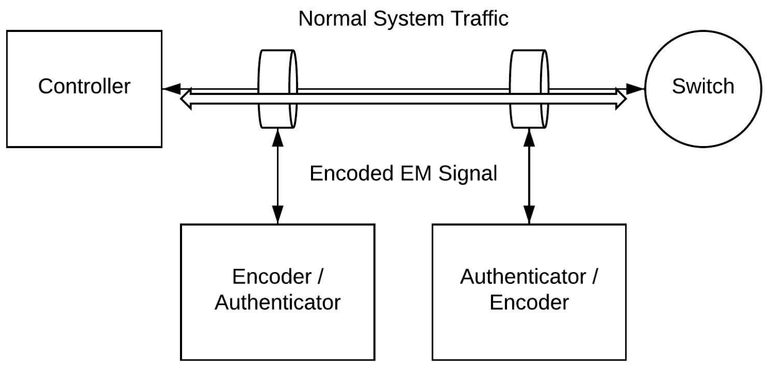


Figure 3.6. Electromagnetic Authentication Method.

3.6.2 Dialecting Proxy

A dialecting proxy is another method to implement a dialect. In this case, the proxy would be called a dialecting proxy and would intercept packets that are sent from a controller modify them in accordance with the dialect then send the new packet across the network. Once the packet reaches the destination, a corresponding dialecting proxy at the target intercepts the packet to verify the dialecting, and then it passes it to the target component. This process would be repeated for traffic that is sent back to the controller from the switch. A block diagram of the setup of this architecture and method can be found at Figure 3.7. This proxy implementation assumes that they are placed in-line with their counterpart devices and that one is assigned to every device in the network.

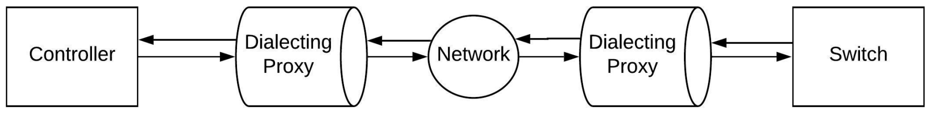


Figure 3.7. Dialecting Proxy Implementation Method.

3.6.3 Binary Modification

Binary modification for dialect implementation consists of changing the root code of a component to include the security mechanism. This form of modification allows for direct modification of open-source components. The new security mechanism introduced by a dialect must not negatively affect the operation of any existing security mechanisms. There are many benefits to binary modification of standard components such as a similar number of components, integrated hardware/software, and fewer opportunities for the added security control to be bypassed. Binary modification and inclusion of a dialect can be performed as shown in Figure 3.8. In this work, we will not perform binary modification of the controller or switch, but will demonstrate the use of dialects using proxies.

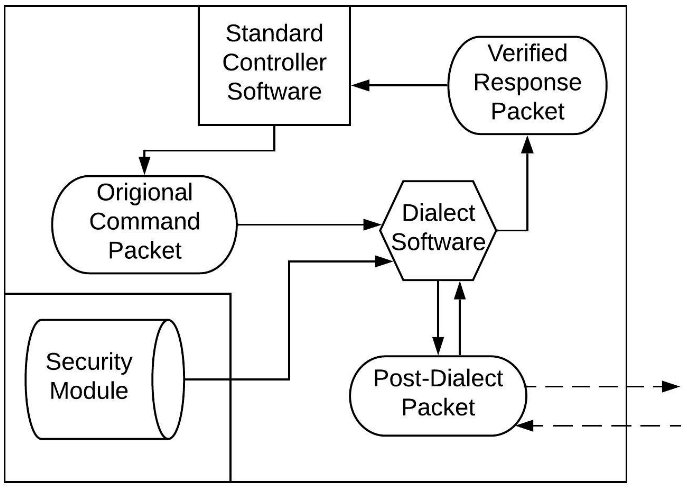


Figure 3.8. Dialect Binary Modification Process.

3.7 Dialect Management

The dialect management stage follows the selection process for determining the method to include the dialect. Dialect management focuses on the day-to-day maintenance of the dialect as well as recovery systems and procedures in the event of a compromise. The Dialect management Stage can be found below in Figure 3.9. Important attributes of the dialect implementation include effort required by the system administrator and scalability. Scalability in this case is defined as the ability for the system to support the expanded growth in the number and types of devices. This stage does not present a similar selection of options for generic use. Instead, this stage requires a selection of processes and policies to support the dialect.

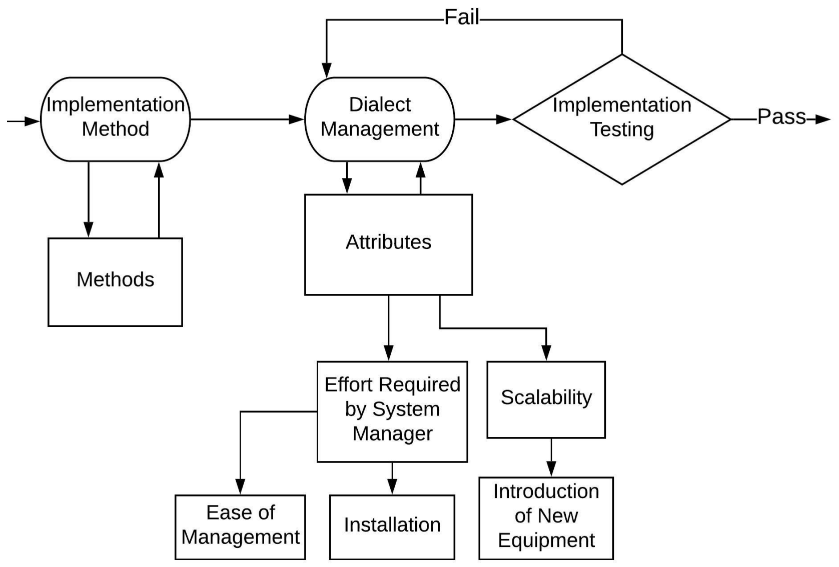


Figure 3.9. Dialect Management Stage.

3.7.1 Attributes

Scalability is an important attribute of dialect management especially in the case of an SDN because an SDN must be scalable. As a network changes and grows, an SDN is expected to adapt and grow as well. Therefore, the scalability of implementation and key management must adapt with the SDN to ensure that each part of the dialect is still performed in any additional or replaced hardware. Scalability is one consideration for dialect management to ensure that SDN size is not limited.

As a system grows and equipment must be replaced or added. It is preferred that minimal work is required of system administrators to add a dialect or expand the SDN network. The ideal effort required by a system manager would be none, while the most extreme would be coding modification of a switch or controller to incorporate a dialect. System manager's efforts are evaluated based on the time to install a dialect and ease of management. Lastly, time overhead caused by the dialect directly influences the likelihood that it can maintain a level of availability required for a network. The efficiency of dialect management is not directly related to the cost analysis already performed on the dialect.

3.8 Implementation Testing

Implementation testing is the final stage of development for a protocol dialect, as seen in Figure 3.10. In this final stage, the dialect is tested to ensure that it is successfully integrated into the protocol and system within security and availability requirements. Implementation begins with testing system performance and operation without a dialect. Then, with normal traffic speed recorded, operation is tested again with the dialect included in the protocol. The comparison between the two speeds provides a ratio. The percent increase in time delay for traffic depends upon an organization's needs and security concerns. While implementation testing should be performed virtually first with mock systems, it should also be tested with the physical devices to ensure that simulations are accurate.

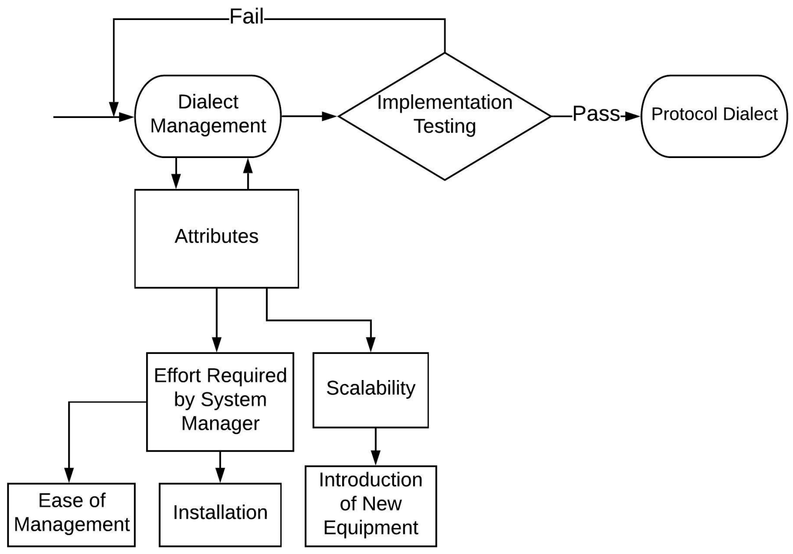


Figure 3.10. Implementation Testing.

3.9 Protocol Dialect

Upon completion of the protocol dialecting process, a specific protocol dialect is ready for implementation into an organization. The selected protocol dialect, as a product of the organization, would then modify traffic and components of the communication system selected. In this case, the OpenFlow protocol is tailored to meet security requirements of a specific organization. A dialect, if applied to a network, would be applied to all of the components that interact in a domain. In the case of OpenFlow, if a controller was modified, all switches managed by that controller would be required to have the same dialect. If authentication was added by a dialect, then if a switch was to be connected in the future it must be modified before connecting to the system to ensure that it could communicate with the system. While this may increase the time to add a switch to the system, the additional time required would depend upon the options and dialect chosen by the dialect designer.

Once a protocol dialect is in place, it provides another layer of defense against attacks. This additional layer, while providing more DiD, can also provide a means of attack detection if paired with an Intrusion Detection System (IDS). An IDS is a system designed to detect attacks against a system or protocol for the purposes of alerting an administrator or log. An IDS can be built directly into the dialecting software or proxy or as a separate component. For example, if an attacker was able to break TLS and send packets that pass encryption checks by either a controller or switch, the dialect would still have to be broken before an attacker was able to send authentic commands. The extent to which TLS attacks can be detected and a protocol dialect can be efficiently incorporated into OpenFlow will be examined in the next chapter through experimentation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: OpenFlow Dialect Design

This chapter describes the actions taken at each stage of the dialecting process to create a dialect titled Message Seal 1 (MS1) for the OpenFlow protocol. A summary of the options selected for the dialect in this research is provided in Table 4.1. This research analyzes the case in which an organization is transitioning to an SDN. The objective of this dialect is to add data authentication to OpenFlow in addition to the existing authentication performed by TLS. This additional security does not rely upon TLS and assumes that a compromise of TLS can occur. The dialect's implementation can be extended to also protect the initial handshake for setting up a TLS session. The dialect will continue to operate once TLS has been established to add another layer of security.

Table 4.1. Dialect Selection Summary.

Stage	Base Protocol	Protocol Analysis	Transport Security	Key Management	Implementation Method
Design Space	OpenFlow TCP IP	Transaction ID Length Version Type Data Buffer ID	TLS Secure Shell (SSH) IP Security (IPSec) MAC Scheme Signature Scheme Encryption Scheme	PKI OOB Identity-Based Cryptography (IBC) Attribute Based Cryptography (ABC) Ratcheting	Binary Proxy Electromagnetic Side-Channel Timing Side-Channel
Design Choice	OpenFlow	Transaction ID	HMAC	OOB	Proxy

The objective of this dialect is to increase the level of effort and time expended by an attacker to modify any switch settings or operation. The primary information security objective is authentication. If an attacker is able to modify switch configuration or flows then they may modify topology, operation, and availability nefariously or against an organization's preset configuration. Preferably, per the standard's recommendation, TLS is enabled in an SDN to secure the communication between the controller and switches [3]. The derivative security mechanisms and protocol dialect provided here is not designed as a replacement to TLS.

Figure 4.1 represents the final goal of the dialecting process. In this case, two derivatives add the desired security goals of the dialect. This diagram demonstrates that the Controller and Switch are no longer directly connected. Instead, each device is connected to a proxy which takes unchanged OpenFlow packets and creates the dialect. The dialect only exists between the proxies, and in this case, two are used. Derivative 1 (D1) demonstrates that a protocol dialect can be used to modify the OpenFlow header directly before translating back to a normal OpenFlow Hello at the target device. The second derivative, Derivative 2 (D2), comes in the form of two messages where the authentication message is an experimenter type message. Both of these derivatives form the dialect MS1.

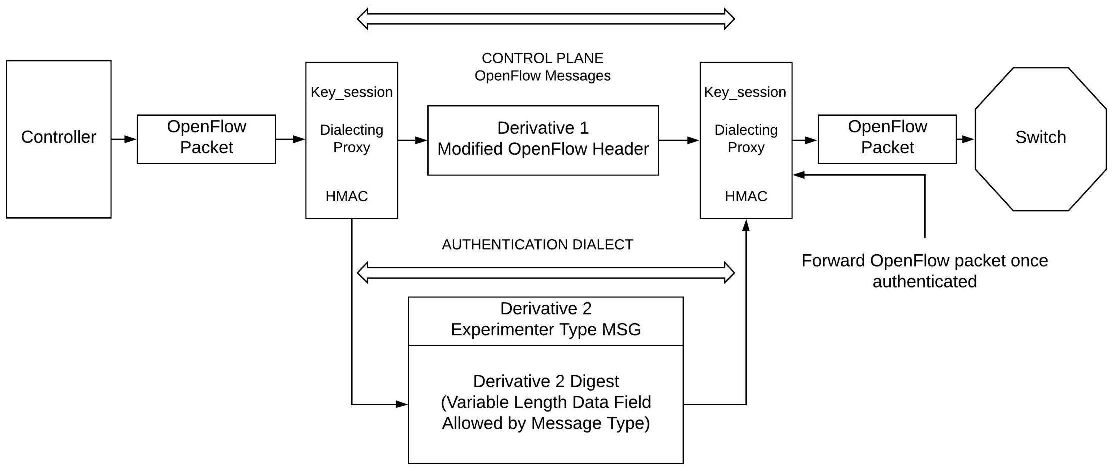


Figure 4.1. Dialect Overview.

4.1 Design Objectives

For this research, we assume that an attacker is able to observe and manipulate command traffic between the proxies of an SDN controller and its switches. We assume that the attacker can access the physical lines of communication between the switches and controller of an SDN. It is also assumed that an attacker has the ability to generate false command packets based on open-source knowledge of the protocol and location on the network. This scenario threatens confidentiality, integrity, authenticity, and availability of a network, particularly when TLS is not enabled.

TLS is not a perfect solution. Chapter 2 identified a variety of attacks on TLS. To improve overall system security, we propose protocol dialects as an additional security layer for DiD. The objectives of this dialect is to enforce earlier authentication, add protection to the TLS handshake, and additional authenticity of OpenFlow data. OpenFlow data includes version numbers, commands, responses, settings, and flow rules.

Without a dialect, SDN switches and controllers utilize TLS to perform authentication.

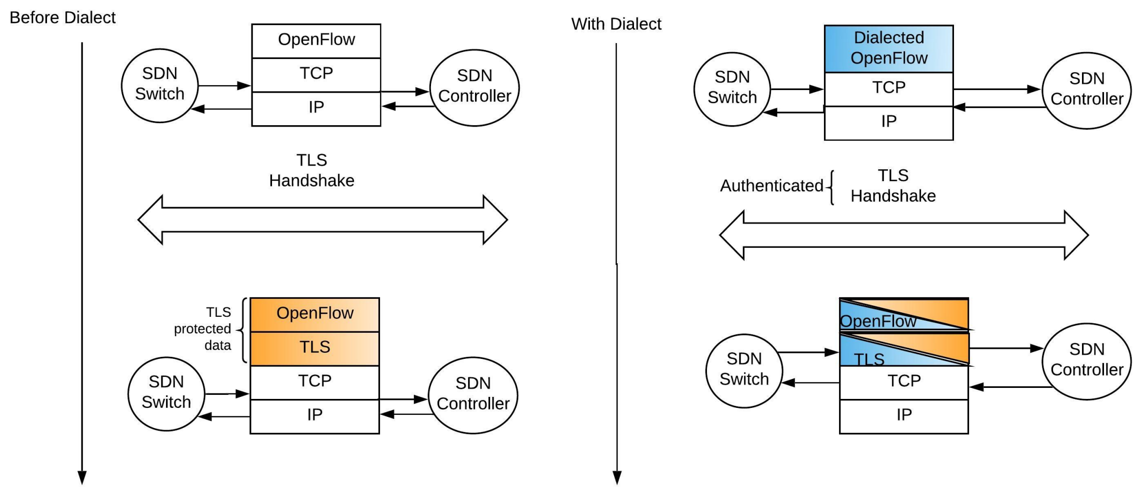
Instead, with D1, the authentication process is started with the first OpenFlow packet. Authentication is important because a modified OpenFlow packet can invoke changes in the network not desired by the system manager. Attacker modification of the original packets is also a concern because integrity and authentication is not enforced by the standard unless TLS is used. Therefore, a dialect designed for message authentication can be used to reduce the capabilities of an attacker. Without improved DiD, the potential for system modification or damage due to unauthenticated packets or commands can lead to system downtime or compromise, and a subsequent loss of availability.

4.2 Base Communication Protocol

We selected OpenFlow as the base communication protocol because of its open-source availability and highly desirable network characteristics, as discussed in Chapter 1. The operation of an SDN relies upon the correct decisions and commands by a controller to its switches. Since the controller is the single point of command and influences all other switches under its control, we selected OpenFlow as the first protocol to add protection to. We assume that TLS will be enabled to provide confidentiality; however, initial testing for this implementation was not encrypted because TLS is off by default in the emulation software selected for testing (Mininet) [12]. In the current OpenFlow standard, the Open Networking Foundation does not require the use of encryption or authentication for communication between controllers and switches utilizing OpenFlow [3]. The dialecting method that we selected for this implementation uses the existing message system for protocol enhancement.

Open-source controllers have enabled organizations to adopt and modify controllers to fit their organizational needs [22]. Some of the example benefits of SDN controllers identified by Khondoker et al. include ease of management, adaptability, efficiency, and security [22]. Open-source controllers and switches allow for easier analysis and modification for a new dialect. Open-source is preferred so that security can be added directly into the protocol; however, open-source is not required. Before modification and establishment can begin, a controller must be selected so the interface options and features available to the system can be identified [3]. A comparison of open-source controllers and their features was completed by Khondoker et al. [22]. The studied controllers include POX, Ryu, Trema, FloodLight, and OpenDaylight.

For this research, we selected both POX and ovs-testcontroller as the controllers for the experimental SDN. While the ovs-testcontroller was not evaluated by Khondoker it is mentioned in other sources in regards to enabling TLS in Mininet [23]. Both controllers are easy to use and and open-source. In this design, a dialect can add security to both OpenFlow and TLS which encrypts OpenFlow. For example, when a dialect is applied as a wrapper for a packet that is protected by TLS then additional integrity is added to the packet. A wrapper adds protection to the encrypted OpenFlow data as well as preventing modification between the proxies. Therefore, this process adds another layer of security to protect against modified command packets from reaching the devices attached to the proxies. The packet encapsulation and timeline for setup are illustrated in Figure 4.2.



Data protected by TLS is shown in orange, while data protected by the dialect is shown in blue. Cross-shading denotes data is protected by both.

Figure 4.2. Dialecting OpenFlow Packets with TLS.

An important consideration of the base communication protocol is the recommended security system provided by the standard. Since TLS is the recommended default security mechanism in the OpenFlow Switch Specification, OpenFlow dialects must not interfere with proper TLS operation [3]. For example, MS1 must not interfere with the TLS handshake from performing successfully, as demonstrated in Figure 4.3.

The timing diagram in Figure 4.3 was created from Wireshark packet traces captured during a Mininet experiment with OpenFlow operation while TLS was turned on. We incorporated protection against interfering with TLS by using two different derivatives

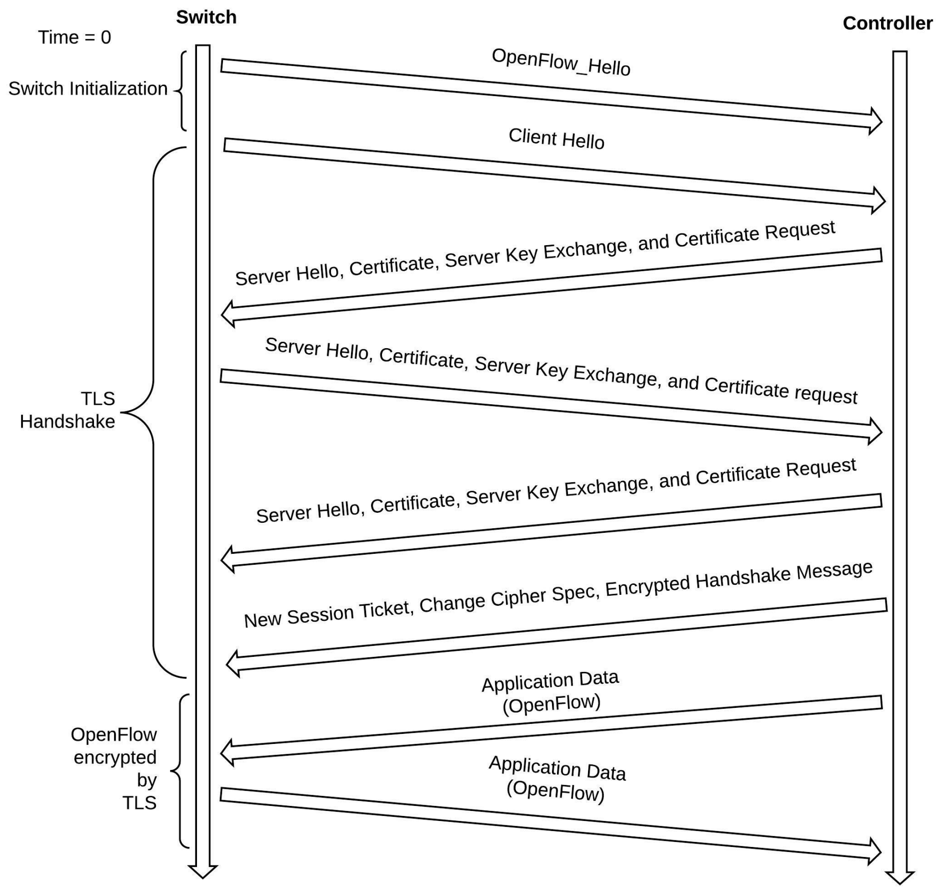

Switch
Controller
Time = 0
Switch Initialization
OpenFlow_Hello
Client Hello
Server Hello, Certificate, Server Key Exchange, and Certificate Request
Server Hello, Certificate, Server Key Exchange, and Certificate request
Server Hello, Certificate, Server Key Exchange, and Certificate Request
New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
TLS Handshake
Application Data (OpenFlow)
Application Data (OpenFlow)
OpenFlow encrypted by TLS

Figure 4.3. TLS Timing Diagram.

inside of one dialect. D1 provides security by an authentication code incorporated into the 32-bit transaction-ID that does not modify the total number of packets sent during the TLS handshake. D2 adds a 1:1 authentication message for every normal OpenFlow message and is used once TLS is established.

This second derivative relies upon a message type provided by the OpenFlow standard called the experimenter type message [3]. This type of message provides a variable length data field to increase the digest size and, therefore, strength of authentication. The modification of the transaction-ID as a signature field is less secure; therefore, its use is limited to before the TLS handshake (see Section 4.4 for more detail). During the setup process, each transaction-ID is augmented to improve authentication of both the setup and OpenFlow commands.

Overall, the modification of packets using the existing system and protocol is key to the MS1 dialect design. Adherence to an original design of a protocol makes the change less obvious to an adversary, however, still useful as a security feature. The use of the existing protocol and components removes the requirements that proxies are able to create or run the same software used in other components. A dialect may also be applied as a wrapper that only adds security around the existing protocol so that an existing security mechanism does not need to be modified or known by the proxies to still operate.

4.3 Protocol Analysis

For this research, we chose the OpenFlow protocol header and transaction ID as the first potential field for a useful dialect. The OpenFlow header is shown in Figure 4.4.

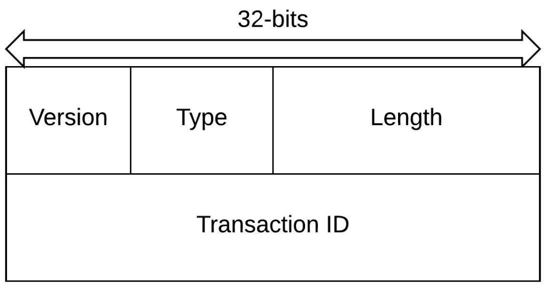


Figure 4.4. OpenFlow Header Layout. Adapted from [24].

The transaction ID field was chosen because it is utilized across all message types, is typically set randomly, and provides the largest field for manipulation by a dialect in the OpenFlow header [3]. The other fields in the OpenFlow header, which are smaller, are version, type, and length [3]. The primary restriction on the transaction ID field is that it is used for conversation tracking between a controller and a switch [3]. The transaction ID field is the same between a controller and switch in a normal SDN [3]. In a dialected SDN that manipulates the transaction ID field, a dialecting proxy should translate any modified field back to its original values before being provided to either the controller or the switch. We later determined that this requirement can be mitigated by providing a set transaction

ID number to the controller or switch at the proxy after the message has been checked. However, conversation tracking would be required for multiple switches connected to one controller. In this research only one controller and switch pair were tested.

The second potential option for the OpenFlow dialect was presented by functionality included in the protocol standard. The OpenFlow switch specification [3] provides another message type, called experimenter, that is not normally used except by vendors or researchers. This message type is not symmetric; however, it does provide for an identification field and then a variable length data field. This message type provides the opportunity for a dialect because it can be used for transporting a digest or authentication code of a sufficient length that a dialecting proxy can then confirm before transferring to the intended recipient.

4.4 Dialect Design

To determine security improvements beneficial to OpenFlow, we selected authenticity as the primary security objective of our dialect. We assume an attacker is able to observe all traffic, intercept, modify, and retransmit packets in a timely fashion. Therefore, since there are no included authentication or integrity checks built into the OpenFlow protocol, this is an ideal starting point for a security derivative. The process of designing a derivative begins with the selection of an information security principle, as noted in Table 4.2.

To add authenticity, we selected MAC as the class of algorithm because we wished to design a solution that is lightweight relative to TLS, as measured by computational overhead incurred. In this case, a MAC would be computed and checked for each OpenFlow packet using shared secrets preinstalled at both the controller and switches. Without knowing the shared secret, attackers would be forced to brute-force the MAC. We further selected HMAC because of its avalanche effect which will fill a normally random field with seemingly random bits that create a large change in output based on even a small change in input. Latter, in Section 4.7, the selection of key management methods that influenced the selection of HMAC instead of a digital signature will be discussed. The use of symmetric keys in this dialect design also led to the selection of HMAC for authentication and integrity.

The selection of dialect keys unique and separate from TLS is important to the derivatives in this design. Different keys for TLS and the dialect are necessary for security and implementation reasons. Dialect keys that are distinct from TLS keys ensure that leakage

Table 4.2. Summary of Derivative Design Choices.

Dialect Derivative Security Layer	***Selected Option***
Information Security Objective	Authenticity
Protocol (e.g. Key Exchange, Key Distribution, and Authentication Protocols)	Not Applicable
Algorithm Class	MAC
Algorithm	HMAC
Sub-Algorithms	BLAKE2b
Authentication Code Contents	Packet Data + Shared Secret

and vulnerabilities are not introduced against TLS. In the implementation of this system, the use of proxies ensured that the keys were different. We controlled key life via our key generation method demonstrated in Figure 4.5. .

The process to create the MAC tag follows three steps. The first step is the generation and delivery of OOB Initial Keying Material (IKM). IKM is the root keying material which will be used to seed and synchronize the key generators and is possessed only by the parties using the dialect. This IKM is used in conjunction with salts to create a pre-TLS key by a HMAC-based Extract-and-Expand Key Derivation Function (HKDF). For this design, we selected time as our salt to synchronize values without having to store values in the proxies. HKDF uses SHA-512 to generate an ephemeral key for the MAC tag creation via another hash algorithm, in this case, BLAKE2b. Use of different hash algorithms for key generation and tag generation provides additional DiD. The keys used in the BLAKE2b function for MAC tag generation will change every five seconds for improved security due to short MAC tag size in D1. BLAKE2b generates a key with the length that is required for the transaction ID. Key management and generation follows the structure in Figure 4.5:

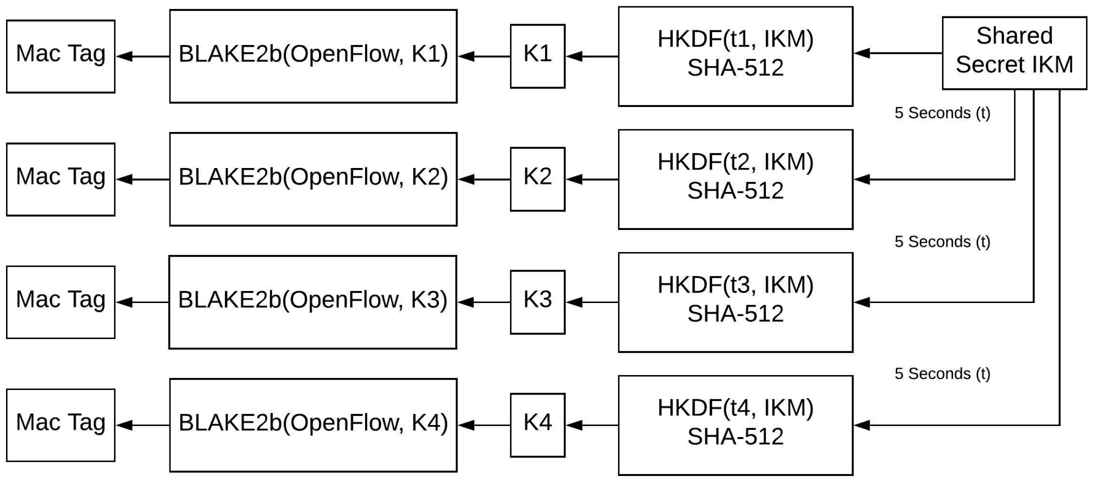


Figure 4.5. Key Generation Process.

While the 32-bit field is the largest modifiable field in the OpenFlow header, a 32-bit long MAC is relatively weak against forgeries. Therefore, the protocol governing the establishment of the MAC keys and digests must support a relatively short-lived MAC in order to mitigate the risks of using a MAC of a short bit-length. The primary concern due to the small bit-space available for modification is the life of the MAC tag. Not only are MAC tag forgeries a concern, but second preimage attacks are possible; this means that another key may be found that can produce the same MAC tag in a short time [25].

Second preimage is the primary concern as a threat to D1. A second preimage occurs when a message that creates the same MAC tag as the authentication tag will contain different OpenFlow commands than the original message. These types of message are a concern because they pose a threat to authentication. If a second preimage is available for a given MAC tag then that OpenFlow message could potentially change the network topology or perform unauthorized operations on the network. Therefore, as a countermeasure to the potential for attacks on the life of a MAC tag, we have developed a key generation scheme that ensures a short MAC life.

Five seconds was selected as the amount of time for key rollover because Mininet operation also relies upon 5 seconds. This time frame is also shorter than the time required to break half of the bit-strength of that provided by the field. The amount of time to complete a SDN controller and switch setup without TLS enabled is approximately 5 seconds. The SDN controller and switch also check for liveliness every 5 seconds; therefore, 5 seconds ensures that each new key and MAC life persist for one liveliness check. The estimate for the time to perform the MAC in this research correlates with previous estimates in the original research for Blake2 [26]. The machine specifications used to determine the time to create a MAC using python was a 1.8 gigahertz processor and 8 gigabytes of random access memory. The justification and selection of 5 seconds is demonstrated below:

$$\textit{Time to Create } 32 - \textit{bit MAC} = 10^{-4} \textit{ seconds (Tested in Python)}$$

$$\textit{Collision Bit Strength} = \frac{n}{2} = \frac{32 \textit{ bits}}{2}$$

$$\textit{Bit Strength} = 16 \textit{ bits}$$

$$\textit{Average number of computations to find a collision} = 2^{\textit{Bit Strength}}$$

$$\textit{Average number of computed MACs to find a collision} = 2^{16} = 65{,}536$$

$$\frac{\textit{Total MACs}}{10{,}000 \textit{ MACs in } 1 \textit{ Second}} > \textit{Recommended MAC Life (seconds)}$$

$$\frac{65{,}536}{10{,}000 \textit{ MACs in } 1 \textit{ Second}} > \textit{Recommended MAC Life (seconds)}$$

$$6.5 \textit{ Seconds to find a collision} > \textit{Recommended MAC Life}$$

We also built in defense in depth by using a SHA-512 key generator with a BLAKE2b MAC generator. This combination of hash functions, OOB initial key material system, and short-lived MAC life enable more defense for the limited bit space. Overall, the key management systems or generation ensure that the time to produce a required forgery is larger than the time in which the key is used. In summary, an extremely short lifetime was chosen for the MAC creation key to reduce the possibilities of successful MAC forgeries and increase the possibility of detecting such forgery attacks.

The short 32-bit field is particularly vulnerable to forgeries, so we included in the dialect a second derivative (D2) that generates a companion OpenFlow message of the experimenter type to carry a full-length MAC for a given OpenFlow message. A dialecting proxy handles both the 32-bit field modification for transaction-ID and experimenter type message to ensure that messages are verified as authentic before they are passed on to the intended receiver. The limiting factor with D2 is the necessity to send a 1:1 number of authentication messages as well as original messages for OpenFlow commands and data flow.

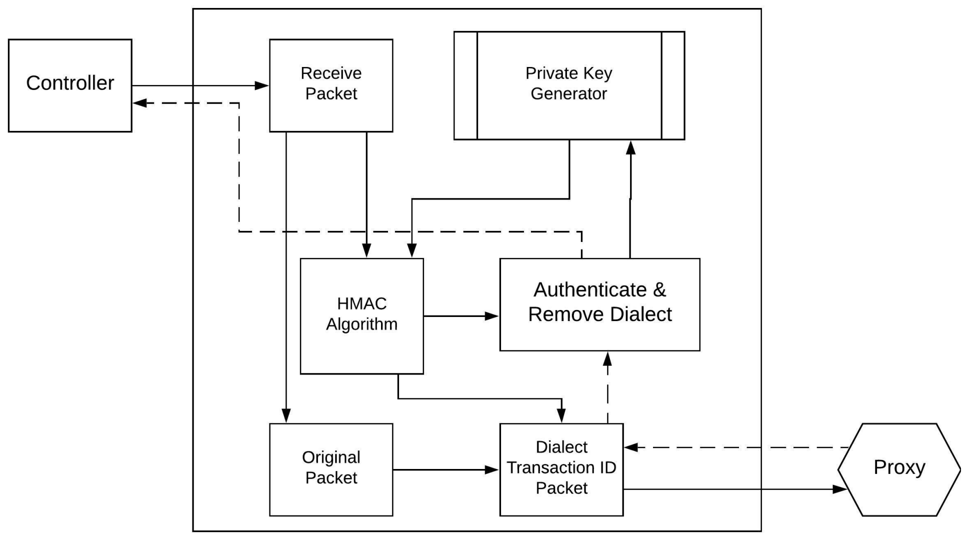
We selected a hybrid approach where D1 is used during the initial OpenFlow Hello exchange between a controller and a switch before TLS is established; then, D2 is used to generate experimenter type messages after the TLS handshake has been completed successfully. D1 adds another authentication mechanism prior to TLS. By adding another authentication layer to the system, we can reduce the possibility of an cipher-suite downgrade attack such as DROWN [15]. By also using D2 with a message dedicated to security our scheme provides greater bit strength and another layer of authentication without requiring an interface with TLS.

4.5 Security and Cost Analysis

Security analysis for this work leverages research already performed through both formal and computational methods on the underlying algorithms, and in the case, of TLS. We did not consider a separate protocol analysis for the composed security, but leave such security analysis as future work. For D1, a MAC was selected that can be set for bytes output so that existing research can be utilized and not depend upon truncation of a MAC tag. Cost analysis is the limitation imposed by the dialect. An assessment of overhead caused by dialect will be given in Chapter 5.

4.6 Implementation Method

A dialecting proxy was chosen as the primary method of testing and including a dialect into OpenFlow. It was selected because this dialect is designed for integration with a messaging system and does not rely upon electromagnetic monitoring or modification. Timing authentication was not chosen because of the potential negative impact on availability of the network. This proxy allows for modification without affecting the rest of the system and can be used with various types of derivatives. Specifically, for OpenFlow and Mininet, this design allows for rapid prototyping and modification without any changes being required to the switch or controller software. The overall design of the dialecting proxy in Figure 4.6.



The solid line represents outbound traffic. The dashed line represents inbound traffic.

Figure 4.6. Dialect Proxy Design.

Figure 4.6 demonstrates the steps the dialecting proxy will take to receive, process, dialect, authenticate, and transmit a message. There is a dialecting proxy attached to each controller or switch of the network to perform MAC checks before accepting incoming OF messages. In this experiment, the OpenFlow packet will be received by the first module. Private-key generation is also completed by the proxy or delivered OOB. The HMAC algorithm is stored on the proxy as well so that the digest can be added to the original OpenFlow message. The digest is created from the original OpenFlow packet minus the transaction ID.

Binary modification was selected as the backup implementation method due to the complexity of code for the switch and controller system already included in Mininet. Ongoing research in the field of binary modification is orthogonal to this research and will allow for future modification without requiring extensive knowledge of a system or programming. A potential security benefit to the inclusion of a dialect at the binary level is the eliminated risk of subversion of an external device. Binary modification of a standard controller could also prevent an attacker from being able to bypass the security system from the network.

4.7 Dialect Management

Dialect management is concerned with the efforts required by system administrators and operators to maintain and include a protocol dialect within a protocol. The objective of dialect management is to maintain a high ratio of return on security with minimal operator effort. The effort required for the implementation of this dialect is reduced by the selection of an appropriate implementation method and key management system. While we selected a dialecting proxy design and OOB key management, we considered various types of key management schemes in Section 3.4.3. The types of key management schemes considered included IBC, PKI, ABC, and OOB. The selection of an OOB key delivery management system reduces the required setup time and management of a trusted third party (TTP) entity that is separate from the SDN.

OOB consists of the delivery of secret keys outside of traffic that already exist on the network. For this research, we have chosen to start with OOB for simplicity and to provide flexibility in the types and system requirements for including the dialect. The negative aspects to selecting this type of key management system include automation and scalability. The second choice for this research was PKI because it has already been designed as the key management system for TLS [3]. PKI can be modified for use with a dialect instead of just relying upon TLS. For example, by using the pre-shared keys from a PKI system in addition to different keys and security systems for the dialect we are able to add security benefits. Even though such a dialect may not achieve future secrecy (FS) flows before the TLS set-up, the use of a modified PKI system and not just TLS presents the opportunity to add more security layers.

Another option for key management included IBC. An IBC operates by associating "between identities, public keys, and validity" [27]. Inclusion of IBC would remove the requirement for public keys because identity features would be used for encryption [27]. In comparison to PKI, ABC incorporates attributes of a recipient to determine part of the security scheme [21]. ABC, for the future, contains desirable qualities for an SDN because the controller could have identity features of installed network components preregistered in memory. This rapid update of private keys would be produced by a private-key generator (PKG) [27]. The trust relationship, PKG location, and time to update are limiting factors for inclusion in an SDN due to the concern for availability. While there are a variety of types of PKI and ABC that do not always require a TTP, the flexibility of dialect design allows for the introduction of different methods to perform key management. For this research, OOB provided simplicity and flexibility for testing.

4.8 Implementation Testing

Implementation testing ensures that the dialect will not negatively affect network performance beyond a given threshold. For our research, we measured the increase in latency incurred by derivatives to understand the performance trade-off. This percentage is chosen because it is deemed that the security benefits justify the network delay due to the security goals chosen. The goal of preventing non-authentic packets from causing unauthorized network changes requires the rebalancing of priorities of integrity and availability. Ideally, the increase in latency should be much less than that incurred by TLS, which has a much wider range of security goals than the derivatives. Implementation testing for this research requires the establishment of baseline performance data of the SDN before dialecting the protocol and performance after implementation. Once the dialect has been installed and is operational, it is run under normal network conditions to ensure an organization's performance requirements are met. Lastly, due to the inclusion of the protocol dialect in the messaging system and base communication protocol of OpenFlow, implementation testing requires verification that no change to the OpenFlow standard occurs.

CHAPTER 5: Experiment and Results

The experimentation for this research includes Python script prototyping while leveraging open-source SDN emulation software. The dialect presented here utilizes three different derivatives to add security. The first derivative modifies OpenFlow Hello messages transaction IDs to include a MAC before establishing TLS. The second derivative creates an experimenter type message which is used to authenticate messages once TLS has been established. The third derivative wraps TLS and OpenFlow messages to add another security layer without requiring access to TLS keys. This chapter details the process to create these dialects and evaluate them using the SDN emulation software.

The first stage of experimentation consisted of testing outside of the SDN emulation software (Mininet) to perform rapid code development using Python libraries. The second stage of the experiment incorporated Python scripts and Mininet. Lastly, TLS was enabled with Mininet and Python scripts for the proxies. Programming and testing in Python first allowed for dialect creation, input/output management, and logic before integration with Mininet. The design process from Chapter 3 and options from Chapter 4 are included in this experiment as a proof of concept. An example of the message-handling process performed inside of a pair of proxies attached to a controller and switch for derivatives is shown in Figure 5.1:

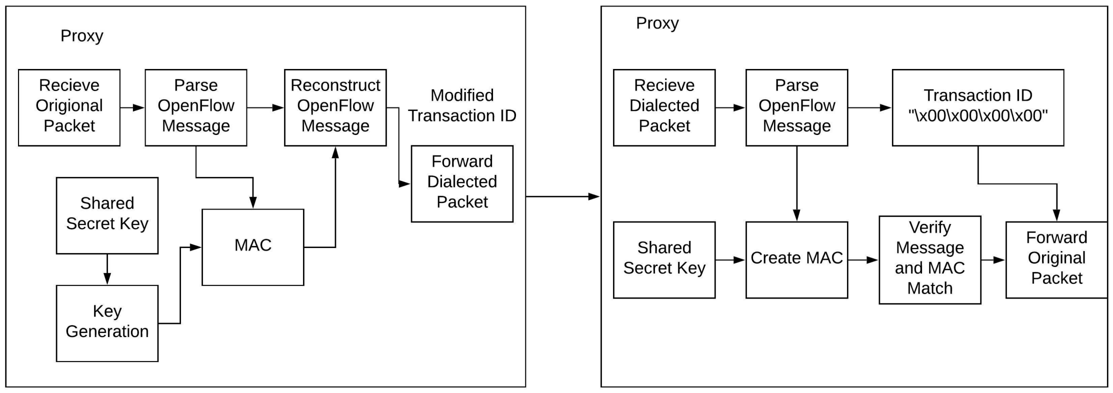


Figure 5.1. Dialect Proxy Software.

Figure 5.1 demonstrates two key functions performed by the dialecting proxy: (1) a MAC creation function that intercepts every outbound OpenFlow packet, adds a MAC inside of the OpenFlow header (D1), creates an additional experimenter packet (D2), or wraps contents (Derivative 3 (D3)) for verification before forwarding the original packet to the intended destination; (2) a MAC verification function that authenticates that data sent by the other proxy. Therefore, Figure 5.1 shows one-way of the two-way authentication. The process for both proxies in a system requires the receipt of a message, parsing for applicable contents, creation of a MAC tag, message reconstruction, and forwarding to the next device in the communication chain. In this experiment, if a message fails the authentication check then the receiving proxy will discontinue the set up process. The process of passing command data between controller and switch is identified in Figure 3.7.

5.1 Experiment

Mininet Network Emulator was selected as the virtual software for this experiment [12]. This emulator system generates and processes actual OpenFlow packets at bit level and supports open-source SDN controller and switch software systems that are widely used. Initial experimentation was performed using Python only to test different security algorithms and processes that the proxy uses. Once the security derivatives were designed and tested outside of the emulator, they were then integrated into Mininet. This system was selected because of the quick ability to prototype and perform packet analysis all on one system. The second part of this experiment is tailored for use with Mininet and analysis using free software such as Wireshark [13] for packet sniffing. Wireshark allowed the detection and parsing of packets in the emulated system to observe and time baseline operation, and to verify and time dialect implementation. The experiment topology may be found in Figure 5.2.

This topology utilized our selected SDN emulator, Python scripts for the proxies, and the POX open-source controller on one machine. The switch and controller for the SDN run the same software that is used in physical systems; however, the physical ports have been reconfigured to operate on a single machine. In this configuration, the Open vSwitch (OVS) switch is assigned to an external controller IP address which is the IP address of the Switch Proxy. In this design, both the controller and switch lack knowledge of the proxy. This topology requires no modification of the switch or controller software before installation of the dialect. The proxies then coordinate their communication. Instead of one client-

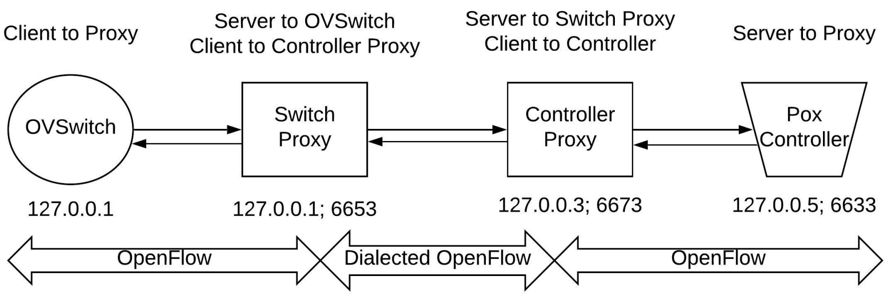


Figure 5.2. Experiment Topology.

server relationship, there are now three. This relationship and topology allows for different versions of OpenFlow to be used between a controller and switch via the proxy system. It is anticipated for this research that the attack vector would be between the two proxies.

The first experiments utilized Mininet without TLS enabled so that each packet could be observed between the proxies while using D1 and D2. The second portion of the experiment utilized TLS and required a different controller. The controller utilized with TLS enabled was the ovs-testcontroller and a TLS procedure for Mininet [23]. The layout for this experiment was the same as Figure 5.2.

5.1.1 Derivative 1: OpenFlow Hello Transaction ID Modification

D1 modifies the transaction ID field in OpenFlow Hello messages to include a MAC tag to add authentication to the initial message exchange between a SDN controller and its switches. This MAC tag is verified at a receiving proxy before the message is passed to the intended destination. The pre-TLS key is created from a HKDF using SHA-512 and an OOB IKM. This pre-TLS key will be 512 bits long and is used by the MAC generator algorithm, BLAKE2b, to create a MAC designed to have an adjustable output. It is adjusted to a compressed size of 32 bits to fit in the transaction ID field of OpenFlow headers. A new key is generated every 5 seconds to improve security of the MAC tag. A summary of the security mechanisms, algorithms, and design choices for this derivative included in the script may be found below.

The creation of OpenFlow messages using the open-source *python-openflow* library was customized to show the binary position of the transaction ID bytes. The byte "x08" was

Table 5.1. Derivative Design Selection.

Dialect Derivative Security Feature	***Selected Option***
Key Generator	HKDF - SHA512
Pre-TLS Key	512-bits
Key Regeneration Period	5 Seconds
MAC Generator	BLAKE2b

identified as the last portion of the OpenFlow header; therefore, the remaining bytes are removed since they are part of the transaction ID. This *python-openflow* library also allowed for identification of the binary differences between messages. Overall, the format for the OpenFlow Hello message after modification with D1 has the layout shown in Figure 5.3.

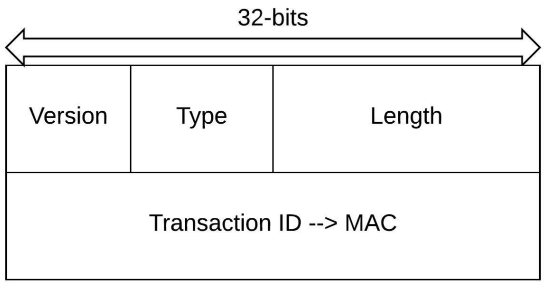


Figure 5.3. Derivative 1 Message Layout. Adapted from [24].

While many types of messages demonstrate this same byte pattern, the initial focus of this research and byte modification was OpenFlow Hello. This message type was chosen for a proof of concept because it is sent as the first means of establishing communication and version negotiation between an SDN controller and switch [3]. This is the first type of message sent before any TLS session is established. Later derivative designs allow for other tools such as larger security bit space, key management options, and TLS protection. D1 stands as an example of Protocol Dialect that exists within the bounds of the protocol. However, this restrictive space is not a replacement for TLS and should be combined with other derivative options that provide greater bit-level security and options.

5.1.2 Derivative 2: Experimenter Type Message

D2 utilizes a larger bit space to ensure security of another OpenFlow message by transporting a MAC and an associated transaction ID number. The experimenter message is defined in the OpenFlow switch specification [3]. The components of the second derivative consist of the OpenFlow Header, Experiment ID, Experiment Type, and variable-length data field. The OpenFlow header contains the type of message, OpenFlow version, and length fields as required by the OpenFlow Switch Specification standard [3]. Ideally, if dialect software was located on a switch or controller, then both D1 and D2 could be used to provide more security for each OpenFlow message. To ensure that the security benefits of both are enforced, the derivatives should be linked. The overall timing and execution of using both an experimenter type message and OpenFlow message may be found in Figure 5.4.

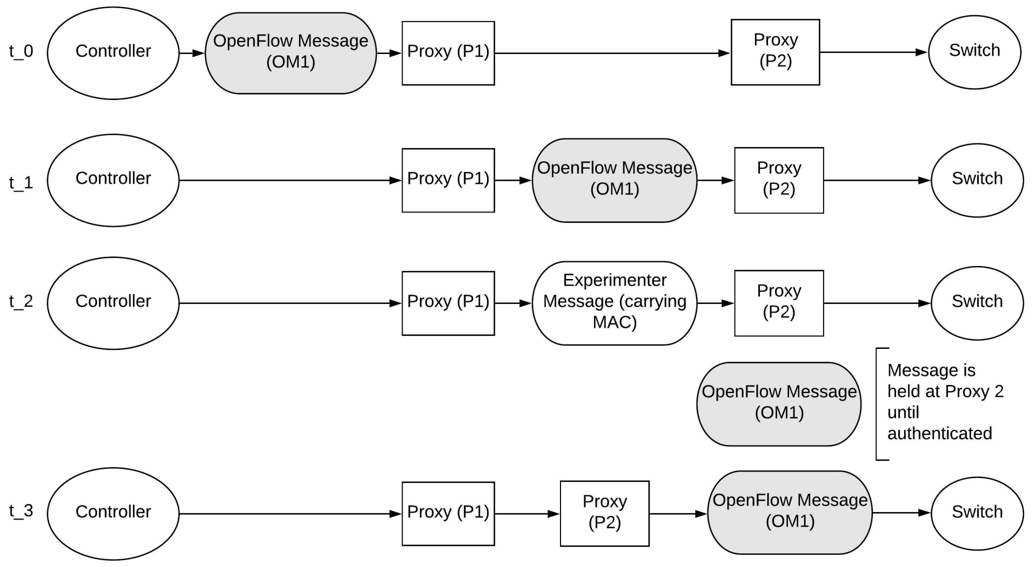


Figure 5.4. Derivative 2 Timing and Execution.

Each field of the experimenter type message has been tailored for use by D2. The D2 message is not sent by itself, but instead is paired with another OpenFlow message that it verifies by containing the MAC for both the OpenFlow command and experimenter type message. The OpenFlow header also matches the requirement of a normal experimenter type message. We selected an arbitrary experiment ID and type values for testing because they are only utilized between proxies. The arbitrary values may confuse packet-monitoring software such as Wireshark, but should be set to a value different than the OpenFlow standard-approved experiment IDs. The experiment type message has been set to a value to represent Naval Postgraduate School (NPS) for testing.

```
Experiment_ID = b"\x01\x01\x01\x01"
Experiment_type_nps = b"\x02\x02\x02\x02"
```

The experiment ID does not need to be approved by the OpenFlow Foundation before system integration because it only interacts with the dialecting proxies, not the SDN controller or switch. The overall layout of the message is shown in Figure 5.5.

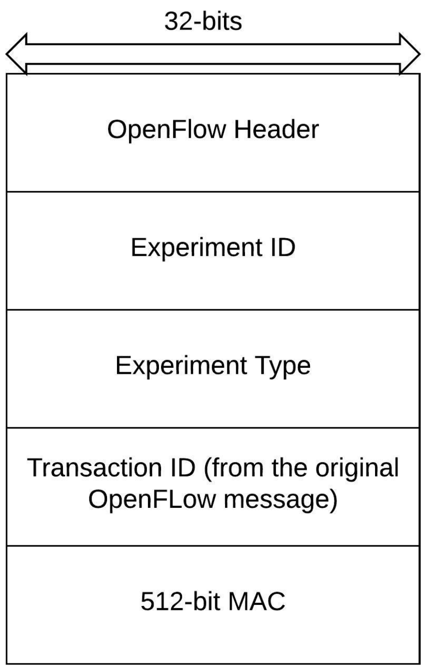


Figure 5.5. Derivative 2 Message Layout.

In future work the above message type could be used for key exchange, protocol dialect updates, and other management features, depending on design choice of incorporation of such features. This design was created with the purpose of compliance with the requirements of the OpenFlow standard [3] even though it is only sent between proxies. The D2 message type abides by the requirements of the OpenFlow standard since it can be tailored to fit an organization's requirements and could be approved by the OpenFlow Foundation in the future. Yet, the new message type does not need to be approved by the OpenFlow Foundation to work with an existing SDN system because it is only sent between the proxies in this implementation.

5.1.3 Derivative 3: TLS and OpenFlow Derivative

Due to the implementation method that was chosen for this research, another derivative was designed for DiD consisting of both TLS and OpenFlow. This derivative is wrapped externally to both TLS and OpenFlow data. The derivative starts after the first OpenFlow Hello message with D1 and is sent for every TLS and OpenFlow message following. If authentication does not pass during this step then the connection is broken. The proxies, in this experiment, do not have access to TLS keys, and therefore, D2 could not function as designed. The separation of keys also improves DiD by reducing the damage to security if the proxy or the SDN devices are compromised. To add protection to TLS during the handshake and operation we added a wrapper that used the same BLAKE2b MAC. D3 appends a 512-bit MAC to every message, verified at the next proxy, and is removed before being sent to the corresponding device. This additional security is only provided between proxies, each of which is assumed to be colocated with its communicating device. The layout of this message and process may be found in Figure 5.6.

5.1.4 TLS and Mininet

TLS is off by default for Mininet. Therefore, part of this experiment required enabling TLS for both a switch and controller. Only a few online sources provide a recommended method to enable TLS [23]. The following steps to setup TLS are updated for the newest version of Mininet at the time of this writing and provide additional instructions. Establishing TLS in Mininet requires that keys are created and registered with an Open vSwitch and ovs-controller [23]. The only other changes to the Mininet TLS procedure for this research was the modification of the IP address and port number for the remote controller [23]. The addresses and port numbers assigned with TLS matched those in Figure 5.2. The programs that were utilized to create the TLS enabled OVS, start the controller with TLS, and associated proxies may be found in Appendix A.

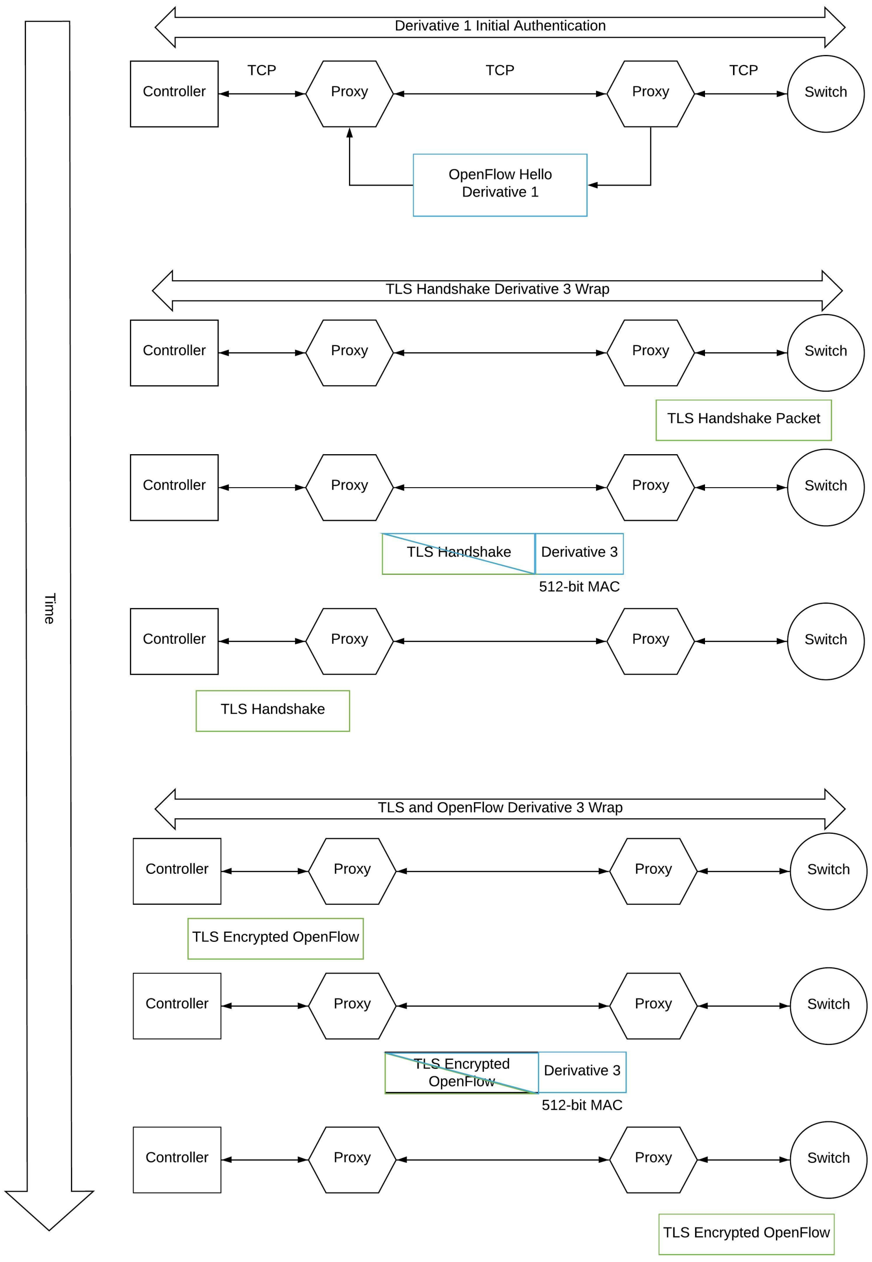


Figure 5.6. Derivative 1 and 3 with TLS.

5.2 Results

The results from this experiment are separated into two main sections: those consisting of results with and without TLS enabled. However, before these results were obtained, control tests were performed. The control tests for this experiment consisted of running

Mininet with and without TLS enabled with no proxies. The first experiment with the dialect included in the system via proxies containing D1 and D2 with TLS disabled. The second portion of the results were obtained with TLS enabled and proxies containing D1 and D3. Results from the experiment with TLS turned off were obtained using Wireshark. For the second set of results with TLS enabled results were obtained from terminal output from the ovs-testcontroller. Wireshark was not used for the second portion of the experiment because of the encryption of data provided by TLS.

An important note about these results and associated time overhead from the dialect is that Python was used as the language for the proxies. This means that any associated delays due to being a higher-level language added time to the results. We anticipate that the results may be significantly faster in lower-level languages; therefore, reducing the overhead identified in this experiment. Section 4.4 notes the speed considerations with regards to security and research associated with Blake2. Our selection of proxies also added time for this derivative. The use of two new TCP/IP connections adds more time to the overhead compared to binary modification of the switch or controller directly.

5.2.1 Results without TLS enabled

The following results include data from normal Mininet operation, the addition of derivatives 1-3, and the enabling of TLS. The baseline data was collected using Mininet without any dialect software to obtain baseline data on the timing requirements for OVS and controllers in an emulated network. Follow-on results were found once a proxy was established between a POX controller and OVS, and lastly after each derivative was included into the network one at a time to assess the time added due to each derivative. All of these initial results were collected without TLS. Timing without TLS was tested first to determine the ability to include a dialect and confirm its operation with Wireshark.

Each experiment was measured twenty times to determine a minimum, average, and maximum time required to perform the setup between a POX controller and OVS in Mininet. The time was measured from the time of the first OpenFlow Hello message to the first OpenFlow Echo Request message. The OpenFlow Hello message signifies the beginning of communication between a switch and controller, while the echo request type message signifies that setup of the switch has been completed by the controller. Therefore, this overall time difference records the time to setup a new switch that is initialized with a controller.

Table 5.2. Derivative Time Performance.

Time Without Proxies		Time with Derivative 1 & 2	
Min	5.00	Min	5.00
Max	5.011	Max	6.21
Avg	5.00	Avg	5.52
All Time Is In Seconds		Average Percent Increase	10.28%

Testing indicated that the total time increase for the setup portion of the OpenFlow communication was 10.28% for a proxy that provides feedback to a user. However, it is not expected that this increase would cause a major delay to operations because this transfer occurs before any flow rule changes or packet-in requests. Once D1 and D2 measurements were complete we analyzed methods that could be used to reduce the overhead identified with the current proxy design. Therefore, for the second version of the proxy only contained scripts for D1 and D2 as well as the TCP connections. This optimized version of the proxies contains no print statements and reduced the overhead significantly. The results for the optimized version of the proxies and D1 and D2 may be found in Table 5.3.

Table 5.3. Optimized Proxy Time Performance.

Time To Perform Derivative 1 & 2 (Optimized)	
Min	5.00
Max	5.042
Avg	5.0071
Average Percent Increase	0.041%

The results for the optimized proxy demonstrate that the overall time added to perform the dialect during startup is less than 1%. This percent increase was also calculated against the normal Mininet operation and averaged from 30 runs for each experiment. The greatest contribution to time for these runs was the time to perform key generation and MAC generation using hash algorithms. This simplified and optimized proxy has a minimal performance impact on normal operations on a Mininet and imposes minimal delay in messaging.

5.2.2 Results with TLS enabled

An experiment was also performed with the dialect and TLS enabled. The following results were collected using the same Mininet platform; however, a different controller and proxy were required to establish and maintain communications. Therefore, the results for this section were first performed without proxies to establish a baseline. Once a baseline was established, it was observed that TLS took about twice as long to establish a setup between a controller and switch. While this controller was different from the POX controller used in the first set of experiments, we assume that the majority of time added was due to TLS and not the controller design itself. Once TLS was checked between the devices, the proxies with both D1 and D3 were added. Without TLS, the proxies were able to print and identify OpenFlow packets sent between devices. With TLS enabled this meant that the status of messages sent and received was reported by the controller before and after decryption. The results for testing from runs with TLS for average, maximum, and minimum may be found in Table 5.4.

Table 5.4. TLS Time Performance.

Time Without Proxies		Derivative 1 & 3 (Optimized)	
Min	8.00	Min	9.00
Max	11	Max	15.00
Avg	9.33	Avg	11.37
All Time Is In Seconds		Average Percent Increase	21.86%

We determined from the 30 runs that the average percent increase in overhead due to TLS being enabled to the system was 86%. We then measured the time added due to adding proxies with D1 and D3 for the set up between a controller and switch was 22% with OOB pre-shared long-term derivative keys. This higher percentage delay accounts for the larger data and digest being processed by BLAKE2b. These results identify the estimated additional overhead of the derivative with TLS, but do not include the potential time to perform key management of the dialect. These results include the time that it takes to add D3 to TLS handshake messages from OpenFlow Hello to OpenFlow echo request. The time to perform normal messages between the controller and switches showed no additional time delay at the controller or switch. The time interval that OpenFlow echo requests and echo reply are sent are every 5 seconds and they were sent and received within one second with both TLS and D3.

CHAPTER 6: Conclusion and Future Work

This chapter details limitations associated with the MS1 dialect design and the evaluation experiment, conclusions from the overall thesis research, and recommended future work. The main results provided from this research include: (i) formulation of a systematic methodology for dialecting network protocols, (ii) the selected design and implementation for an OpenFlow dialect, and (iii) the time delay associated with including MS1 into OpenFlow using an emulator.

6.1 Limitations

The limitations for this experiment included the emulation of an environment compared to experiment on physical components, proxy design to include error handling, formal security analysis, and timing. One limitation inherent to starting the experiment is timing. The key generators are synchronized using time as a salt. This means that the proxies must be started within a second of each other or else MAC verification failures will occur, even in the absence of an attacker. The limitation on synchronization of starting the proxies was created to not require pre-shared salts between the proxies. For this specific implementation, the life of a MAC was limited to 5 seconds. The decision for a 5-second MAC life may be found in Section 4.4. The current proxy logic design supports a time interval down to 2 seconds before failing to authenticate any messages. At a MAC life of less than 2 seconds, message authentication verification failed and the connections were broken by the proxies. Further logic required to handle a one-key-per-message creation and validation was not built into the proxies.

Another limitation associated with this research is the quantification of the security benefits of protocol dialection. Formal and computational security analysis was not completed for dialect MS1. For example, the time and resources required to break the dialected protocol was not determined experimentally. Formal analysis is required to determine any potential dangers associated with using two different security systems (dialect and TLS) for OpenFlow. Each security system contained different keys; however, the keys for both derivatives were the same. Security analysis is required to determine any risks or vulnerabilities associated with use of the same keys for each derivative or if different keys are required. This analysis is required to determine the extent to which the security objectives have been met by the addition of one or more derivatives. Individual derivative analysis is also necessary to verify that each derivative should be added to a dialect.

A third limitation associated with this experiment type is that we did not perform these same tests on physical components and instead only performed operations on a single machine. A limitation in regards to robustness and fault tolerance is that the proxy design for this experiment was basic and designed primarily for allowing the inclusion of the security modules by decoupling and intercepting the communication path of a switch and controller. This proxy does not handle all cases and conditions between a switch and controller that would be required for all operations. The proxies do not recover a dropped connection automatically, which could lead to a denial-of-service vulnerability. Therefore, exception handling is one limitation associated with this basic SDN proxy.

During the set up process, the chosen OpenFlow message parsing package exhibited a limitation during proxy operation. Based on the OpenFlow message parser used in this experiment, two out of ten messages for set up could not be dialected due they were too large. Both large messages or small messages (near zero) can disrupt proxy operation due to the current OpenFlow message parsing package. The package that exhibited this limitation is the *Kytos* package message parser. The messages that caused failure of the parser were excluded from the derivative process for D1 and D2. Further investigation of D2 is needed. This limitation is not applicable to D1 and D3 because D1 is a set length and D3 operates as a wrapper that does not analyze the internal contents of the OpenFlow packet.

6.2 Conclusion

This research provides a framework for the creation of a protocol dialect, a novel authentication defense layer that can be used directly in OpenFlow, a novel method to protect TLS handshake authentication, implementation of a dialect, and time measurements of the proposed security measures in the Mininet testing environment. The framework provided in Chapter 3 provides a process to add information security to existing protocols. The experiment for this research demonstrates an example of this process with OpenFlow as a proof-of-concept. It was determined during implementation that this process also allows for authentication before TLS is enabled. In this case, the designed protocol dialect provides enforced authentication before a TLS handshake is allowed via a proxy system and during the handshake with D3. This design provides another layer of defense against an attacker from performing some attacks on TLS such as a cipher-suite downgrade attack.

This dialect can provide a method of authentication on the very first OpenFlow packet that is sent and on each following packet with few exceptions. Previous exceptions were mentioned in Chapter 5 limitations. The proxies for this research allowed for one controller (external to Mininet) and one switch to setup and maintain connectivity. The results from testing demonstrated that the overhead assigned to a one-controller, one-switch, and two-proxy system was approximately 10% or less. In the event that an optimized proxy is used, that delay was reduced to less than 1 percent. With TLS enabled, less than 22% overhead was added. This higher percentage with TLS accounts for a larger message due to the addition of encryption.

This system demonstrates that dialect creation and verification should be scalable based on the types of algorithms chosen. It is anticipated that using lower level languages in the future would reduce the overhead identified in this experiment. The selection of a proxy method of implementation also added time to our results because two additional TCP/IP connections were made compared to the original SDN for this experiment. Therefore, the results identified here are expected to be near worst-case due to the additional time to perform Python functions.

This proof-of-concept demonstrates that it is possible to incorporate a dialect into an existing protocol in OpenFlow. With verification through experimentation, the OpenFlow protocol is a viable candidate for dialects and in this case the addition of a MAC into the protocol header. Using the experimenter message format, the OpenFlow protocol also allows for the introduction of new key management methods, dialect management, and authentication systems using a custom-tailored message format that still meets the OpenFlow protocol standard. This research also demonstrated that it is possible to create proxies that can operate within a Mininet environment.

D1 provides 16-bits of security and a 5-second MAC lifespan to add an initial layer of authentication before any follow-on handshake (with or without TLS) is allowed. The MAC life is extremely limited to reduce the ability of an attacker from being able to determine the

OOB long-term keys used in the system. D1 is only used once per conversation. Unique keys are generated every 5 seconds which improves upon the 16-bits of initial security by D1. As another method to improve security D2, the variable data field size has allowed for 512-bits to be added which can be used to authenticate the derivative message as well as a corresponding OpenFlow message that is referenced by its transaction ID. Lastly, D3 is provided as an external wrapper to protect OpenFlow and TLS as an additional security layer. This wrapper is utilized during and after the TLS handshake by encapsulating all TLS and OpenFlow data for additional authentication.

The addition of TLS to this experiment was limited due to the proxy configuration. Running TLS from the controller to switch would not allow for modification by the proxy, therefore the only method to include TLS would be by creating a TLS connection from proxy to proxy for this implementation. The ability to run both a dialect and TLS should improve information security for the command and control channel of an SDN. It is essential that the derivatives and TLS utilize different keys to ensure that vulnerabilities are not added to TLS via the addition of a dialect.

Protocol dialects present challenges in both design, implementation, and analysis. A protocol dialect takes time to design, time to implement, additional resources, and potentially more complex security analysis. Time is added in both design, implementation, and use when a dialect is added to a protocol. Protocol dialect design can also be reduced by future standardization which is only modified by the use of different keys or tailored algorithm settings. The time overhead associated with the addition of a dialect can be reduced by an appropriate selection of algorithms and programming language. The memory and resources required to add a dialect can also be reduced by the selection of algorithms and key management systems as well as the location of where the dialect is installed (binary vs proxy).

6.3 Future Work

Further research into this field requires integration with physical machines, such as those used in production SDNs. The integration with an SDN could be performed once a physical proxy is designed and placed in-line with an SDN controller and switch. The inclusion within a physical system will allow for portability, testing, and verification that security and performance requirements for an organization are met. Security analysis is also required to assess the extent that security is improved by incorporating a dialect in an existing protocol. This security analysis should also ensure that there is no vulnerability added to TLS by the addition of the dialect. Future work should include further exception handling by both the derivative software and proxy software. It would also include creating a multiple proxy system that had an assigned proxy for each switch and utilizes a many switch system to test scalability. Open-source protocols and the dialecting process enables additional layers of security to be built into base communication network protocols.

Improvements and optimization of the functions and techniques used inside of the proxies is recommended. One such example would be randomized salts and key management design so that OOB is not the primary key management method. The current proxy design utilizes time as the salt synchronization between proxies, but could be improved with salts that are shared via D2 or randomized. D2 also has the potential use for key management in later implementations that allow for normal or emergency key updates for the dialect. D2 also could provide another method to improve synchronization in the event that a disconnect occurs for fault tolerance or availability attack mitigation.

Based on the derivative overhead determined from this experiment, it may be desirable to combine all three derivatives designed here to be in one dialect. If the derivative programs were included into the software for a controller and switch and had access to the internal data of OpenFlow messages, then all three derivatives could be utilized depending upon the security analysis results for the intended network. It is important to note that limiting additional security risks to TLS should be prioritized and considered when adding dialect, especially for an SDN. D1 is utilized as an initial layer of defense to prevent unauthorized connections from attempting to setup or exchange TLS key set-up information. The second phase of communication would add security through the incorporation of D3 for the TLS handshake and all follow on messages as a wrapper. The last phase of communication would be D2 for dialect management such as key updates.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Source Code Overview

This appendix contains the required commands, scripts, and software required to perform the experiments in this thesis. A virtual machine (VM) is provided which contains all of the software both experiments. As another option, the scripts for the proxies, controllers, and switches are provided in Appendices B-F. The proxies were developed utilizing the following references during code development [28] [29] [30] [31] [32] [33] [34].

Protocol Dialect Experiment Virtual Machine:

An Ubuntu VM in .ova format is available at the following link containing all software used in this thesis. Software included inside of the experiment VM includes Mininet 2.2.2, Python 3.6.8, and the POX controller. The password for the VM is "default".

```
https://drive.google.com/file/d/16zsDmageep6gMjQ946PswjtD1IBjNi6q/view?usp=
 sharing
```

Once the .ova file has been downloaded and imported to a Virtual Machine software the following steps are required to repeat the experiments performed in this thesis.

To perform the TLS disabled experiment:

- Controller Command (POX Controller) (**Appendix B**)
- Mininet Command (**Appendix B**)
- Controller Proxy Synchronous Start Up (**Appendix C**)
- Switch Proxy Synchronous Start Up (**Appendix D**)

To perform the TLS enabled experiment:

- Controller Command (ovs-testcontroller) (**Appendix B**)
- Mininet Command (**Appendix B**)
- Controller Proxy Synchronous Start Up(**Appendix E**)
- Switch Proxy Synchronous Start Up (**Appendix F**)

Proxies and Dialect without TLS:

To run the experiment with TLS disabled (Mininet by default) requires Mininet, POX controller, and both proxies. These proxies contain D1 and D2. The first script is ran as a proxy for the controller in an SDN. The second script is ran as the proxy for a switch in an SDN. The optimized version of these scripts can be ran by commenting out the print statements provided by this script.

- **Derivative 1 and 2 Controller Proxy** (**Appendix C**): This program is the proxy that operates with the POX controller. It serves as a client to the POX controller which operates as a server. This controller proxy contains D1 and D2 and is started after the POX controller has been started. The controller proxy program must run before the proxy for the switch because the controller proxy acts as a server for the other proxy.
- **Derivative 1 and 2 Switch Proxy** (**Appendix D**): The following proxy script contains D1 and D2. The proxy connects to the OVS first as a server in a client-server relationship. This proxy then connects to the other proxy as a client. D1 and D2 in this proxy allow for OpenFlow messages to have additional authentication security as well as an initial authentication check with D1.

Proxies and Dialect with TLS:

To run the experiment with TLS enabled two different proxies, a TLS enabled OVS, and an ovs-testcontroller. These proxies contain D1 and D3. The proxies provided have print statements enable to show the results and messages that are being process. While TLS is enabled using the following programs the proxies are unable to read the TLS encrypted packet. The optimized versions of these scripts require commenting out the print statements provided with each proxy. The ovs-testcontroller debug statements are the primary method to verify and observe traffic that is being sent in the emulated network.

- **Derivative 1 and 3 Controller Proxy** (**Appendix E**): This proxy operates with the ovs-testcontroller. It serves as a client to the ovs-testcontroller which operates as a server and contains D1 and D3. It is designed to not read TLS packets. This script is started after the ovs-testcontroller has been started and the Mininet environment. This program must run before the proxy for the switch because the controller proxy acts as a server for the other proxy.

- **Derivative 1 and 3 Switch Proxy** (**Appendix F**): This proxy script contains D1 and D3. The proxy connects to the OVS first as a server in a client-server relationship. This proxy then connects to the other proxy as a client. D1 and D3 in this proxy allow for OpenFlow and TLS messages to have additional authentication security.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
SDN Controller Commands and Switch Setup

TLS Disabled Controller Set up:

To start the emulation system in a method that support the addition of the proxies requires a specific set of command to start the switch and controller. Both of these commands must be performed before the above proxy scripts are started. The command to start the external (outside of Mini-Net) POX controller consists of the OpenFlow version to support, an assigned IP address, and port number for OpenFlow packets. The command to start the POX controller must be performed inside of the folder containing POX.

```
./pox.py openflow.of_01 --address=127.0.0.5 --port=6633
```

TLS Disabled Emulator Command:

Once the POX controller has been started the Mini-Net emulation system must be started with the specifications of an IP address for the switch proxy (appears as controller) and the type of switch to use for the emulator. The command to start Mini-Net with an external controller is as follows:

```
sudo mn --controller=remote,127.0.0.1 --mac -i 10.1.1.0/24 --switch=ovsk
```

TLS Enabled Environment Controller Command:

The following command is required to start an ovs-testcontroller with TLS enabled. The certificates required by this command must be created first in accordance with [23]. This command must also be performed before sslexp.py. In the process of establishing the connection the ovs-testcontroller is are started first, then the Mini-Net emulator, and finally the proxies. This means that the controller and switch will not communicate until the proxies are started.

```
sudo ovs-testcontroller -v pssl:6633:127.0.0.5 \

- p /etc/openvswich/ctl-privkey.pem -c /etc/openvswitch/ctl-cert.pem \

- C /var/lib/openvswitch/pki/switchca/cacert.pem
```

TLS Emulator Script sslexp.py:

The following script is required to run the TLS enabled Mini-Net environment. This program starts the Mini-Net environment without a controller. The controller must be started first in another terminal window before this program is started. This script has been modified from [23]. The IP addresses and port numbers have been modified to meet the design of Figure 5.2.

```
#File: ssl_exp.py
#Name: Michael Sjoholm-Sierchio
#Modified from https://github.com/mininet/mininet/wiki/SSL-on-Open-vSwitch-a
#nd-ovs-controller

from mininet.net import Mininet
from mininet.node import Controller, RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def emptyNet():
    net = Mininet( controller=RemoteController )
    net.addController( 'c0' )
    h1 = net.addHost( 'h1' )
    h2 = net.addHost( 'h2' )
    s1 = net.addSwitch( 's1' )
    net.addLink( h1, s1 )
    net.addLink( h2, s1 )

    net.start()
    s1.cmd('ovs-vsctl set-controller s1 ssl:127.0.0.1:6653')
    CLI( net )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    emptyNet()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: D1&D2 without TLS Controller Proxy

```
#Michael Sjoholm-Sierchio
#File: D1&D2_withouttls_ControllerProxy.py
#Ref: https://pymotw.com/2/socket/binary.html
#Ref: https://stackoverflow.com/questions/14043886/python
#-2-3-convert-integer-to-bytes-cleanly
# Reference:  docs.kytos.io kytos developer guide
# Reference: Working with binary data in python dev_dungeon
# Reference: Low Level OpenFlow Messages Parser used by Kytos SDN
#Platform https://kytos.io
# Reference: For secure hash library https://docs.python.org/dev/library/hashlib.html?
#highlight=s
# Reference: https://www.devdungeon.com/content/working-binary-data-python
# Reference: https://pymotw.com/3/hmac
# Reference: https://docs.python.org/3/library/hashlib.html
# Reference: https://en.wikipedia.org/wiki/HKDF

import socket
import sys
import time
import binascii
import struct
import sys
import os
import datetime
import hashlib
import hmac
import base64
import binascii
import socket
from hashlib import blake2b
from threading import Timer
from math import ceil
from pyof.foundation.base import GenericStruct
#from pyof.foundation.basic_types import UBInt8, UBInt16
from pyof.v0x01.common.utils import unpack_message
from pyof.foundation.base import GenericMessage
from pyof.v0x04.common.header import Header
from pyof.v0x01.symmetric.hello import Hello
from pyof.v0x01.controller2switch.features_request import FeaturesRequest
from pyof.v0x01.symmetric.echo_request import EchoRequest
from pyof.v0x01.symmetric.vendor_header import VendorHeader
#***************IMPORT ABOVE*******************

#_____________________________________________
#Setup the key and generation

print("\n")
print("Program is running")
print("Protocol Dialect Walkthrough Initiated")
print("-------------------------------------")
#print("Include starting Mini-Net here")
print("\n")

#********Read In Long-Term Key and Create Keys******
print("Reading in shared key")
#Obtain the shared secret key at the beginning of operation from a file or OOB

key =  b'457c311719813785096ef45f466aead3db4e535f4a7b0d06084621c0e01220a6b
43b90879fc23189d4fed6456e31529905bdc83056feda5940444893a83808bd'
#print("Here is the key from file")
##print(key)
print("Here is the length of the key")
print(len(key))
#Creat a salt, which will change every 5 seconds
standard_time = time.time()
s = round(standard_time)
```

```
salt = str(s)

def salt_time(salt):
    salt = int(salt)
    time_check = round(time.time())
    time_result = time_check - salt
    if(time_result > 5):
        s = round(time.time())
        salt = str(s)
        print("Here is the salt")
        print(salt)
    return(salt)

print("\n")
salt = salt_time(salt)

#*******************COMPLETE MAKING SALT*************

#*******************KEY GENERATOR************
data = b'597133743677397A24432646294A404E635266556A586E5A72347537782141254
42A472D4B6150645367566B59703373357638792F423F4528482B4D62516554'
hash_len = 32
length = 64
info = b""

def hmac_sha512(key, data):
    return hmac.new(key, data, hashlib.sha512).digest()

ikm = hmac_sha512(key,data)

def hkdf(length, ikm, salt, info):
    prk = hmac_sha512(salt if len(salt) > 0 else bytes([0]*hash_len), ikm)
    t = b""
    okm = b""
    for i in range(int(ceil(length / hash_len))):
        t = hmac_sha512(prk, t + info + bytes([1+i]))
        okm += t
    return okm[:length]

salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
#print("Here is the first generated key")
#print(key)
print("************************************")

#_____________________________________________________
#Derivative create and check

def derivative1_create(message, salt):
    print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    dialected_message = message

    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(dialected_message)
    message_copy = message
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The current length is:")
    print(len(mutable_bytes))
    print("The modified message without xid is:")
    print(mutable_bytes)
    total_message = mutable_bytes
    digester = blake2b(key, digest_size = 4)
    digester.update(mutable_bytes)
    mac1 = digester.hexdigest()
    print("Origional MAC Printed here")
    print(mac1)
    print("\n")
    translate_hex = bytes.fromhex(mac1)
    translate_hex = bytearray(translate_hex)
    print("Lastly, the new xid is:")
    print(translate_hex)
    print("Now to add the translate mac back to the message")
    total_message = total_message + translate_hex
    print(total_message)
    #dialected_message = message
    dialected_message = total_message
    return dialected_message

def derivative1_check(msg_recv,salt):
    #Transaction ID 0
    print("The msg_recv is:")
    print(msg_recv)
    transaction_id0 = b"\x00\x00\x00\x00"
    #checked_message = msg_recv
    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(msg_recv)
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
```

```
print("The current length is:")
print(len(mutable_bytes))
print("The modified message without xid is:")
print(mutable_bytes)

#Creating a MAC to verify that the recv message matches what is expected
print("\n")
print("The key is:")
salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
#print(key)
print("The length of key is:")
print(len(key))
digester2 = blake2b(key, digest_size = 4)
digester2.update(mutable_bytes)
mac2 = digester2.hexdigest()
print("MAC Printed here")
print(mac2)
print("\n")
new_mac = binascii.unhexlify(mac2)
new_mac = mac2
new_number = int(new_mac, 16)
print("Digest_calc is:")
digest_calc = new_number
print(digest_calc)

#Deteremine the value sent by the message
binary_msg = msg_recv
msg = unpack_message(binary_msg)
print("The type of the mesage is:")
print(msg.header.message_type)
print("The xid is:")
```

```
    print(msg.header.xid)
    value1_compare = msg.header.xid

    #Compare the two values and send to device is expected matches received
    print("Comparing values:")
    print(value1_compare)
    print(digest_calc)
    if(value1_compare == digest_calc):
        final_message = mutable_bytes + transaction_id0
    else:
        final_message = b"\x11\x11\x11\x11\x11\x11"
        #quit(1)
    checked_message = final_message
    print("The message to send on to device is:")
    print(checked_message)
    return checked_message

#_____________________________________________________
#Define functions for derivative 2

#*********Create Experimenter Message*********

def derivative2_create(message,salt):
    print("Creating derivative 2 message")

    version = b"x04"
    type_openflow = b"x04" #Experimenter type message
    length_openflow = b"x08"

    total_message = bytearray()
    openflow_header_exp = (b'\x01\x04\x08')
    print("Here is the first portion of the header")
    total_message = total_message + openflow_header_exp
    print(total_message)
    print("Setting the experiment ID")
    print(b"\x01\x01\x01\x01")
    experiment_id = b"\x01\x01\x01\x01"
    print("The new message with the temp exp id for protocol dialecting is:")
    total_message = total_message + experiment_id
    print(total_message)
    print("\n")

    #********Read in OpenFlow message to correlate xid***
    print("The experimenter type is the unique code for NPS")
    print("The unique code for NPS is")
    print(b'\x02\x02\x02\x02')
    experiment_type_nps = b"\x02\x02\x02\x02"
    print("The new total message is:")
    total_message = total_message + experiment_type_nps

    #Determine the xid of the normal OpenFlow message
    print("Here is the type of file read in at the proxy")
    binary_msg = message
    msg = unpack_message(binary_msg)
    print(msg.header.message_type)
    print(msg)
    print("\n")

    print("Here is the xid for the read in file which is used for the experiment type")
    mutable_bytes = bytearray(binary_msg)
    #print("Current length is:")
    #print(len(mutable_bytes))
    print(msg.header.xid)
    value = msg.header.xid
    value = (int(str(value)))
    value_hex = (value).to_bytes(4, byteorder="big")
    print(value_hex)
    print("The new message with the experiment type (other message transaction ID is:")
    total_message = total_message + value_hex
```

```
    print(total_message)
    print("\n")

    #**********Create Final MAC for Verification of Experimenter Message**
    print("\n")
    print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    #print(key)
    print("Creating the data portion of the message for the MAC:")
    #print("The data section with 512 bit MAC is:")
    digester4 = blake2b(key, digest_size = 64)
    digester4.update(binary_msg)
    digester4.update(total_message)
    mac2 = digester4.hexdigest()
    #print(mac2)
    translate_hex = bytes.fromhex(mac2)
    print("The translated mac is:")
    print(translate_hex)
    print("The total message is:")
    total_message = total_message + translate_hex
    print(total_message)
    derivative2_msg = total_message
    print("\n")
    return derivative2_msg


def derivative2_verify(openflow_msg, deriv2_message,salt):
    #***Verify the OpenFlow message*********************

    #It is necessary to read in both the OpenFlow Message and the Authentication Message
    print("***********************************************************")
    print("Reading in the other OpenFlow message")
    #This is the OpenFlow message that is being looked at first
    print("Here is the type of file read in at the proxy")
    msg_tocheck = unpack_message(openflow_msg)
    print(msg_tocheck.header.message_type)
    print(msg_tocheck)
    print("\n")

    print("Reading in authentication message to validate the OpenFlow message")
    data_from_proxy = deriv2_message

    print("\n")
    print("Removing the previous MAC to perform MAC check")
    mutable_bytes = bytearray(data_from_proxy)
    mutable_bytes_copy = bytearray(data_from_proxy)
    after_length = len(mutable_bytes) - 64

    j = 0
    while(True):
        del mutable_bytes_copy[j]
        if(len(mutable_bytes_copy) == 64):
            break

    while((len(mutable_bytes)) > after_length):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The modified message without MAC is:")
    print(mutable_bytes)

    print("The MAC parsed out from message is:")
    #mac_fill = hex(mac_fill)
    print(mutable_bytes_copy)

    salt = salt_time(salt)
```

```
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    print("Here is the key")
    #print(key)

    digester5 = blake2b(key, digest_size = 64)
    digester5.update(openflow_msg)
    print(openflow_msg)
    digester5.update(mutable_bytes)
    mac2 = digester5.hexdigest()
    translate_hex = bytes.fromhex(mac2)
    translate_hex = bytearray(translate_hex)
    print("\n")
    print("Generated MAC to check against Printed here")
    print(translate_hex)
    print("The MAC from the message is:")
    print(mutable_bytes_copy)

    if(translate_hex == mutable_bytes_copy):
        print("The MAC has passed the check and returning to proxy")
        return openflow_msg
    else:
        print("The MAC did not pass the check")
        return b"\x11\x11\x11\x11\x11\x11"

#_________________________________________________
#Setup network connections

#Establish the socket settings to the controller
sock = socket.socket(socket.AF_INET, socket. SOCK_STREAM)
server_name = 'localhost'
server_address = ('127.0.0.5', 6633)
sock.connect(server_address)
```

```
print("Connected to controller")

print("\n")
print("Waiting for the other proxy to connect")
#Establish the socket settings for the switch proxy
#This script is the server
sock2 = socket.socket(socket.AF_INET, socket. SOCK_STREAM)
server2_address = ('127.0.0.3', 6673)
sock2.bind(server2_address)
sock2.listen(1)
connection, client_address = sock2.accept()
print("The other proxy has connected")

#____________________________________________________________
#Receive hello from controller
data = sock.recv(32000)
receipt = 0
receipt = data
print("Controller 1 sent:", receipt)
print(type(receipt))
print("\n")

#Send OpenFlow Hello to deriivative1_create
receipt = derivative1_create(receipt, salt)

#Send dialected hello from controller to proxy
print("Sending openflow message to proxy")
print(receipt)
connection.sendall(receipt)
print("Data sent")
print("\n")

#Receive hello from proxy
data = connection.recv(32000)
receipt = 0
receipt = data
print("Switch sent:", receipt)
print(type(receipt))
print("\n")

#Check authentication of OpenFlow Hello
receipt = derivative1_check(receipt, salt)

#Send hello from proxy to controller
print("Sending data to controller")
print(receipt)
sock.sendall(receipt)
print("Data sent")
print("\n")

#Receive command from controller
data = sock.recv(32000)
receipt = 0
receipt = data
print("Controller sent:", receipt)
print(type(receipt))
print("\n")

#Send data to proxy
print("Sending data to proxy")
connection.sendall(receipt)
print("Data sent to proxy")
print("\n")

#Test Derivative 2 create
deriv2_message = derivative2_create(receipt,salt)

print("Sending experimenter to proxy")
print(deriv2_message)
connection.sendall(deriv2_message)
```

```
print("Data sent")
print("\n")

#Recieve data from proxy
data = connection.recv(32000)
receipt = 0
receipt = data
print("Switch sent:", receipt)
print(type(receipt))
print("\n")

if(len(receipt) > 70):
    if(receipt == b''):
        print("Do not check this due to size")
    else:
    #Send data to controller
        print("Sending data back to controller")
        sock.sendall(receipt)
        print("Data sent back to controller")
        print("\n")
else:
    #Receive derivative 2 from proxy
    data = connection.recv(32000)
    deriv2_message = 0
    deriv2_message = data
    print("Proxy sent derv2 message:", deriv2_message)
    print(type(deriv2_message))
    print("\n")

    receipt = derivative2_verify(receipt, deriv2_message,salt)

    if(receipt == b''):
        print("Empty message sent")
    else:
        #Send data to controller
        print("Sending data back to controller")
        sock.sendall(receipt)
        print("Data sent back to controller")
        print("\n")

while(True):
    #Recieve command from controller
    data = sock.recv(32000)
    receipt = 0
    receipt = data
    print("Controller sent:", receipt)
    print(type(receipt))
    print("\n")

    #Send data to proxy
    print("Sending data to proxy")
    connection.sendall(receipt)
    print("Data sent to proxy")
    print("\n")

    try:
        deriv2_message = derivative2_create(receipt,salt)
        print("Sending experimenter to proxy")
        print(deriv2_message)
        connection.sendall(deriv2_message)
        print("Sent experimenter to proxy")
        print("\n")
    except:
        print("Had an exception")
        print("Caused by message:")
        print(receipt)
        deriv2_message = b''

    #Recieve data from proxy
    data = connection.recv(32000)
```

```
receipt = 0
receipt = data
print("Switch sent:", receipt)
print(type(receipt))
print("\n")

if(receipt == b''):
    print("Empty message sent")
    continue
else:
    #Receive derivative 2 from proxy
    data = connection.recv(32000)
    deriv2_message = 0
    deriv2_message = data
    print("Proxy sent derv2 message:", deriv2_message)
    print(type(deriv2_message))
    print("\n")

    try:
        receipt = derivative2_verify(receipt, deriv2_message,salt)
        #Send data to controller
        print("Sending data back to controller")
        sock.sendall(receipt)
        print("Data sent back to controller")
        print("\n")
    except:
        print("Had an exception during verification")
```

APPENDIX D:
D1&D2 without TLS Switch Proxy

```
#Michael Sjoholm-Sierchio
#File: D1&D2_withouttls_SwitchProxy.py
#Ref: https://pymotw.com/2/socket/binary.html
#Ref: https://stackoverflow.com/questions/14043886/python
#-2-3-convert-integer-to-bytes-cleanly
#Ref: https://www.cyberciti.biz/faq/python-convert-string-to-int-functions
# Reference:  docs.kytos.io kytos developer guide
# Reference: Low Level OpenFlow Messages Parser used by Kytos SDN
#Platform https://kytos.io
# Reference: For secure hash library https://docs.python.org/dev/library/hashlib.html?
#highlight=s
# Reference: https://www.devdungeon.com/content/working-binary-data-python
# Reference: https://pymotw.com/3/hmac
# Reference: https://docs.python.org/3/library/hashlib.html
# Reference: https://en.wikipedia.org/wiki/HKDF

import sys
import socket
import time
import random
import binascii
import struct
import os
import datetime
import hashlib
import hmac
import base64
import binascii
import socket
from hashlib import blake2b
from threading import Timer
from math import ceil
from pyof.foundation.base import GenericStruct
from pyof.v0x01.common.utils import unpack_message
from pyof.foundation.base import GenericMessage
from pyof.v0x04.common.header import Header
from pyof.v0x01.symmetric.hello import Hello
from pyof.v0x01.controller2switch.features_request import FeaturesRequest
from pyof.v0x01.symmetric.echo_request import EchoRequest
from pyof.v0x01.symmetric.vendor_header import VendorHeader
#***************IMPORT ABOVE*******************

#____________________________________________
#Setup the key and generation

#********Read In Long-Term Key and Create Keys******
print("Reading in shared key")
#Obtain the shared secret key at the beginning of operation from a file or OOB

key =  b'457c311719813785096ef45f466aead3db4e535f4a7b0d06084621c0e01220a6b
43b90879fc23189d4fed6456e31529905bdc83056feda5940444893a83808bd'
#print("Here is the key from file")
#print(key)
print("Here is the length of the key")
print(len(key))
#Create a salt, which will change every 5 seconds
standard_time = time.time()
s = round(standard_time)
salt = str(s)

def salt_time(salt):
    salt = int(salt)
    time_check = round(time.time())
    time_result = time_check - salt
    if(time_result > 5):
        s = round(time.time())
        salt = str(s)
        print("Here is the salt")
        print(salt)
    return(salt)

print("\n")
salt = salt_time(salt)

#******************KEY GENERATOR************
data = b'597133743677397A24432646294A404E635266556A586E5A72347537782141254
42A472D4B6150645367566B59703373357638792F423F4528482B4D62516554'
hash_len = 32
length = 64
info = b""

def hmac_sha512(key, data):
    return hmac.new(key, data, hashlib.sha512).digest()

ikm = hmac_sha512(key,data)

def hkdf(length, ikm, salt, info):
    prk = hmac_sha512(salt if len(salt) > 0 else bytes([0]*hash_len), ikm)
    t = b""
    okm = b""
    for i in range(int(ceil(length / hash_len))):
        t = hmac_sha512(prk, t + info + bytes([1+i]))
        okm += t
    return okm[:length]

salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
#print("Here is the first generated key")
```

```
#print(key)
print("*************************************")

#_______________________________________________________
#Derivative create and check

def derivative1_create(message, salt):
    print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    dialected_message = message

    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(dialected_message)
    message_copy = message
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The current length is:")
    print(len(mutable_bytes))
    print("The modified message without xid is:")
    print(mutable_bytes)
    total_message = mutable_bytes
    digester = blake2b(key, digest_size = 4)
    digester.update(mutable_bytes)
    mac1 = digester.hexdigest()
    print("Origional MAC Printed here")
    print(mac1)
    print("\n")
    translate_hex = bytes.fromhex(mac1)
    translate_hex = bytearray(translate_hex)
    print("Lastly, the new xid is:")
    print(translate_hex)
    print("Now to add the translate mac back to the message")
    total_message = total_message + translate_hex
    print(total_message)
    #dialected_message = message
    dialected_message = total_message
    return dialected_message

def derivative1_check(msg_recv,salt):
    #Transaction ID 0
    print("The msg_recv is:")
    print(msg_recv)
    transaction_id0 = b"\x00\x00\x00\x00"
    #checked_message = msg_recv
    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(msg_recv)
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The current length is:")
    print(len(mutable_bytes))
    print("The modified message without xid is:")
    print(mutable_bytes)

    #Creating a MAC to verify that the recv message matches what is expected
    print("\n")
    #print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    #print(key)
    print("The length of key is:")
    print(len(key))
    digester2 = blake2b(key, digest_size = 4)
    digester2.update(mutable_bytes)
    mac2 = digester2.hexdigest()
    print("MAC Printed here")
    print(mac2)
    print("\n")
    new_mac = binascii.unhexlify(mac2)
    new_mac = mac2
    new_number = int(new_mac, 16)
    print("Digest_calc is:")
    digest_calc = new_number
    print(digest_calc)

    #Deteremine the value sent by the message
    binary_msg = msg_recv
    msg = unpack_message(binary_msg)
    print("The type of the mesage is:")
    print(msg.header.message_type)
    print("The xid is:")
    print(msg.header.xid)
    value1_compare = msg.header.xid

    #Compare the two values and send to device is expected matches received
    print("Comparing values:")
    print(value1_compare)
    print(digest_calc)
    if(value1_compare == digest_calc):
        final_message = mutable_bytes + transaction_id0
    else:
        final_message = b"\x11\x11\x11\x11\x11\x11"
    checked_message = final_message
    print("The message to send on to device is:")
    print(checked_message)
    return checked_message

#_____________________________________________________
#Define functions for derivative 2

#*********Create Experimenter Message**********

def derivative2_create(message,salt):
    print("Creating derivative 2 message")

    version = b"x04"
    type_openflow = b"x04" #Experimenter type message
    length_openflow = b"x08" #Not sure about this
    total_message = bytearray()
    openflow_header_exp = (b'\x01\x04\x08')
    print("Here is the first portion of the header")
    total_message = total_message + openflow_header_exp
    print(total_message)

    print("The following is a temporary experiment ID:")
    print(b"\x01\x01\x01\x01")
    experiment_id = b"\x01\x01\x01\x01"
    print("The new message with the temp exp id for protocol dialecting is:")
    total_message = total_message + experiment_id
    print(total_message)
    print("\n")

    #********Read in OpenFlow message to correlate xid***
    print("The experimenter type is the unique code for NPS")
    print("The unique code for NPS is")
    print(b'\x02\x02\x02\x02')
```

```
experiment_type_nps = b"\x02\x02\x02\x02"
print("The new total message is:")
total_message = total_message + experiment_type_nps

#Determine the xid of the normal OpenFlow message
print("Here is the type of file read in at the proxy")
binary_msg = message
msg = unpack_message(binary_msg)
print(msg.header.message_type)
print(msg)
print("\n")

print("Here is the xid for the read in file which is used for the experiment type")
mutable_bytes = bytearray(binary_msg)
#print("Current length is:")
#print(len(mutable_bytes))
print(msg.header.xid)
value = msg.header.xid
value = (int(str(value)))
value_hex = (value).to_bytes(4, byteorder="big")
print(value_hex)
print("The new message with the experiment type (other message transaction ID is:")
total_message = total_message + value_hex
print(total_message)
print("\n")

#**********Create Final MAC for Verification of Experimenter Message**
print("\n")
#print("The key is:")
salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
#print(key)
```

```
    print("Creating the data portion of the message for the MAC:")
    #print("The data section with 512 bit MAC is:")
    digester4 = blake2b(key, digest_size = 64)
    digester4.update(binary_msg)
    digester4.update(total_message)
    mac2 = digester4.hexdigest()
    #print(mac2)
    translate_hex = bytes.fromhex(mac2)
    print("The translated mac is:")
    print(translate_hex)
    print("The total message is:")
    total_message = total_message + translate_hex
    print(total_message)
    derivative2_msg = total_message
    print("\n")
    return derivative2_msg


def derivative2_verify(openflow_msg, deriv2_message,salt):
    #***Verify the OpenFlow message**********************

    #It is necessary to read in both the OpenFlow Message and the Authentication Message
    print("*************************************************************")
    print("Reading in the other OpenFlow message")
    #This is the OpenFlow message that is being looked at first
    print("Here is the type of file read in at the proxy")
    msg_tocheck = unpack_message(openflow_msg)
    print(msg_tocheck.header.message_type)
    print(msg_tocheck)
    print("\n")

    print("Reading in authentication message to validate the OpenFlow message")
    data_from_proxy = deriv2_message

    print("\n")
    print("Removing the previous MAC to perform MAC check")
    mutable_bytes = bytearray(data_from_proxy)
    mutable_bytes_copy = bytearray(data_from_proxy)
    after_length = len(mutable_bytes) - 64

    j = 0
    while(True):
        del mutable_bytes_copy[j]
        if(len(mutable_bytes_copy) == 64):
            break

    while((len(mutable_bytes)) > after_length):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The modified message without MAC is:")
    print(mutable_bytes)

    print("The MAC parsed out from message is:")
    #mac_fill = hex(mac_fill)
    print(mutable_bytes_copy)



    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    print("Here is the key")
    #print(key)

    digester5 = blake2b(key, digest_size = 64)
    digester5.update(openflow_msg)
    print(openflow_msg)
    digester5.update(mutable_bytes)
```

```
    mac2 = digester5.hexdigest()
    translate_hex = bytes.fromhex(mac2)
    translate_hex = bytearray(translate_hex)
    print("\n")
    print("Generated MAC to check against Printed here")
    print(translate_hex)
    print("The MAC from the message is:")
    print(mutable_bytes_copy)

    if(translate_hex == mutable_bytes_copy):
        print("The derivative 2 MAC has passed the check")
        return openflow_msg
    else:
        print("The derivative 2 MAC did not pass the check")
        return b"\x11\x11\x11\x11\x11\x11"

#_________________________________________________________
#Setup network connections

#Create TCP/IP socket and UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("\n**Server is starting up**")

print("**Server is running**")
print("Ensure that Controller Proxy is running first")

#Bind the socket to the address and wait for the switch to attempt to connect
server_address = ('127.0.0.1', 6653)
sock.bind(server_address)
print("Starting up on %s port %s\n" % server_address)
sock.listen(1)
connection, client_address = sock.accept()


#Connect to the Controller Proxy
```

```
sock2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#Address of proxy
server2_address = ('127.0.0.3', 6673)
sock2.connect(server2_address)

#________________________________________

#Recieve hello from switch
data = connection.recv(32000)
receipt = 0
receipt = data
print("Switch 1 sent:", receipt)
print(type(receipt))

#Send OpenFlow Hello to deriivative1_create
receipt = derivative1_create(receipt, salt)

#Send dialected hello to proxy
print("Sending hello to proxy")
sock2.sendall(receipt)
print("Data sent back to controller")
print("\n")

#Receive hello from proxy
data = sock2.recv(32000)
receipt = 0
receipt = data
print("Proxy sent:", receipt)
print(type(receipt))
print("\n")

#Check authentication of OpenFlow Hello with derivative 1
receipt = derivative1_check(receipt, salt)

#Send data to switch
```

```
print("Sending data to switch")
connection.sendall(receipt)
print("Data sent to switch")
print("\n")

#Recieve data from proxy
data = sock2.recv(32000)
receipt = 0
receipt = data
print("Proxy sent:", receipt)
print(type(receipt))
print("\n")

#Receive derivative 2 from proxy
data = sock2.recv(32000)
deriv2_message = 0
deriv2_message = data
print("Proxy sent derv2 message:", deriv2_message)
print(type(deriv2_message))
print("\n")

#Need to verify derivative 2 and therefore openflow message
receipt = derivative2_verify(receipt, deriv2_message,salt)

#Send data to switch
print("Sending data to switch")
connection.sendall(receipt)
print("Data sent to switch")
print("\n")

#Recieve data from switch
data = connection.recv(32000)
receipt = 0
receipt = data
print("Switch sent:", receipt)
print(type(receipt))
print("\n")
print("Here is the length of the message from switch")
print(len(receipt))

if(len(receipt) > 300):
    print("Do not try to create derivative 2 due to size overload")
    #Send data to proxy
    print("Sending data back to controller")
    sock2.sendall(receipt)
    print("Data sent back to controller")
    print("\n")
else:
    #Send data to proxy
    print("Sending data back to controller")
    sock2.sendall(receipt)
    print("Data sent back to controller")
    print("\n")

    #Create deriv2 message
    deriv2_message = derivative2_create(receipt,salt)

    print("Sending experimenter to proxy")
    print(deriv2_message)
    sock2.sendall(deriv2_message)
    print("Data sent")
    print("\n")

#************

while(True):
    #Recieve data from proxy
    data = sock2.recv(32000)
    receipt = 0
    receipt = data
```

```
print("Proxy sent:", receipt)
print(type(receipt))
print("\n")
if(len(receipt) <70):
    #Receive derivative 2 from proxy
    data = sock2.recv(32000)
    deriv2_message = 0
    deriv2_message = data
    print("Proxy sent derv2 message:", deriv2_message)
    print(type(deriv2_message))
    print("\n")

#Need to verify derivative 2 and therefore openflow message
try:
    receipt = derivative2_verify(receipt, deriv2_message,salt)
except:
    print("Had an exception during verification of message.")

#Send data to switch
print("Sending data to switch")
connection.sendall(receipt)
print("Data sent to switch")
print("\n")

#Recieve data from switch
data = connection.recv(32000)
receipt = 0
receipt = data
print("Switch sent:", receipt)
print(type(receipt))
print("\n")
print("Here is the length of the message from switch")
print(len(receipt))

if(receipt == b''):
    #Send data to proxy’
    print("Received an empty string from switch so not sending experimenter message")
    print("Sending data back to controller")
    sock2.sendall(receipt)
    print("Data sent back to controller")
    print("\n")
    continue
else:
    #Send data to proxy
    print("Sending data back to controller")
    sock2.sendall(receipt)
    print("Data sent back to controller")
    print("\n")
    #Create deriv2 message
    deriv2_message = derivative2_create(receipt,salt)

    print("Sending experimenter to proxy")
    print(deriv2_message)
    sock2.sendall(deriv2_message)
    print("Data sent")
    print("\n")
```

APPENDIX E:
D1&D3 with TLS Controller Proxy

```
#Michael Sjoholm-Sierchio
#File: D1&D3_withtls_ControllerProxy.py
#Ref: https://pymotw.com/2/socket/binary.html
#Ref: https://stackoverflow.com/questions/14043886/python
#-2-3-convert-integer-to-bytes-cleanly
# Reference:  docs.kytos.io kytos developer guide
# Reference: Working with binary data in python dev_dungeon
# Reference: Low Level OpenFlow Messages Parser used by Kytos SDN
#Platform https://kytos.io
# Reference: For secure hash library https://docs.python.org/dev/library/hashlib.html?
#highlight=s
# Reference: https://www.devdungeon.com/content/working-binary-data-python
# Reference: https://pymotw.com/3/hmac
# Reference: https://docs.python.org/3/library/hashlib.html
# Reference: https://en.wikipedia.org/wiki/HKDF

import socket
import sys
import time
import binascii
import struct
import sys
import os
import datetime
import hashlib
import hmac
import base64
import binascii
import socket
from hashlib import blake2b
from threading import Timer
from math import ceil
from pyof.foundation.base import GenericStruct
#from pyof.foundation.basic_types import UBInt8, UBInt16
from pyof.v0x01.common.utils import unpack_message
from pyof.foundation.base import GenericMessage
from pyof.v0x04.common.header import Header
from pyof.v0x01.symmetric.hello import Hello
from pyof.v0x01.controller2switch.features_request import FeaturesRequest
from pyof.v0x01.symmetric.echo_request import EchoRequest
from pyof.v0x01.symmetric.vendor_header import VendorHeader
#***************IMPORT ABOVE*******************

#___________________________________________
#Setup the key and generation

print("\n")
print("Program is running")
print("Protocol Dialect Walkthrough Initiated")
print("--------------------------------------")
#print("Include starting Mini-Net here")
print("\n")

#********Read In Long-Term Key and Create Keys******
print("Reading in shared key")
#Obtain the shared secret key at the beginning of operation from a file or OOB

key =  b'457c311719813785096ef45f466aead3db4e535f4a7b0d06084621c0e01220a6b
43b90879fc23189d4fed6456e31529905bdc83056feda5940444893a83808bd'
print("Here is the key from file")
#print(key)
print("Here is the length of the key")
print(len(key))
#Creat a salt, which will change every 5 seconds
standard_time = time.time()
s = round(standard_time)
```

```
salt = str(s)

def salt_time(salt):
    salt = int(salt)
    time_check = round(time.time())
    time_result = time_check - salt
    if(time_result > 5):
        s = round(time.time())
        salt = str(s)
        print("Here is the salt")
        print(salt)
    return(salt)

print("\n")
salt = salt_time(salt)

#******************COMPLETE MAKING SALT*************

#******************KEY GENERATOR************
data = b'597133743677397A24432646294A404E635266556A586E5A72347537782141254442
A472D4B6150645367566B5970337335763879 2F423F4528482B4D62516554'
hash_len = 32
length = 64
info = b""


def hmac_sha512(key, data):
    return hmac.new(key, data, hashlib.sha512).digest()

ikm = hmac_sha512(key,data)

def hkdf(length, ikm, salt, info):
    prk = hmac_sha512(salt if len(salt) > 0 else bytes([0]*hash_len), ikm)
    t = b""
    okm = b""
    for i in range(int(ceil(length / hash_len))):
        t = hmac_sha512(prk, t + info + bytes([1+i]))
        okm += t
    return okm[:length]


salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
print("Here is the first generated key")
#print(key)
print("*************************************")


#_______________________________________________________
#Derivative create and check

def derivative1_create(message, salt):
    #print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    dialected_message = message


    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(dialected_message)
    message_copy = message
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The current length is:")
    print(len(mutable_bytes))
    print("The modified message without xid is:")
    print(mutable_bytes)
    total_message = mutable_bytes
    digester = blake2b(key, digest_size = 4)
    digester.update(mutable_bytes)
    mac1 = digester.hexdigest()
    print("Origional MAC Printed here")
    print(mac1)
    print("\n")
    translate_hex = bytes.fromhex(mac1)
    translate_hex = bytearray(translate_hex)
    print("Lastly, the new xid is:")
    print(translate_hex)
    print("Now to add the translate mac back to the message")
    total_message = total_message + translate_hex
    print(total_message)
    #dialected_message = message
    dialected_message = total_message
    return dialected_message

def derivative1_check(msg_recv,salt):
    #Transaction ID 0
    print("The msg_recv is:")
    print(msg_recv)
    transaction_id0 = b"\x00\x00\x00\x00"
    #checked_message = msg_recv
    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(msg_recv)
    print("Current length is:")
    print(len(mutable_bytes))
```

```
while((len(mutable_bytes)) > 4):
    i = len(mutable_bytes)
    i = i-1
    del mutable_bytes[i]
print("The current length is:")
print(len(mutable_bytes))
print("The modified message without xid is:")
print(mutable_bytes)

#Creating a MAC to verify that the recv message matches what is expected
print("\n")
#print("The key is:")
salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
#print(key)
print("The length of key is:")
print(len(key))
digester2 = blake2b(key, digest_size = 4)
digester2.update(mutable_bytes)
mac2 = digester2.hexdigest()
print("MAC Printed here")
print(mac2)
print("\n")
new_mac = binascii.unhexlify(mac2)
new_mac = mac2
new_number = int(new_mac, 16)
print("Digest_calc is:")
digest_calc = new_number
print(digest_calc)

#Deteremine the value sent by the message
binary_msg = msg_recv
```

```
    msg = unpack_message(binary_msg)
    print("The type of the mesage is:")
    print(msg.header.message_type)
    print("The xid is:")
    print(msg.header.xid)
    value1_compare = msg.header.xid

    #Compare the two values and send to device is expected matches received
    print("Comparing values:")
    print(value1_compare)
    print(digest_calc)
    if(value1_compare == digest_calc):
        #final_message = mutable_bytes + transaction_id0
        final_message = 1
    else:
        final_message = 0
        #quit(1)
    checked_message = final_message
    print("The message to send on to device is:")
    print(checked_message)
    return checked_message

#Define functions for derivative 3
#*********Create Derivative 3 Message*********

def derivative3_create(message,salt):
    print("Creating derivative 3 message")

    total_message = bytearray()
    total_message = total_message + message
    print("Here is the original message")
    print(total_message)
    print("\n")

    #**********Create MAC for Message**
    print("\n")
    #print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    #print(key)
    print("Creating the data portion of the message for the MAC:")
    #print("The data section with 512 bit MAC is:")
    digester4 = blake2b(key, digest_size = 64)
    digester4.update(total_message)
    mac2 = digester4.hexdigest()
    #print(mac2)
    translate_hex = bytes.fromhex(mac2)
    print("The translated mac is:")
    print(translate_hex)
    print("The total message is:")
    total_message = total_message + translate_hex
    print(total_message)
    derivative3_msg = total_message
    print("\n")
    return derivative3_msg


def derivative3_verify(deriv3_message,salt):
    #***Verify the OpenFlow message*********************

    print("Reading in authentication message to validate the OpenFlow message")
    data_from_proxy = deriv3_message

    print("\n")
    print("Removing the previous MAC to perform MAC check")
    mutable_bytes = bytearray(data_from_proxy)
    mutable_bytes_copy = bytearray(data_from_proxy)
```

```
after_length = len(mutable_bytes) - 64

j = 0
while(True):
    del mutable_bytes_copy[j]
    if(len(mutable_bytes_copy) == 64):
        break

while((len(mutable_bytes)) > after_length):
    i = len(mutable_bytes)
    i = i-1
    del mutable_bytes[i]
print("The modified message without MAC is:")
print(mutable_bytes)

print("The MAC parsed out from message is:")
#mac_fill = hex(mac_fill)
print(mutable_bytes_copy)



salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
print("Here is the key")
#print(key)

digester5 = blake2b(key, digest_size = 64)
digester5.update(mutable_bytes)
mac2 = digester5.hexdigest()
translate_hex = bytes.fromhex(mac2)
translate_hex = bytearray(translate_hex)
print("\n")
print("Generated MAC to check against Printed here")
```

```
    print(translate_hex)
    print("The MAC from the message is:")
    print(mutable_bytes_copy)

    if(translate_hex == mutable_bytes_copy):
        print("The MAC has passed the check and returning to proxy")
        print("Here is the message to forward")
        print(mutable_bytes)
        return mutable_bytes
    else:
        print("The MAC did not pass the check")
        return b"\x11\x11\x11\x11\x11\x11"




#_________________________________________________
#Setup network connections



#Establish the socket settings to the controller
sock = socket.socket(socket.AF_INET, socket. SOCK_STREAM)
server_name = 'localhost'
server_address = ('127.0.0.5', 6633)
sock.connect(server_address)
print("Connected to controller")

print("\n")
print("Waiting for the other proxy to connect")
#Establish the socket settings for the switch proxy
#This script is the server
sock2 = socket.socket(socket.AF_INET, socket. SOCK_STREAM)
server2_address = ('127.0.0.3', 6673)
sock2.bind(server2_address)
sock2.listen(1)
connection, client_address = sock2.accept()
```

```
print("The other proxy has connected")


#____________________________________________________________
#Receive hello from proxy
data = connection.recv(32000)
receipt = 0
receipt = data
print("Switch sent:", receipt)
print(type(receipt))
print("\n")

#Check authentication of OpenFlow Hello
receipt = derivative1_check(receipt, salt)

if(receipt == 1):
    print("The check of derivative has passed.")
else:
    print("The check of the derivative has not passed.")
    print("Closing the connection.")
    exit()

while(True):
    #Receive data from proxy
    data = connection.recv(60000)
    receipt = 0
    receipt = data
    print("Switch sent:", receipt)
    print(type(receipt))
    print("\n")

    receipt = derivative3_verify(receipt,salt)

    #Send data from proxy to controller
    print("Sending data to controller")
    print(receipt)
    sock.sendall(receipt)
    print("Data sent")
    print("\n")

    #Recieve data from controller
    data = sock.recv(60000)
    receipt = 0
    receipt = data
    print("Controller 1 sent:", receipt)
    print(type(receipt))
    print("\n")

    receipt = derivative3_create(receipt,salt)

    #Send data from controller to proxy
    print("Sending data to proxy")
    print(receipt)
    connection.sendall(receipt)
    print("Data sent")
    print("\n")
```

APPENDIX F:
D1&D3 with TLS Switch Proxy

```
#Michael Sjoholm-Sierchio
#File: D1&D3_withtls_SwitchProxy.py
#Ref: https://pymotw.com/2/socket/binary.html
#Ref: https://stackoverflow.com/questions/14043886/python
#-2-3-convert-integer-to-bytes-cleanly
#Ref: https://www.cyberciti.biz/faq/python-convert-string-to-int-functions/
# Reference:  docs.kytos.io kytos developer guide
# Reference: Working with binary data in python dev_dungeon
# Reference: Low Level OpenFlow Messages Parser used by Kytos SDN
#Platform https://kytos.io
# Reference: For secure hash library https://docs.python.org/dev/library/hashlib.html?
#highlight=s
# Reference: https://www.devdungeon.com/content/working-binary-data-python
# Reference: https://pymotw.com/3/hmac
# Reference: https://docs.python.org/3/library/hashlib.html
# Reference: https://en.wikipedia.org/wiki/HKDF

import sys
import socket
import time
import random
import binascii
import struct
import os
import datetime
import hashlib
import hmac
import base64
import binascii
import socket
from hashlib import blake2b
from threading import Timer
from math import ceil
from pyof.foundation.base import GenericStruct
#from pyof.foundation.basic_types import UBInt8, UBInt16
from pyof.v0x01.common.utils import unpack_message
from pyof.foundation.base import GenericMessage
from pyof.v0x04.common.header import Header
from pyof.v0x01.symmetric.hello import Hello
from pyof.v0x01.controller2switch.features_request import FeaturesRequest
from pyof.v0x01.symmetric.echo_request import EchoRequest
from pyof.v0x01.symmetric.vendor_header import VendorHeader
#***************IMPORT ABOVE*******************

#____________________________________________
#Setup the key and generation

#********Read In Long-Term Key and Create Keys******
print("Reading in shared key")
#Obtain the shared secret key at the beginning of operation from a file or OOB

key =  b'457c311719813785096ef45f466aead3db4e535f4a7b0d06084621c0e01220a6b
43b90879fc23189d4fed6456e31529905bdc83056feda5940444893a83808bd'
print("Here is the key from file")
#print(key)
print("Here is the length of the key")
print(len(key))
#Creat a salt, which will change every 5 seconds
standard_time = time.time()
s = round(standard_time)
salt = str(s)

def salt_time(salt):
    salt = int(salt)
    time_check = round(time.time())
    time_result = time_check - salt
    if(time_result > 5):
        s = round(time.time())
        salt = str(s)
        print("Here is the salt")
        print(salt)
    return(salt)

print("\n")
salt = salt_time(salt)

#******************KEY GENERATOR************
data = b'597133743677397A24432646294A404E635266556A586E5A72347537782141254442A472D4B61506453
hash_len = 32
length = 64
info = b""

def hmac_sha512(key, data):
    return hmac.new(key, data, hashlib.sha512).digest()

ikm = hmac_sha512(key,data)

def hkdf(length, ikm, salt, info):
    prk = hmac_sha512(salt if len(salt) > 0 else bytes([0]*hash_len), ikm)
    t = b""
    okm = b""
    for i in range(int(ceil(length / hash_len))):
        t = hmac_sha512(prk, t + info + bytes([1+i]))
        okm += t
    return okm[:length]

salt = salt_time(salt)
salt = str(salt)
salt = bytes(salt, "utf8")
key = hkdf(length, ikm, salt, info)
key = binascii.hexlify(key)
```

```
#print("Here is the first generated key")
#print(key)
print("*************************************")

#________________________________________________________
#Derivative create and check

def derivative1_create(message, salt):
    #print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    dialected_message = message

    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(dialected_message)
    message_copy = message
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The current length is:")
    print(len(mutable_bytes))
    print("The modified message without xid is:")
    print(mutable_bytes)
    total_message = mutable_bytes
    digester = blake2b(key, digest_size = 4)
    digester.update(mutable_bytes)
    mac1 = digester.hexdigest()
    print("Origional MAC Printed here")
    print(mac1)
    print("\n")
    translate_hex = bytes.fromhex(mac1)
    translate_hex = bytearray(translate_hex)
    print("Lastly, the new xid is:")
    print(translate_hex)
    print("Now to add the translate mac back to the message")
    total_message = total_message + translate_hex
    print(total_message)
    #dialected_message = message
    dialected_message = total_message
    return dialected_message

def derivative1_check(msg_recv,salt):
    #Transaction ID 0
    print("The msg_recv is:")
    print(msg_recv)
    transaction_id0 = b"\x00\x00\x00\x00"
    #checked_message = msg_recv
    print("Removing the previous xid to perform MAC")
    mutable_bytes = bytearray(msg_recv)
    print("Current length is:")
    print(len(mutable_bytes))
    while((len(mutable_bytes)) > 4):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The current length is:")
    print(len(mutable_bytes))
    print("The modified message without xid is:")
    print(mutable_bytes)

    #Creating a MAC to verify that the recv message matches what is expected
    print("\n")
    #print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    #print(key)
    print("The length of key is:")
    print(len(key))
    digester2 = blake2b(key, digest_size = 4)
    digester2.update(mutable_bytes)
    mac2 = digester2.hexdigest()
    print("MAC Printed here")
    print(mac2)
    print("\n")
    new_mac = binascii.unhexlify(mac2)
    new_mac = mac2
    new_number = int(new_mac, 16)
    print("Digest_calc is:")
    digest_calc = new_number
    print(digest_calc)

    #Deteremine the value sent by the message
    binary_msg = msg_recv
    msg = unpack_message(binary_msg)
    print("The type of the mesage is:")
    print(msg.header.message_type)
    print("The xid is:")
    print(msg.header.xid)
    value1_compare = msg.header.xid

    #Compare the two values and send to device is expected matches received
    print("Comparing values:")
    print(value1_compare)
    print(digest_calc)
    if(value1_compare == digest_calc):
        final_message = mutable_bytes + transaction_id0
    else:
        final_message = b"\x11\x11\x11\x11\x11\x11"
        #quit(1)
    checked_message = final_message
    print("The message to send on to device is:")
    print(checked_message)
    return checked_message

#Define functions for derivative 3

#*********Create Experimenter Message*********

def derivative3_create(message,salt):
    print("Creating derivative 3 message")

    total_message = bytearray()
    total_message = total_message + message
    print("Here is the original message")
    print(total_message)
    print("\n")

    #**********Create MAC for Message**
    print("\n")
    #print("The key is:")
    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    #print(key)
    print("Creating the data portion of the message for the MAC:")
    #print("The data section with 512 bit MAC is:")
    digester4 = blake2b(key, digest_size = 64)
    digester4.update(total_message)
    mac2 = digester4.hexdigest()
    #print(mac2)
    translate_hex = bytes.fromhex(mac2)
    print("The translated mac is:")
    print(translate_hex)
    print("The total message is:")
    total_message = total_message + translate_hex
    print(total_message)
    derivative3_msg = total_message
    print("\n")
    return derivative3_msg

def derivative3_verify(deriv3_message,salt):
    #***Verify the message**********************

    print("Reading in authentication message to validate the OpenFlow message")
    data_from_proxy = deriv3_message
    print("\n")
    print("Removing the previous MAC to perform MAC check")
    mutable_bytes = bytearray(data_from_proxy)
    mutable_bytes_copy = bytearray(data_from_proxy)
    after_length = len(mutable_bytes) - 64

    j = 0
    while(True):
        del mutable_bytes_copy[j]
        if(len(mutable_bytes_copy) == 64):
            break

    while((len(mutable_bytes)) > after_length):
        i = len(mutable_bytes)
        i = i-1
        del mutable_bytes[i]
    print("The modified message without MAC is:")
    print(mutable_bytes)
    print("The MAC parsed out from message is:")
    #mac_fill = hex(mac_fill)
    print(mutable_bytes_copy)

    salt = salt_time(salt)
    salt = str(salt)
    salt = bytes(salt, "utf8")
    key = hkdf(length, ikm, salt, info)
    key = binascii.hexlify(key)
    #print("Here is the key")
    #print(key)

    digester5 = blake2b(key, digest_size = 64)
    digester5.update(mutable_bytes)
    mac2 = digester5.hexdigest()
    translate_hex = bytes.fromhex(mac2)
    translate_hex = bytearray(translate_hex)
    print("\n")
    print("Generated MAC to check against Printed here")
    print(translate_hex)
    print("The MAC from the message is:")
    print(mutable_bytes_copy)

    if(translate_hex == mutable_bytes_copy):
        print("The MAC has passed the check and returning to proxy")
        print("Here is the message to forward")
        print(mutable_bytes)
        return mutable_bytes
    else:
        print("The MAC did not pass the check")
        return b"\x11\x11\x11\x11\x11\x11"

#________________________________________________
#Setup network connections
```

```
#Create TCP/IP socket and UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("\n**Server is starting up**")

print("**Server is running**")
print("Ensure that Controller Proxy is running first")

#Bind the socket to the address and wait for the switch to attempt to connect
server_address = ('127.0.0.1', 6653)
sock.bind(server_address)
print("Starting up on %s port %s\n" % server_address)
sock.listen(1)
connection, client_address = sock.accept()

#Connect to the Controller Proxy
sock2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#Address of proxy
server2_address = ('127.0.0.3', 6673)
sock2.connect(server2_address)

#_________________________________________

receipt = b'\x05\x00\x00\x08\x00\x00\x02\x92'
receipt = derivative1_create(receipt,salt)

#Send dialected hello to proxy
print("Sending hello to proxy")
sock2.sendall(receipt)
print("Data sent back to controller")
print("\n")

while(True):
    #Recieve data from switch
    data = connection.recv(60000)
    receipt = 0
    receipt = data
    print("Switch 1 sent:", receipt)
    print(type(receipt))

    receipt = derivative3_create(receipt,salt)

    #Send data from switch to proxy
    print("Sending hello to proxy")
    sock2.sendall(receipt)
    print("Data sent to proxy")
    print("\n")

    #Receive data from proxy
    data = sock2.recv(60000)
    receipt = 0
    receipt = data
    print("Proxy sent:", receipt)
    print(type(receipt))
    print("\n")

    if(receipt == b''):
        continue

    receipt = derivative3_verify(receipt,salt)

    #Send data to switch
    print("Sending data to switch")
    connection.sendall(receipt)
    print("Data sent to switch")
    print("\n")
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[2] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[3] "OpenFlow Switch Specification Version 1.5.1 (Protocol Version 0x06)," https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, accessed: 2018-09-02.

[4] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2013, pp. 151–152.

[5] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN Control Plane," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 451–468.

[6] Joint Task Force, "Sp 800-37 Rev 2. Risk management framework for information systems and organizations: A Systems Life Cycle Approach for Security and Privacy," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep., 2018.

[7] S. Mishra and M. Polychronakis, "Shredder: Breaking Exploits through API Specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 1–16.

[8] M. Nieles, D. Kelley, and P. Y. Victoria, "Sp 800-12 Rev 1. An introduction to computer security," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep., 2017.

[9] S. Jajodia, S. Noel, P. Kalapa, M. Albanese, and J. Williams, "Cauldron Mission-centric cyber situational awareness with defense in depth," in *2011-MILCOM 2011 Military Communications Conference*. IEEE, 2011, pp. 1339–1344.

[10] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, 2016, pp. 164–178.

[11] F. Hu, *Network Innovation through OpenFlow and SDN: Principles and Design*. CRC Press, 2014.

[12] “Mininet An Instant Virtual Network on your Laptop,” http://mininet.org/, accessed: 2018-09-02.

[13] “Wireshark user’s guide,” https://www.wireshark.org/docs/wsug_html_chunked/, accessed: 2018-09-01.

[14] Y. Sheffer, R. Holz, and P. Saint-Andre, “Summarizing known attacks on Transport Layer Security (TLS) and Datagram Tls (DTLS),” Tech. Rep., 2015.

[15] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, “DROWN: Breaking TLS with SSLv2,” in *25th USENIX Security Symposium*, Aug. 2016.

[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[17] J. B. Perazzone, P. L. Yu, B. M. Sadler, and R. S. Blum, “Cryptographic side-channel signaling and authentication via fingerprint embedding,” *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 9, pp. 2216–2225, Sep. 2018.

[18] K. Sakiyama, M. Kasuya, T. Machida, A. Matsubara, Y. Kuai, Y.-i. Hayashi, T. Mizuki, N. Miura, and M. Nagata, “Physical authentication using side-channel information,” in *2016 4th International Conference on Information and Communication Technology (ICoICT)*. IEEE, 2016, pp. 1–6.

[19] D. Samociuk, “Secure communication between OpenFlow switches and controllers,” *AFIN 2015*, vol. 39, 2015.

[20] R. Perlman, “An overview of PKI trust models,” *IEEE Network*, vol. 13, no. 6, pp. 38–43, 1999.

[21] B. Hale, C. Carr, and D. Gligoroski, “CARIBE: Cascaded IBE for maximum flexibility and user-side control,” in *International Conference on Cryptology in Malaysia*. Springer, 2016, pp. 389–408.

[22] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, “Feature-based comparison and selection of software defined networking (SDN) controllers,” in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, Jan 2014, pp. 1–7.

[23] "SSL on Open vSwitch and OVS controller," Apr 2014, accessed: 2019-06-02. Available: https://github.com/mininet/mininet/wiki/SSL-on-Open-vSwitch-and-ovs-controller

[24] "Message Layer," accessed: 2018-11-04. Available: http://flowgrammable.org/sdn/openflow/message-layer/

[25] C.-H. J. Wu and J. D. Irwin, *Introduction to Computer Networks and Cybersecurity*. CRC Press, 2016.

[26] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: Simpler, smaller, fast as MD5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[27] A. Boldyreva, V. Goyal, and V. Kumar, "Identity-based encryption with efficient revocation," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 417–426.

[28] D. Hellmann, "Sending binary data," Mar 2019, accessed: 2018-11-10. Available: https://pymotw.com/2/socket/binary.html

[29] A. Hayden, "Python 2,3 Convert integer to "bytes" cleanly," Nov 2014, accessed: 2018-11-05. Available: https://stackoverflow.com/questions/14043886/python-2-3-convert-integer-to-bytes-cleanly

[30] "Developer Guide," accessed: 2018-11-02. Available: https://docs.kytos.io/developer/

[31] NanoDano, "Working with binary data in Python," Nov 2018, accessed: 2018-11-01. Available: https://www.devdungeon.com/content/working-binary-data-python

[32] D. Hellmann, "HMAC - cryptographic message signing and verification," Jul 2017, accessed: 2019-04-10. Available: https://pymotw.com/3/hmac/

[33] "Hashlib - secure hashes and message digests," Oct 2019, accessed: 2019-05-04. Available: https://docs.python.org/3/library/hashlib.html

[34] "HKDF," Feb 2019, accessed: 2019-06-02. Available: https://en.wikipedia.org/wiki/HKDF

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California