November 2016

Protecting Controllers against Denial-of-Service Attacks in Software-Defined Networks

Jingrui Li
University of Massachusetts Amherst

PROTECTING CONTROLLERS AGAINST DENIAL-OF-SERVICE ATTACKS IN SOFTWARE-DEFINED NETWORKS

A Thesis Presented

by

JINGRUI LI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2016

Electrical and Computer Engineering

PROTECTING CONTROLLERS AGAINST DENIAL-OF-SERVICE ATTACKS IN SOFTWARE-DEFINED NETWORKS

A Thesis Presented

by

JINGRUI LI

Approved as to style and content by:

Tilman Wolf, Chair

Weibo Gong, Member

David Irwin, Member

C.V. Hollot, Department Head
Electrical and Computer Engineering

ABSTRACT

PROTECTING CONTROLLERS AGAINST DENIAL-OF-SERVICE ATTACKS IN SOFTWARE-DEFINED NETWORKS

SEPTEMBER 2016

JINGRUI LI

B.S., NANJING UNIVERSITY OF SCIENCE AND TECHNOLOGY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Tilman Wolf

Connection setup in software-defined networks (SDN) requires considerable amounts of processing, communication, and memory resources. Attackers can target SDN controllers defense mechanism based on a proof-of-work protocol. This thesis proposes a new protocol to protect controllers against such attacks, shows implementation of the system and analyze the its performance. The key characteristics of this protocol, namely its one-way operation, its requirement for freshness in proofs of work, its adjustable difficulty, its ability to work withmultiple network providers, and its use of existing TCP/IP header fields, ensure that this approach can be used in practice.

TABLE OF CONTENTS

Page

ABSTRACT . . . . **iii**

LIST OF TABLES . . . . **vi**

LIST OF FIGURES . . . . **vii**

CHAPTER

1. INTRODUCTION **1**

1.1 Background 1
1.2 Motivation 2
1.3 Contribution 3
1.4 Organization of the Dissertation 4

2. RELATED WORK **5**

2.1 Attacks against Controllers in SDN 5
2.2 DDoS Defense in SDN 5
2.3 Proof of Work 7

3. PROBLEM STATEMENT **8**

3.1 Connection Setup in SDN and Attack 8
3.2 Security Model 11

3.2.1 Security Requirements 11
3.2.2 Attacker Capabilities 12

3.3 Performance Requirement 12

4. CONTROLLER PROTECTION PROTOCOL DESIGN **13**

4.1 Main Idea 13
4.2 System Architecture 16

4.3 Controller Protection Protocol Operation 18

4.3.1 Parameter Distribution 18
4.3.2 Proof of Work Generation 19
4.3.3 Proof of Work Verification 20
4.3.4 Multiple Network Providers 21

4.4 Proof-of-Work Design 21
4.5 Protocol Details 23

5. EVALUATION 25

5.1 Security Evaluation 25
5.2 Proof of Work Implementation 26

5.2.1 Proof of Work Evaluation 26
5.2.2 Proof-of-work Implementation on End-System 29

5.3 SDN Prototype Evaluation 33

5.3.1 Processing time of controller 33
5.3.2 Performance of controller under attacks in SDN 34

6. SUMMARY AND FUTURE WORK 38

BIBLIOGRAPHY 40

LIST OF TABLES

Table **Page**

5.1 Average of generation time and verification time of proofs of work for different cryptographic functions and complexity parameters. 28

5.2 Average ratio of generation time and verification time of proofs of work for different cryptographic functions and complexity parameters. 28

5.3 Measurement results of processing times on SDN controller. Proof of work uses SHA-256 with a complexity of c_t=12 bits of zeros. The regular SDN controller cannot distinguish between connection requests with valid or invalid POWs. Results are averages with standard deviation over 1000 connection request measurements. 33

LIST OF FIGURES

Figure **Page**

1.1 Architecture of software-defined network. 1

3.1 Simple topology of example software-defined network. 8

3.2 Space-time diagram of network interactions during connection setup in a software-defined network (topology from Figure 3.1, single SDN network provider, uni-directional communication). 9

3.3 Space-time diagram of denial-of-service attack on software-defined network controller (topology from Figure 3.1, single SDN network provider, uni-directional communication, packet forward through SDN omitted for readability). 11

4.1 Space-time diagram of network interactions during connection setup in a software-defined network using Controller Protection Protocol (topology from Figure 3.1, single SDN network provider, uni-directional communication). 14

4.2 Space-time diagram of denial-of-service attack on software-defined network controller using Controller Protection Protocol (topology from Figure 3.1, single SDN network provider, uni-directional communication). 15

4.3 System architecture of Controller Protection Protocol with central Controller Protection Protocol Authority that distributes current CPP parameters to SDN controllers for verification of proofs of work in connection requests. 17

4.4 Layout of header fields in IP header and TCP header used 5-tuple flow information and proof of work storage. 24

5.1 Distribution of generation and verification times for proofs of work in CPP (complexity c_t=10 zeroes). 27

5.2 Average generation and verification times for proofs of work in CPP for different complexity parameters (complexity c_t=$[8, 9, 10, 11, 12, 13, 14, 16, 20, 24]$ zeroes from bottom to top). 27

5.3 Initial idea of implementation for CPPA protocol on host 30

5.4 Generating proof of work in Ubuntu 31

5.5 Wireshark in Ubuntu 32

5.6 Verify the proof-of-work generated in Ubuntu 32

5.7 Topology of experiment on GENI 34

5.8 Attacks to regular SDN controller (attack with 1000 packets/second) 35

5.9 Attacks to regular SDN controller (attack with 2000 packets/second) 36

5.10 Attacks to regular SDN controller (attack with 3000 packets/second) 36

5.11 Attacks to regular SDN controller (attack with 10000 packets/second) 36

5.12 Attack to CPP-enabled SDN controller (attack with 1000 packets/second) 36

5.13 Attack to CPP-enabled SDN controller(attacks with 10000 packets/second) 37

5.14 Flood attack to CPP-enabled SDN controller 37

CHAPTER 1

INTRODUCTION

1.1 Background

Software-defined networking (SDN) is an attracting architecture that decouples the network control and forwarding functions, and the underlying infrastructure, enabling the network control to be prpgrammable. Therefore, the SDN is centrally managable, dynamic and adaptable. [1,2]

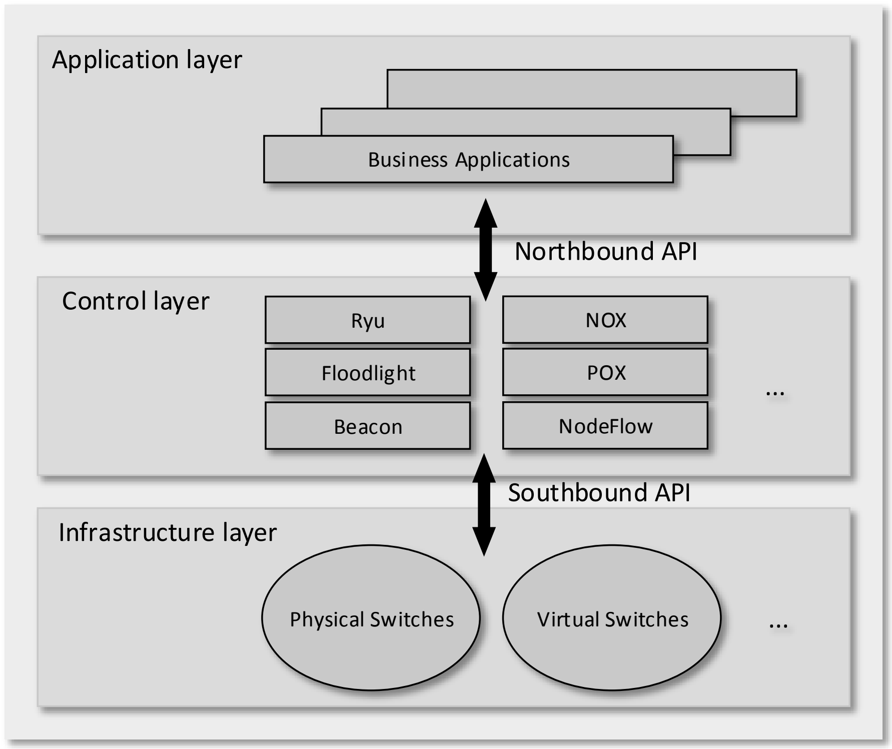


Figure 1.1. Architecture of software-defined network.

The high-level architecture of SDN presented by Open Networking Foundation (ONF), an organization dedicated to the development and standarization of SDN, is shown in Figure 1.1 [3].

- Infrastructure layer: Also named as data plane, it consists forwarding elements, such as phisical and virtual switches. These switches are accessible via an open interface to switch and forwarded packets.
- Control layer: Also named as control plane, it consists software-based controllers. The SDN controller is a logical centralized entity summarizing the network state for applications and translating application requirements to low-level rules.
- Application layer: It mainly consists of end-user applications.

1.2 Motivation

SDN switches match incoming traffic against a set of flow rules that have been installed by the controller. For new connections that have no matching rules, the switch forwards the flow information to the SDN controller. The controller then makes a routing decision and informs all switches along the path so that they can install a matching rule in their flow tables.

The ability to control the routing of individual flows in SDN is convenient for traffic management in data centers [4], for implementing virtualized network functions [5], and for offering customized network services [6]. A critical aspect of SDN is the operation and performance of the controller, which is a (logically or physically) centralized component and needs to handle routing decisions for all traffic through the network [7]. To ensure that these controllers have a suitable level of performance and reliability, a variety of distributed designs have been proposed [8, 9].

Despite the robust designs for SDN controllers, these systems represent attractive targets for malicious attackers. The basic operation of SDN exhibits an imbalance between the small amount of work that is necessary (sending of a packet with a new 5-tuple) to trigger the large amount to trigger large amount of work that is then performed by the SDN controller (route computation and setup of flow rules in switches). An attacker can exploit this imbalance by simply sending crafted flows, triggering a route computation with each packet and effectively overloading the SDN controller and filling flow tables in switches.

To level this imbalance, we introduce the Controller Protection Protocol (CPP), which requires systems wanting to connect through an SDN network to commit resources *before* an SDN controller commits resources for route computation and setup. In our case, the connecting system needs to include a proof-of-work (POW) [10] with the initial packet of a connection. The SDN controller can verify the correctness of the POW easily and thus discard attack traffic with invalid POWs with low overhead. Using this approach, an attacker needs to dedicate a large amount of computational resources in order to send large amounts of attack traffic that triggers route computation on the SDN controller, thus making an attack potentially prohibitively expensive.

1.3 Contribution

This thesis addresses the how to defense flood attack towards control plane of SDN. First, we review related researches about security issues in SDN and come up with the idea of using proof-of-work to solve flood attack targeting controllers in SDN. Then, we study different algorithms and design some experiment to find out the proper one used in SDN. Finally, we propose the entire system to implement the initial idea.

The main contributions are as follows:

- Propose a one-way transmission of POW to provide necessary efficiency for operation in SDN.
- Design the entire system applying the POW to protect controllers in SDN against flood attack.
- Devise the experiments and applications to implement CPP.

1.4 Organization of the Dissertation

The remainder of the thesis is organized as follows: Chapter 2 discusses related work. The problem of using proof-of-work in network protocols is explained in more detail in Chapter 3. The design of Controller Protection Protocol, including the design of the proof-of-work and the communication protocol, is described in Chapter 4. The evaluation results showing effectiveness of CPP is in Chapter 5, and Chapter 6 summarize the entire approach.

CHAPTER 2

RELATED WORK

2.1 Attacks against Controllers in SDN

Since the SDN seperate the control plane from the data plane, the data plane have to ask the control plane for flow rules if the coming packets don't match the current flow table. Therefore, it usually takes more time to handle the initial packet than the next ones. The attacker can easily detect SDN by measuring the difference of response time for new-flow and exist-flow [11]. After successfully detecting SDN, the attacker can make the flooding attack towards the controller, causing denial-of-service in SDN.

2.2 DDoS Defense in SDN

SDN controller attack is conceptually similar to SYN flood attack on servers [12]. When a client need to establish a new connection to the server, TCP three-way handshake is applied. The steps are as follows:

1. The client sends a SYN message to request a new connection.
2. The server replies a SYN-ACK message to the client to acknowlege the client that the request is received.
3. The client sends a ACK message back to server for the SYN-ACK message. The connection than is established.

To make a denial-of-service attack, the attacker can send traffic to server requesting TCP connectio without responding the ACK message in step 3. Or, the attacker can spoof the IP address, letting the server send SYN-ACK to the falsified IP address so that the server cannot get the responce. Therefore, the server has to keep a lot of half-open-connections causing depletion of resourses and denial of legitimate requests.

There is an effective solution to protecting servers through SYN cookies [13]. When the server receives the TCP SYN packet, the server sends a TCP SYN-ACK packet back to the client with a structed sequence number since the initial sequence number is chosen by the sender. The sequece number, i.e., the SYN cookie is structed according to the maximum segment size that the server uses to store the SYN queue entry and the information of the SYN packet, i.e., the IP address and port number of the client, and the timestamp. When the server get the ACK packet from the client, the server can retrieve the information from the acknowlege number which is related to the SYN-ACK sequence number, and than setup the conneciotn. By using SYN cookies, the servers need not keep the half-open-connetions so the flood attack cannot exhaut the resources of the server. This approach is suitable for web requests since the exchange can be elegantly combined with the Transmission Control Protocol (TCP) connection setup.

Avant-Guard is proposed to solve data-to-control plane saturation attack [14]. One part of this design, called connection migration, is using the similar approach as SYN cookies. In our work, however, we cannot use such an approach since multiple networks are along the path from sender to receiver and multiple partial round-trips would be necessary for connection setup using two-way communication.

Heuristics to detect attacks based on traffic volume have been proposed to protect from SYN attacks [15] and other network attacks [16,17]. Recently, a similar, volume-based protection approach was proposed for software-defined networks [18]. This work is difficult to implement in practice since it requires that all requesting hosts be categorized based on trust and connection volume thresholds be established a priori.

2.3 Proof of Work

Proof-of-work is a mechanism that established trust explicitly by requiring the requesting entity to perform work, i.e., commit computational resources, to show its sincerity [10]. Proof-of-work has been used before to protect network protocols [19], but may require multiple exchanges between the communicating entities (e.g., to exchange the challenge and to set difficulty set). Our work focuses on a one-way POW protocol that does not require time-consuming parameter exchanges. Proof-of-work has been argued to not work in the context of unsolicited email (spam) reduction since it poses an undue burden on some legitimate senders [20]. As a response, the use of reputation has been proposed to make POW work [21]. More recently, in the context of digital currencies, such as bitcoin [22], POW has become a more widely used approach [23].

CHAPTER 3
PROBLEM STATEMENT

The problem that our work aims to solve is as follows: *Given that an SDN network by design needs to invest considerable resources to set up a network path, how can be ensured that the entity initiating the connections is required to commit a comparable (or higher) amount of resources* before *the connection is established?* Before describing our solution in Chapter 4, we review the operation of SDN, describe attacks on SDN controllers, and formally state the security model underlying our work. Examples discussed in this section are based on the simple topology shown in Figure 3.1.

3.1 Connection Setup in SDN and Attack

A new connection is established in a software-defined network whenever a packet arrives at the network edge with a five-tuple (i.e., source and destination address, source and destination port, and transport layer protocol identifier) that does not

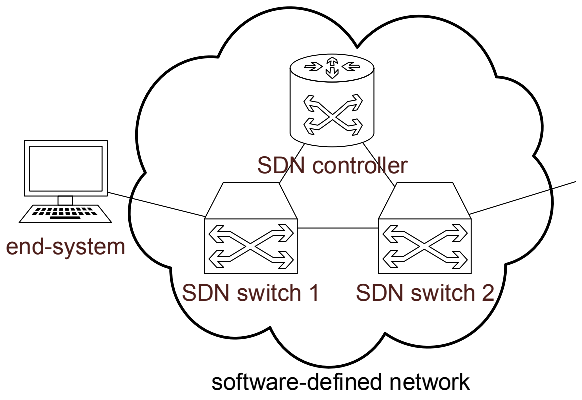


Figure 3.1. Simple topology of example software-defined network.

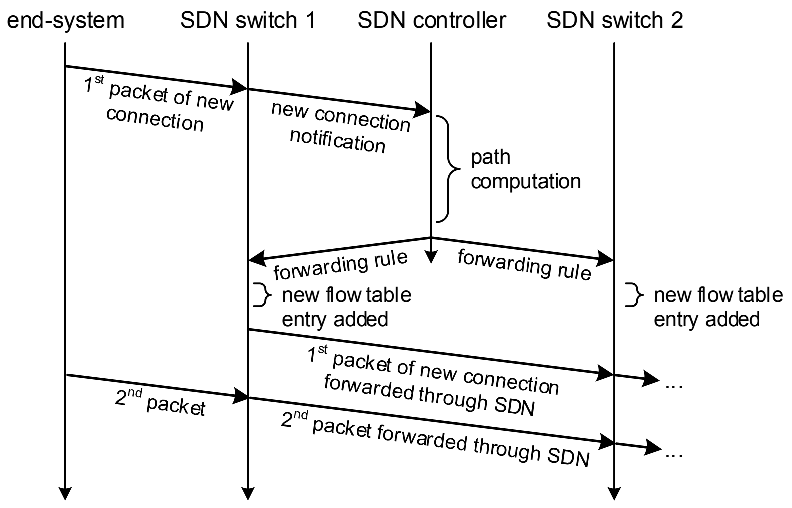

end-system
SDN switch 1
SDN controller
SDN switch 2
1st packet of new connection
new connection notification
path computation
forwarding rule
forwarding rule
new flow table entry added
new flow table entry added
1st packet of new connection forwarded through SDN
2nd packet
2nd packet forwarded through SDN
...
...

Figure 3.2. Space-time diagram of network interactions during connection setup in a software-defined network (topology from Figure 3.1, single SDN network provider, uni-directional communication).

match any rule that has been previously installed in that switch. The steps that are then performed are as follows (see space-time diagram in Figure 3.2):

1. The switch that receives the new packet forwards the five-tuple information to the SDN controller ("new connection notification" in Figure 3.2).

2. The SDN controller computes a suitable path for this new traffic through the SDN. This computation is performed by an SDN application that is accessed through the north-bound interface of the controller. For simplicity of discussion, we consider the SDN controller and SDN application a single unit (since they are implemented on the same physical device). Depending on the algorithm used, the size of the SDN, and the constraints imposed by policies and other existing connections, the path computation may require a considerable amount of processing on the controller.

3. Once the path for the new connection has been determined, the switches along that path are informed ("forwarding rule" in Figure 3.2). Then, a new rule

matching the connection's five-tuple (or a more general rule for forwarding traffic aggregates) is installed in the switch table directing that traffic to the appropriate output port ("new flow table entry added" in Figure 3.2).

4. Once the path through the SDN has been configured, the original packet is forwarded by the switch that initially received it. All later packets of the connection are forwarded directly by the SDN switch without involvement of the SDN controller (until the flow table entry for that connection expires and is removed).

It is apparent from the above listing that the amount of resources committed by the SDN for any new connection involves (1) communication bandwidth between switches and the controller, (2) processing on the controller, and (3) memory in the switch tables. This resource commitment is triggered by merely sending a packet that has new five-tuple values (i.e., a 64-byte packet with an IP and TCP header and no payload).

A malicious attacker can easily launch a denial-of-service (DoS) attack on an SDN by sending a large number of packets with different, new five-tuple values. This type of attack is illustrated in Figure 3.3. Each packet is handled as a new connection and requires the connection setup steps discussed above. Sending attack packets requires very little commitment on the attacker side. However, each packet triggers a considerable resource commitment on the SDN side, thus leading to resource exhaustion. Depending on the configuration of the SDN, this resource exhaustion may occur either on the control links between switches and the controller, on the processor of the SDN controller, or on the state tables in the switches. The effect of the attack is that legitimate connection requests cannot be processed (and their traffic cannot be forwarded).

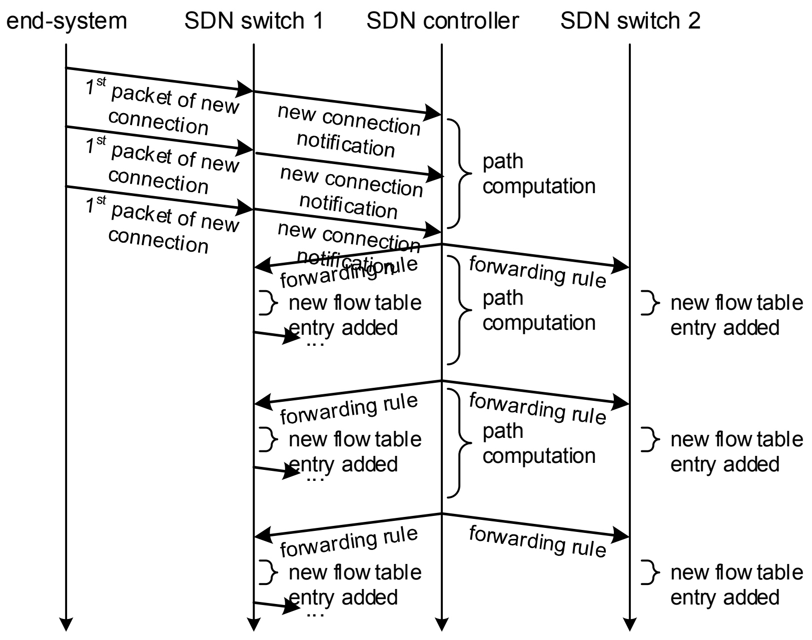


Figure 3.3. Space-time diagram of denial-of-service attack on software-defined network controller (topology from Figure 3.1, single SDN network provider, unidirectional communication, packet forward through SDN omitted for readability).

3.2 Security Model

Before arguing the design to protect SDN controller, we formalize the discussion by defining a security model.

3.2.1 Security Requirements

SR1 An attacker cannot set up a connection through the SDN network without committing computational resources.

SR2 An attacker is not able to use resources committed for a previous or different successfully established connection for a new, different connection.

SR3 An attacker cannot stockpile proofs of work at a slow pace to be used in a sudden attack.

3.2.2 Attacker Capabilities

We assume the attackers have the following capabilities and limitations:

AC1 An attacker can send any type of traffic, including any connection setup requests with real or fake proofs of work.

AC2 An attacker cannot solve a proof of work except by performing the work.

AC3 An attacker cannot predict future values of a true random number generator.

3.3 Performance Requirement

In addition to secure operation, it is also important that CPP achieves efficient operation. We aim to achieve the following performance requirements:

PR1 A correct connection needs to be established in a single pass.

PR2 An incorrect proof of work needs to be identified with little computational resources.

PR3 A correct proof of work needs to require resources that are comparable to (or higher than) those committed by an SDN controller for settings up a connection through the SDN network.

We discuss in Section 5.2.1 how our proposed CPP meets these performance requirements.

CHAPTER 4
CONTROLLER PROTECTION PROTOCOL DESIGN

4.1 Main Idea

The main idea for the Controller Protection Protocol is to use a proof of work during connection setup. This proof of work requires the end-system requesting the connection to commit considerable resources before resources are committed on the side of the SDN controller. When an attacker sends large numbers of connection requests (without committing the resources to include valid proofs of work in each packet), then these packets can be identified and discarded with very little overhead.

The connection setup process based on the Controller Protection Protocol is shown in Figure 4.1. This space time diagram is based on the topology in Figure 3.1 and shows the changes compared to the standard SDN connection setup process shown in Figure 3.2. In CPP, the end-system first computes a proof of work (details on the parameters for this computation are described in Section 4.4). The result from this computation, i.e., the proof of work, is included in the first packet sent by the new connection (e.g., the TCP SYN packet). When the first SDN switch encounters the packet from this new connection, it forwards the connection information, including the proof of work, to the SDN controller. The controller then checks the validity of the proof of work before performing path computation or any other action. In case the proof-of-work validation fails, the controller discards the packet and the connection is not set up (i.e., no path computation takes plane and no forwarding rule is installed in the SDN switches). In case the proof-of-work validation succeeds, the path computation and forwarding rule installation is performed as in conventional

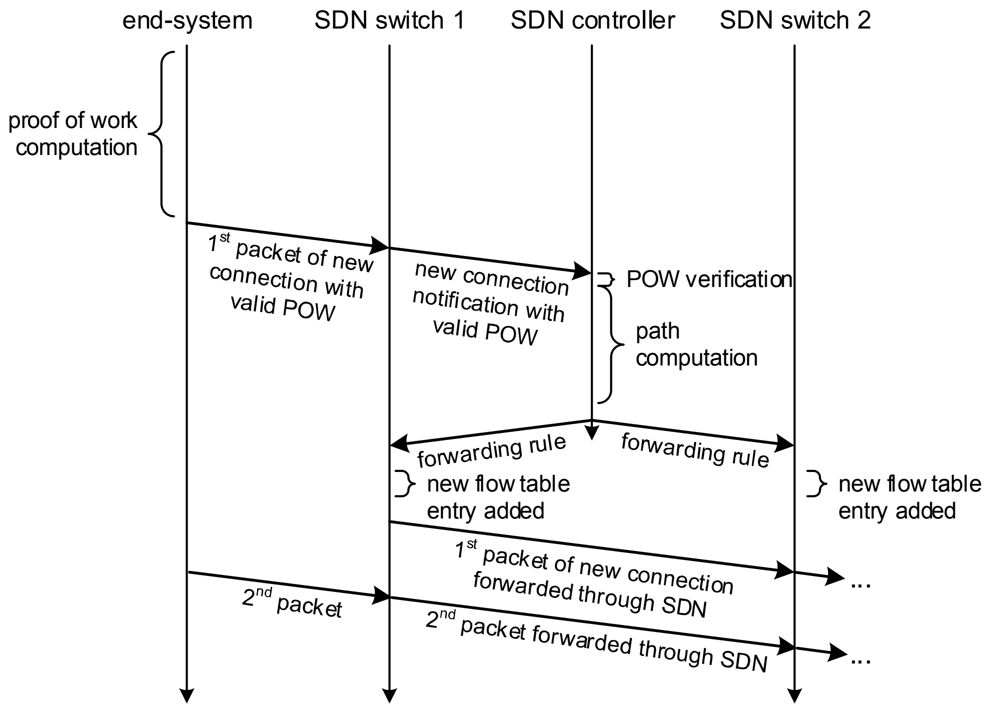

end-system
SDN switch 1
SDN controller
SDN switch 2
proof of work computation
1st packet of new connection with valid POW
new connection notification with valid POW
POW verification
path computation
forwarding rule
forwarding rule
new flow table entry added
new flow table entry added
1st packet of new connection forwarded through SDN
...
2nd packet
2nd packet forwarded through SDN
...

Figure 4.1. Space-time diagram of network interactions during connection setup in a software-defined network using Controller Protection Protocol (topology from Figure 3.1, single SDN network provider, uni-directional communication).

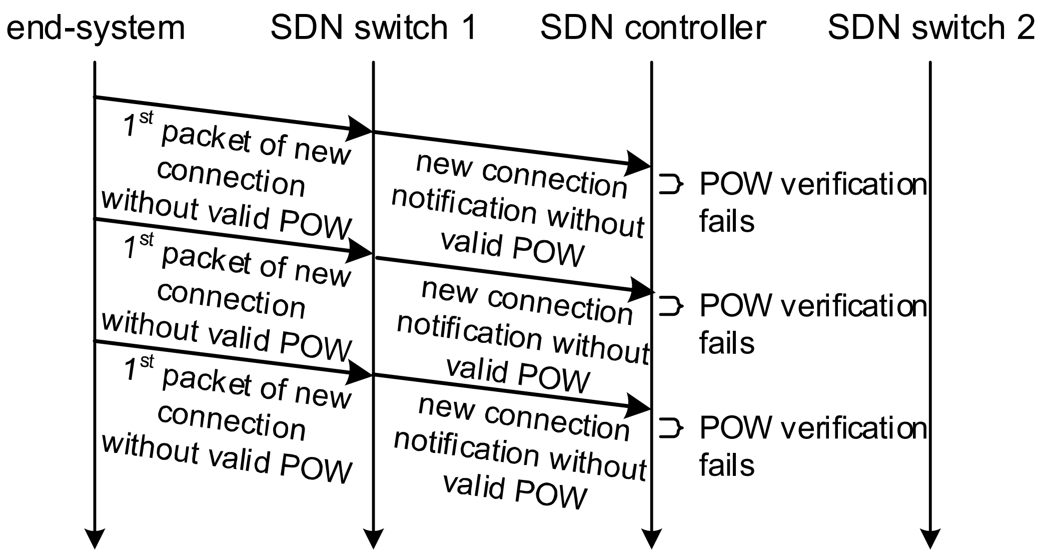

end-system
SDN switch 1
SDN controller
SDN switch 2
1st packet of new connection without valid POW
new connection notification without valid POW
POW verification fails
1st packet of new connection without valid POW
new connection notification without valid POW
POW verification fails
1st packet of new connection without valid POW
new connection notification without valid POW
POW verification fails

Figure 4.2. Space-time diagram of denial-of-service attack on software-defined network controller using Controller Protection Protocol (topology from Figure 3.1, single SDN network provider, uni-directional communication).

SDN. Once the connection has been established, later packets of that connection do not contain a proof of work, but are forwarded by the SDN switches as in conventional SDN.

If an end-system wants to launch a denial-of-service attack on the SDN controller (as shown previously in Figure 3.3), then there are two possible approaches:

- Attack with valid proofs of work: If the attacker includes valid proofs of work, then the SDN controller performs the connection setup as described above. However, the design of the proof of work in CPP is such that generating a valid proof of work requires considerable resources on the end-system initiating the connection request. Therefore, an attacker would need a lot of (costly) computational power to launch a successful attack on the SDN controller.

- Attack with invalid proofs of work: If the attacker does not include valid proofs of work, which is a much cheaper approach, then the SDN controller can detect this lack of a valid proof of work during the verification step. Since the proof-of-work validation fails, no path computation or flow setup resources are committed by the router then. The denial-of-service attack thus fails and only consumes bandwidth resources to forward this initial attack traffic (which can be throttled with conventional DoS protection mechanisms if necessary). This scenario is shown in Figure 4.2.

A variant of this protocol is to check the proof of work on the first SDN switch (i.e., before sending a connection notification to the SDN controller). This variant would reduce intra-SDN communication and thus exhibit even more resilience to attacks. However, most practical SDN switches do not have much compute functionality beyond simply matching of packet headers to flow table entries. Thus, the implementation of this variant is not practical in current SDN. However, if future SDN employ switches with more processing capabilities (or dedicated functions to implement CPP), then the proof-of-work verification can be performed by the SDN switch that encounters the first packet from a new connection.

4.2 System Architecture

One of the key requirements for any connection setup protocol is that it operates in a one-way fashion. An implication of this requirement is that the proof of work needs to be calculated by the end-system initiating the connection before it is known which path the packet takes through the Internet and which network providers are encountered. This lack of knowledge of the path implies that the proof of work cannot be customized to any specific network operator's requirements. Since it also cannot be expected that a network provider wants to trust any other provider to check the proof of work on its behalf, we need to design a proof of work that is acceptable to all network providers.

One approach to using a single proof of work for all providers is to define a set of global parameters for the proof of work necessary for any given new connection.

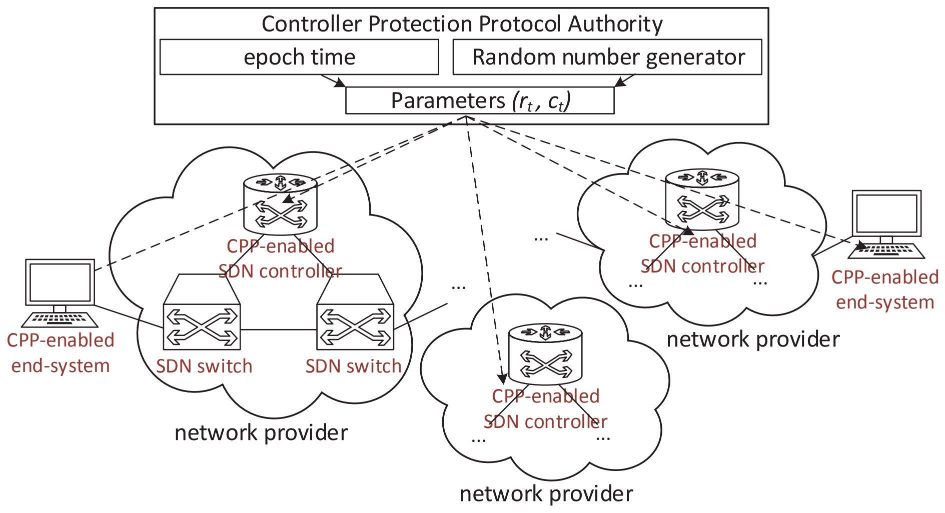


Figure 4.3. System architecture of Controller Protection Protocol with central Controller Protection Protocol Authority that distributes current CPP parameters to SDN controllers for verification of proofs of work in connection requests.

However, if these parameters are fixed (or change in a predictable fashion), then an attacker can stockpile proofs of work, which violates security requirement SR3.

Therefore, we introduce a Controller Protection Protocol Authority (CPPA), which creates and distributes global CPP parameters. These parameters are based on a true random number generator and change over time to ensure that the freshness requirement is met. The CPPA provides the currently active parameter set for pull queries by SDN controllers. Alternatively, the currently active parameter set can be pushed to all SDN controllers through multicast or other content distribution mechanisms. In practice, the Controller Protection Protocol Authority can operate as a logically centralized, but physically distributed system to improve performance, reliability, and resilience to denial-of-service attacks.

The use of a global, logically centralized Controller Protection Protocol Authority may seem like an expensive requirement. However, the CPPA simplifies the problem of needing to establish trust between network providers by acting as trusted intermediary. There are several other, widely deployed network protocol that require central coordination (e.g., Domain Name System (DNS) [24]).

The resulting system architecture for the Controller Protection Protocol is shown in Figure 4.3. The figure shows the the Controller Protection Protocol Authority creates parameters as described below in Section 4.3.1. These parameters are distributed—either through push or through pull mechanisms—to all CPP-enabled components, i.e., end-systems and SDN controllers. (SDN switches do not need to be modified for CPP.) The operation of the Controller Protection Protocol is described in the following section.

4.3 Controller Protection Protocol Operation

This section discusses CPP in detail.

4.3.1 Parameter Distribution

The parameters that are necessary for correct operation of CPP are:

- Random base r: This parameter is a random number generated by the CPPA. This random base is used in the proof-of-work calculation and ensures that proofs of work are only considered valid while this base is active.

- Proof complexity c: This parameter is a number indicating the difficulty of the proof of work (see Section 4.4). Since the computational capabilities of processors continues to grow due to improvements in semiconductor technology (as projected by Moore's law [25]), this parameter enables CPP to adapt over time. (The computational power of SDN controllers also grows with Moore's law, but other resources, such as flow table entries in SDN switches, may not grow as quickly. Therefore, it is convenient to decouple these resources from each other through this explicit parameter.)

The parameter set (r, c) needs to change over time to ensure freshness of proofs and to adapt complexity. We therefore divide time into epochs during which a given parameter set is valid. We define Δt as the epoch duration and epoch n is active during the time interval $[(n-1)\cdot\Delta t, n\cdot\Delta t)$. Any given time t falls into epoch $\lfloor t/(\Delta t)\rfloor + 1$. For simplicity of notation, we denote r_t as parameter r that is valid during the epoch in which time t falls. Similarly, we define c_t. Furthermore, we define r_{t-1} and c_{t-1} as the parameters that were valid in the previous epoch.

The Controller Protection Protocol Authority then distributes (r_t, c_t) during the current epoch. To aid with transient behavior, as we explain below, it is also necessary to distribute the previous set of parameters (r_{t-1}, c_{t-1}). The epoch duration needs to be chosen long enough to ensure that the parameters from the current (and previous) epoch can be distributed throughout the Internet. However, the epoch should not be too long to avoid stockpiling of proofs of work. We envision that Δt values in the order to tens of seconds to minutes are good values for a practical implementation.

4.3.2 Proof of Work Generation

The proof of work that is included in the connection request must be such that it cannot be reused for a different connection request (security requirement SR 2). A straightforward way of ensuring that this requirement is met is to make the proof of work "self-certifying." That is, the flow information itself (i.e., the connection 5-tuple f) is used as a parameter for the proof of work computation. Thus, a proof of work that is valid for the parameters of one connection does not match the parameters of another connection since the connection 5-tuple is different.

In addition, the proof of work is based on the CPPA parameters. Thus, the proof of work for connection f at time t, pow_t^f, is computed by a function p with the following parameters:

$$pow_t^f = p(f, r_t, c_t). \tag{4.1}$$

The details of the function p are provided in Section 4.4.

4.3.3 Proof of Work Verification

The proof of work verification step is a straightforward complement to the proof of work generation step. A function v is used to compute a binary output *valid* indicating if the proof of work is valid. For a matching proof of work, the result is

$$v(f, r_t, c_t, pow_t^f) = true. \tag{4.2}$$

However, if the flow information does not match (e.g., reuse of proof of work $pow_t^{f'}$ from flow f'), then, with high probability,

$$v(f, r_t, c_t, pow_t^{f'}) = false. \tag{4.3}$$

Similarly, the verification step fails with high probability for different CPPA parameters (r', c').

Since information cannot propagate instantaneously in networks, there are situations where epoch parameters have propagated to one part of the network, but not to another. If a connection is initiated from the part of the network with old parameters and reaches the part of the network with new parameters (or vice versa), the verification set fails. To avoid problems during this transient period, the SDN controller performs a second verification step (if the first one fails) with the parameters from the previous epoch: $v(f, r_{t-1}, c_{t-1}, pow_{t-1}^f)$. Thus the window during which a proof of work can be used becomes $2 \cdot \Delta t$ in practice. (There is also a complementary situation where the connection is initiated from the part of the network that has the new parameters. However, if this connection then travels towards the part of the network with the old parameters, it can be assumed that the new parameter values travel equally fast through flooding or multicast and thus arrive before this connection setup request needs to be handled.)

Note that the failure of the verification is based on a probabilistic argument since there is always a chance that an attacker may guess the correct *pow* value. However, as we see below, the chances for a randomly successful attack are exceedingly small and do not pose practical problems.

4.3.4 Multiple Network Providers

A typical end-to-end Internet connection request needs to traverse multiple subnetworks belonging to different network providers. The design of CPP can accommodate such multiple network providers easily. The proof of work sent in the first packet of a connection is independent of a specific provider and the verification step in Equation 4.2 succeeds for any provider.

Since each provider can verify the proof of work independently, there is no need for providers to trust each other to verify traffic on ones behalf. As long as each provider trusts that the Controller Protection Protocol Authority to provide a valid parameter set, no further trust relationships are necessary.

4.4 Proof-of-Work Design

A key question for our design is what proof of work function to use. Dwork and Naor proposed three POW functions based on mathematical hard problems [10]. The first function is finding the square root of an arbitrary x modulo a prime p, which cost at least $\log(p)$ steps to solve and only a simple multiplication to verify. However, to guarantee the existence of the solution, the parameter p and x cannot be chosen randomly, which requires two-way communication to be implemented. The other two functions are based on breaking cryptographic signatures, which may also encounter the same issue as the first function and need two-way communication to be implemented. Abadi proposed a memory-bound POW [26]. The computation time of the function is based on memory latencies, which have much smaller variance than CPU speed. Coelho proposed a protocol based on the Merkle tree [27], which consumes constant effort on solution and verification. However, the verification of this protocol needs multiple cryptographic hash computations, which may be too demanding for an SDN controller.

In CPP, we use an approach similar to previous proofs of work (e.g., such as in bitcoin mining [22]) that can be verified with a single cryptographic hash computation. We require that find an input to an cryptographic hash computation that generates an output starting with a predetermined number of zeros. Since cryptographic hash functions are one-way functions, the entity generating the proof of work has to try by "brute force" to find a suitable input. This search process is on average time consuming and thus requires dedication of computational resources (i.e., "work"). The verification, in contrast, is very simple since the verifier only needs to do a single computations to see if an input (i.e., "proof") yields an output starting with the required number of zeros.

Since we need to customize the proof of work challenge for different connections, epochs, and difficulty levels, we integrate these parameters into the proof of work p (from Equation 4.1) as follows:

$$p(f, r_t, c_t) = w \text{ such that } hash(f \parallel r_t \parallel w) \text{ starts with } c_t \text{ zeros,} \tag{4.4}$$

where '$\|$' denotes a concatenation operation. The work that is performed is to find a suitable w that meets these requirements.

The verification step from Equation 4.2 is then performed as follows:

$$v(f, r_t, c_t, pow_t^f) = \text{if } hash(f \parallel r_t \parallel pow_t^f) \text{ starts with } c_t \text{ zeros.} \tag{4.5}$$

In Section 5.2.1, we show that performing the work is significantly more computationally complex than verifying a proof and that this proportion can be adapted by changing parameter c_t. Note that solving one given proof of work occur very quickly depending on the choice of w in Equation 4.4. However, over multiple proofs (i.e., multiple connection requests), the cost averages out due to the central limit theorem. Also, the probability of simply guessing a correct proof of work is 2^{-c_t} and thus exceedingly small.

4.5 Protocol Details

The proof of work represents additional information that needs to be carried in the first packet of a connection. While it is always possible to add new option fields or design new headers, we have designed an elegant implementation of CPP that does not require any header changes when using Internet Protocol (IP) [28] and Transmission Control Protocol (TCP) [29] headers.

The characteristics of the proof of work discussed above is that it is inherently pseudo-random (non-pseudo-randomness in the proof of work could be exploited to guess a solution more quickly). Therefore, it is possible to utilize existing header fields that use randomized values to carry the proof of work. Specific header fields are:

- IP identification field: This identifier field is 16 bits long and its value is chosen by the sender. The value needs to be unique for a given connection. Since the connection setup packet is the first packet of the connection, any value is acceptable. Thus, this field can carry proof of work information.

- TCP sequence number: This field is 32 bits long and identifies the logical position of the data carried in the payload in the context of the connection stream. The sender chooses a random sequence number at connection setup time and any value is acceptable. Thus, this field can also carry proof of work information.

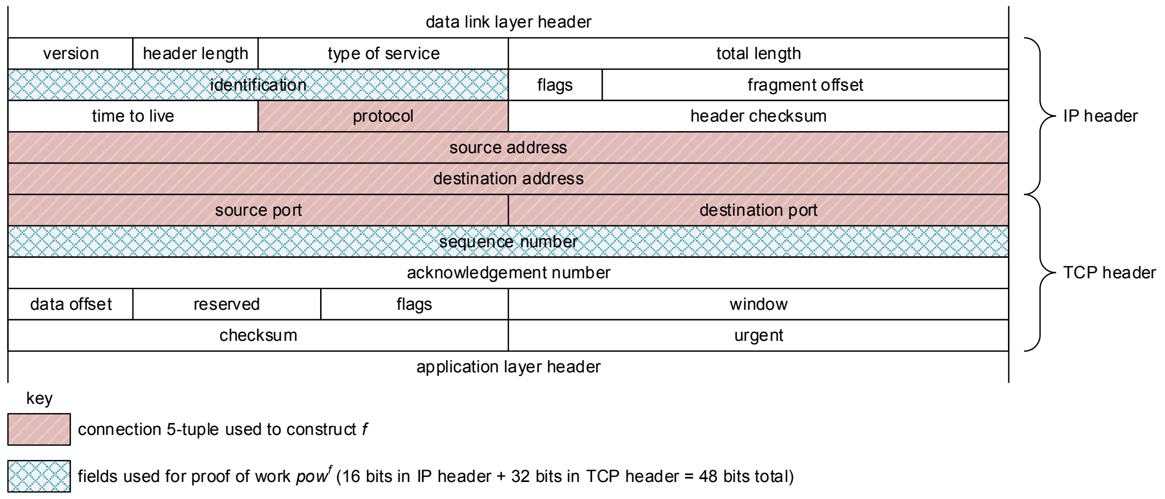


Figure 4.4. Layout of header fields in IP header and TCP header used 5-tuple flow information and proof of work storage.

Figure 4.4 shows an illustration of a TCP/IP header. The fields that are used to determine f are shown, as well as the fields that can carry the proof of work pow^f. In a deployment of CPP, the operating system of the end-system can be modified to use these fields accordingly. For traffic that does not use TCP as transport layer protocol, IP options [28] can be used to carry 32 bits for CPP beyond the 16 bits in the IP identification field discussed above.

CHAPTER 5
EVALUATION

To demonstrate the effectiveness of the Controller Protection Protocol, we discuss how CPP meets the security requirements. Then, we show performance results from an implementation of the proof of work component that demonstrates the computational cost relation between sender and SDN controller for different levels of difficulty.

5.1 Security Evaluation

To argue that CPP meets the security requirements stated in Section 3.2, we revisit each security requirement:

SR1 In CPP, an attacker cannot establish a connection through an SDN since the controller checks for valid proofs of work in all connection requests. An attacker cannot circumvent this requirement by reusing proofs (attacker capability AC2) since proofs of work are parameterized to the flow information and current epoch parameters (Equations 4.1 and 4.4). The probability of a successful guess by the attacker is 2^{-c_t} and thus so small that random proofs of work cannot be used as an attack vector.

SR2 The parametrization of the proof of work based on flow information and current epoch parameters requires commitment of new resources (i.e., a new proof of work computation) for each new connection.

SR3 An attacker cannot use stockpiled proofs of work that are older than two epochs (see Section 4.3.3). The attacker cannot predict future epoch parameters (attacker capability AC3) to pre-calculate proofs. Thus, the largest stockpile is limited by the number of proofs that an attacker can generate and send during this short time.

Based on these arguments, the Controller Protection Protocol meets the security requirements to protect SDN controllers for denial-of-service attacks. One critical aspect in this context is the relationship between the computation time committed by the end-system and the computation time committed by the SDN controller, which we discuss next.

5.2 Proof of Work Implementation

5.2.1 Proof of Work Evaluation

We have experimentally evaluated the proof of work component of CPP for a number of different types of cryptographic hash functions and parameters. The experimental setup uses a 1.8 GHz Intel core i5 processor with 4GB of 1600MHz DDR3 memory. The code is written in C++ on the Xcode platform. The cryptographic hash functions used for this evaluation are SHA-1 (from the Crypto++ library), SHA-2 (SHA-256 and SHA-512 variant, Olivier Gay's implementation) and SHA-3 (Stephan Brumme's implementation).

Figure 5.1 shows the experimental processing times for generation of a proof of work (i.e., Equation 4.4) and verification of a proof of work (i.e., Equation 4.5). As expected, the generation time is a distribution of values based on what value of w is chosen in Equation 4.4. Also, the generation time is significantly larger than the verification time. Thus, the proof of work does cause the necessary commitment of resources on the side of the end-system as stated in security requirement SR1. Also incorrect proofs of work can be detected quickly as stated in performance requirement PR2.

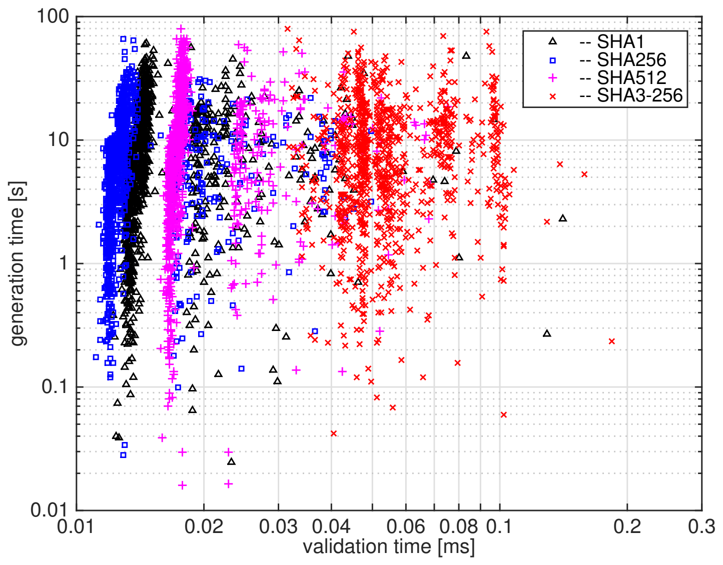


Figure 5.1. Distribution of generation and verification times for proofs of work in CPP (complexity c_t=10 zeroes).

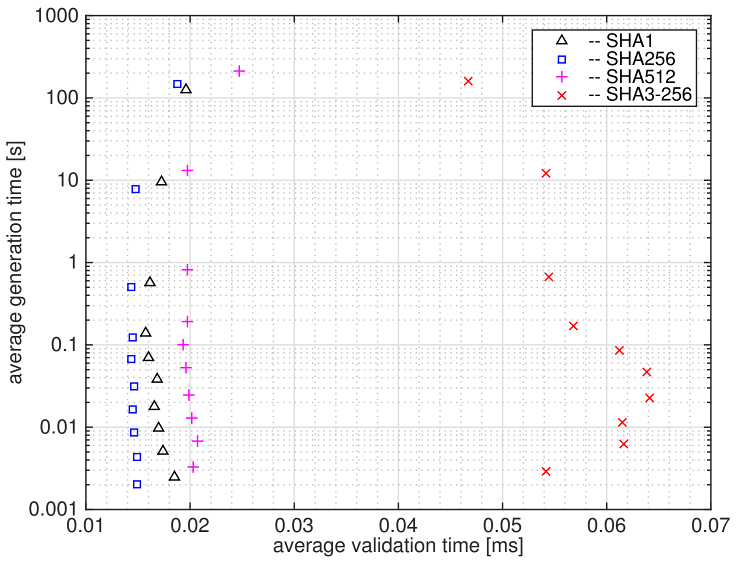


Figure 5.2. Average generation and verification times for proofs of work in CPP for different complexity parameters (complexity c_t=[8, 9, 10, 11, 12, 13, 14, 16, 20, 24] zeroes from bottom to top).

Table 5.1. Average of generation time and verification time of proofs of work for different cryptographic functions and complexity parameters.

hash		complexity c_t					
function	Time (ms)	8	10	12	16	20	24
SHA-1	generation	2.48	9.78	37.73	575.6	9.4k	124.9k
	verification	0.0185	0.0170	0.0169	0.0161	0.0173	0.0196
SHA-256	generation	2.04	8.54	31.71	511.0	7.88k	147.7k
	verification	0.0149	0.0147	0.0147	0.0143	0.0148	0.0188
SHA-512	generation	3.23	12.99	53.84	828.3	13.3k	213.4k
	verification	0.0203	0.0201	0.0196	0.0197	0.0198	0.0247
SHA3-256	generation	2.95	11.45	47.59	669.49	12.2k	160.2k
	verification	0.0542	0.0614	0.0638	0.0544	0.0542	0.0467

Table 5.2. Average ratio of generation time and verification time of proofs of work for different cryptographic functions and complexity parameters.

hash	complexity c_t					
function	8	10	12	16	20	24
SHA-1	134	575	2.23k	35.7k	544.8k	6.37M
SHA-256	136	581	2.16k	35.7k	543.5k	7.86M
SHA-512	159	646	2.75k	42.0k	670.6k	8.64M
SHA3-256	54	186	746	12.3k	226.0k	3.43M

To explore this ratio of computation time committed during proof of work generation and proof of work verification, Figure 5.2 shows these times for a number of different complexity value. The genaration and verification time of proof of work with some complexity parameter values is shown in Table 5.1, and the ratio between generation time and verification time is summarized in Table 5.2. These results show that, depending on the choice of complexity parameter, the resource commitment on the end-system can be a few hundred times the cost of verifying the proof of work or many million times. Thus, performance requirement PR3 is met. Thus, CPP can be adapted to the necessary ratio of resource commitment. Since the complexity parameter c_t is distributed by the Controller Protection Protocol Authority, such adaptation can be implemented with ease.

5.2.2 Proof-of-work Implementation on End-System

Table 5.1 and table 5.2 shows the average time of verification and generation time of different cryptographic functions. The results show that SHA3 has the lowest ratio but the verification time is more than twice as the others. Using SHA3 to implement CPPA will cost more on controller but the same as other cryptographic hash functions at the attacker side. Thus, we will not use SHA3 to implement the protocol on host. Comparing the other three, SHA-256 has the lowest verification time but has a raletively high ratio. Therefore, we use SHA-256 to implement the protocol on end-system.

To implement proof of work on end-system, we need to change information of each SYN packet. According to the discription in Section 4.5, we could choose IP identifier number or TCP sequence number. We use TCP sequence number to contain proofs of work.

The first idea is shown in Figure 5.3. The application needs to capture the outgoing packets, recognise the SYN packets and change the TCP sequence number to a valid proof of work. Then, it remenbers the difference between the original sequence number and proof of work and pairs it with the flow information. When receiving packets, the application needs to look it up from the table according to the flow information and then change back the relative acknowledge number according to the records. Implementtation of this application is difficult since the system is not allowed to change the TCP sequence number outside the kernel. So we need to change the IP/TCP stack in kernel to implement the protocol.

The experiment is taken on Ubuntu 16.04 with kernel version of 4.4.1. By hacking the kernel, we don't need to record the original TCP sequence number since we directly change the initail sequence number (ISP).

According to RFC6528, the initial sequence number is generated with the expresstion: ISN = M + F(localip, localport, remoteip, remoteport, secretkey), where

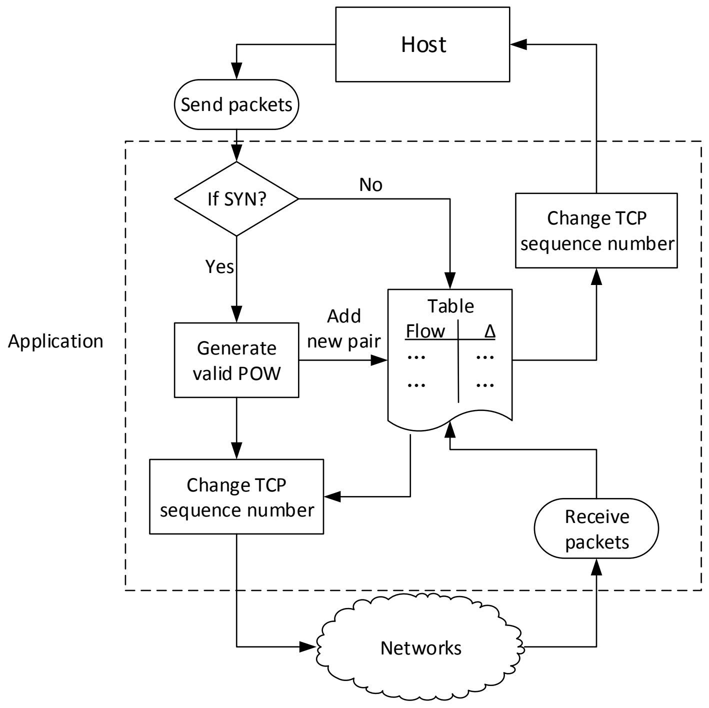


Figure 5.3. Initial idea of implementation for CPPA protocol on host

M is the 4 microsecond timer, and F() is a pseudorandom function (PRF) of the connection-id. In kernel 4.4.1, the F() is MD5 algorithm. The sequece number is generated in file net/core/secure_seq.c. We ignore the random base r_t in Equation 4.4 because we don't implement the CPPA in this experiment. The steps of generating a valid proof of work is shown in Figure 5.4.

1. Generate initial number i according to RFC6528.
2. Calculate SHA-256 of source IP and port, destination IP and port, and i from step 1.
3. Check if the digest of SHA-256 meets the complexity parameter c_t.
4. If the digest meets the requirement, use i as the initial sequence number. If not, increase i by 1 and repeat step 2 - 4.

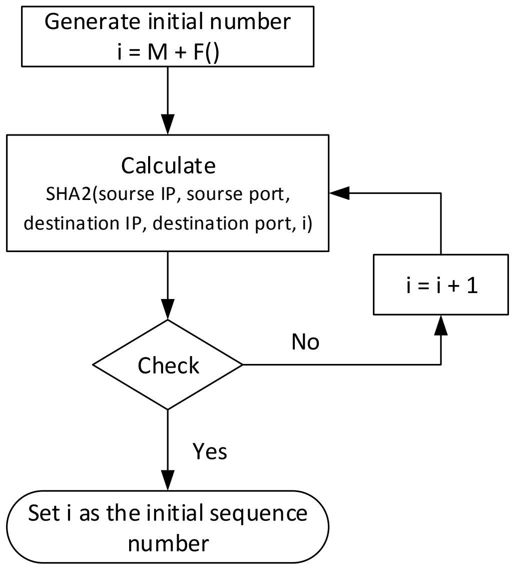
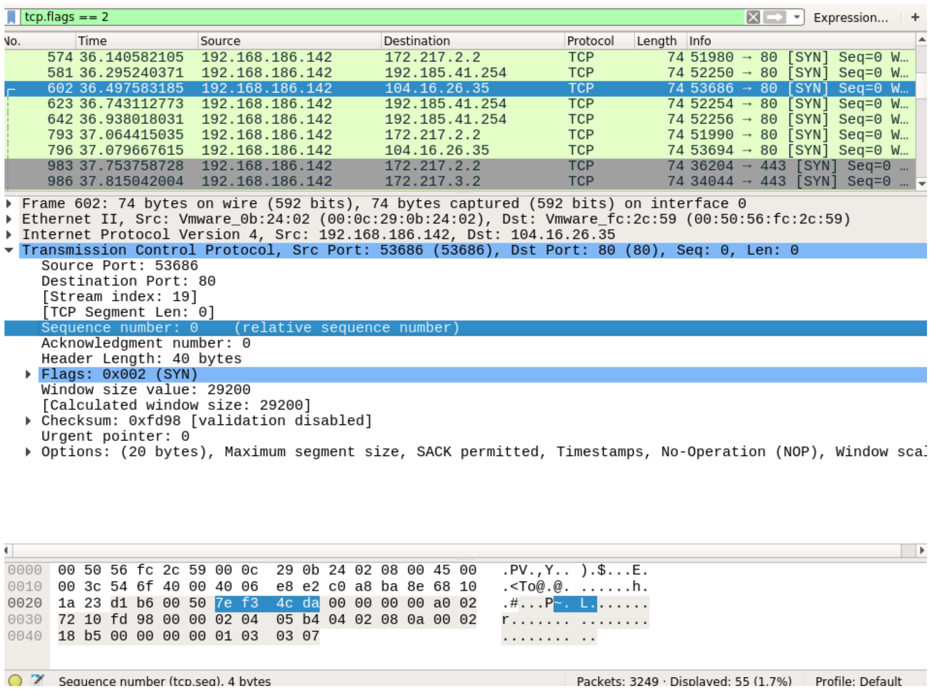

Generate initial number
i = M + F()
Calculate
SHA2(sourse IP, sourse port, destination IP, destination port, i)
i = i + 1
Check
No
Yes
Set i as the initial sequence number

Figure 5.4. Generating proof of work in Ubuntu

We use wireshark to capture the packets and check if the system changes the initial sequence number of every SYN packet. In Figure 5.5, we can see the information of highlighted packet. We use hexadecimal number to be the input of SHA-256 and set the complexity parameter $c = 16$, which means the requirement is 16 bits of zeros. In Figure 5.5, the source IP is $(c0a8ba8e)_{16}$, the souce port is $(d1b6)_{16}$, the destination IP and port are $(68101a23)_{16}$ and $(50)_{16}$ and the sequence number is $(7ef34cda)_{16}$. Figure 5.6 shows the result of SHA-256. The output of SHA-256 is beginning with 16 bits of zeros in binary form (4 zeros in hexadecimal number).

We only check the functionality of this kernel such as browsing webpages, watching online videos and making VoIP call. To measure the delay time of this implementation can be a further work of this experiment.



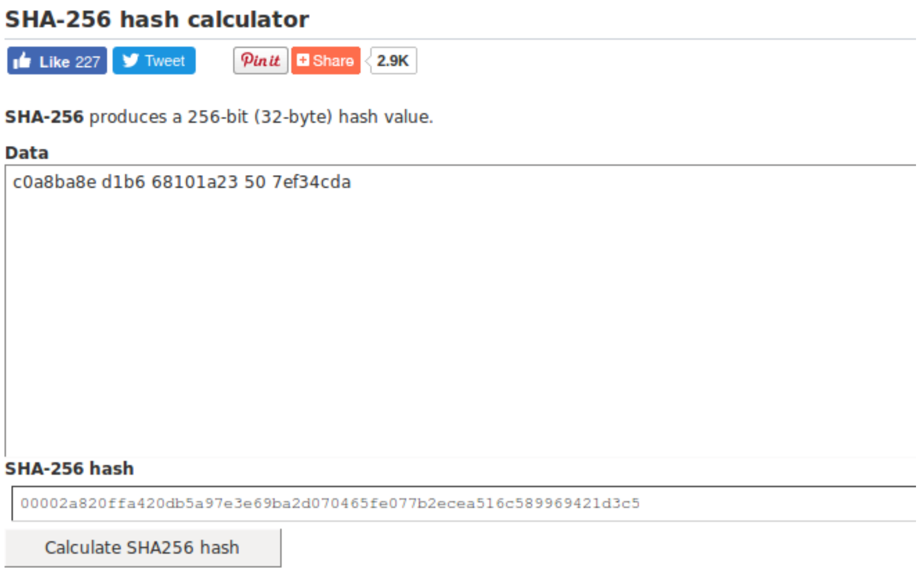
```
tcp.flags == 2
No.   Time          Source           Destination      Protocol  Length Info
  574 36.140582105  192.168.186.142  172.217.2.2      TCP          74 51980 → 80 [SYN] Seq=0 W…
  581 36.295240371  192.168.186.142  192.185.41.254   TCP          74 52250 → 80 [SYN] Seq=0 W…
  602 36.497583185  192.168.186.142  104.16.26.35     TCP          74 53686 → 80 [SYN] Seq=0 W…
  623 36.743112773  192.168.186.142  192.185.41.254   TCP          74 52254 → 80 [SYN] Seq=0 W…
  642 36.938018031  192.168.186.142  192.185.41.254   TCP          74 52256 → 80 [SYN] Seq=0 W…
  793 37.064415035  192.168.186.142  172.217.2.2      TCP          74 51990 → 80 [SYN] Seq=0 W…
  796 37.079667615  192.168.186.142  104.16.26.35     TCP          74 53694 → 80 [SYN] Seq=0 W…
  983 37.753758728  192.168.186.142  172.217.2.2      TCP          74 36204 → 443 [SYN] Seq=0 …
  986 37.815042004  192.168.186.142  172.217.3.2      TCP          74 34044 → 443 [SYN] Seq=0 …

▸ Frame 602: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: Vmware_0b:24:02 (00:0c:29:0b:24:02), Dst: Vmware_fc:2c:59 (00:50:56:fc:2c:59)
▸ Internet Protocol Version 4, Src: 192.168.186.142, Dst: 104.16.26.35
▾ Transmission Control Protocol, Src Port: 53686 (53686), Dst Port: 80 (80), Seq: 0, Len: 0
    Source Port: 53686
    Destination Port: 80
    [Stream index: 19]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 40 bytes
  ▸ Flags: 0x002 (SYN)
    Window size value: 29200
    [Calculated window size: 29200]
  ▸ Checksum: 0xfd98 [validation disabled]
    Urgent pointer: 0
  ▸ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window sca

0000  00 50 56 fc 2c 59 00 0c  29 0b 24 02 08 00 45 00   .PV.,Y.. ).$...E.
0010  00 3c 54 6f 40 00 40 06  e8 e2 c0 a8 ba 8e 68 10   .<To@.@. ......h.
0020  1a 23 d1 b6 00 50 7e f3  4c da 00 00 00 00 a0 02   .#...P~. L.......
0030  72 10 fd 98 00 00 02 04  05 b4 04 02 08 0a 00 02   r....... ........
0040  18 b5 00 00 00 00 01 03  03 07                     ........ ..

Sequence number (tcp.seq), 4 bytes        Packets: 3249 · Displayed: 55 (1.7%)    Profile: Default
```

Figure 5.5. Wireshark in Ubuntu



SHA-256 hash calculator

SHA-256 produces a 256-bit (32-byte) hash value.

Data

```
c0a8ba8e d1b6 68101a23 50 7ef34cda
```

SHA-256 hash

```
00002a820ffa420db5a97e3e69ba2d070465fe077b2ecea516c589969421d3c5
```

Figure 5.6. Verify the proof-of-work generated in Ubuntu

5.3 SDN Prototype Evaluation

5.3.1 Processing time of controller

We have extended a POX SDN controller to implement CPP and check incoming connection requests for valid POWs. We created an OpenFlow network environment, including two virtual switches and two hosts using Mininet in Virtual box with one core of 1.8 GHz Intel core i5 processor. The network traffic with both valid and invalid proofs of work is generated by the Python API provided in Mininet.

Table 5.3. Measurement results of processing times on SDN controller. Proof of work uses SHA-256 with a complexity of c_t=12 bits of zeros. The regular SDN controller cannot distinguish between connection requests with valid or invalid POWs. Results are averages with standard deviation over 1000 connection request measurements.

	Regular SDN Controller		CPP-enabled SDN Controller	
Connection request with no POW or with invalid POW	reception (μs)	150.97 ± 64.75	Reception (μs)	137.09 ± 56.37
	no POW check (μs)	0.41 ± 0.55	POW check (μs)	41.53 ± 16.67
	path computation (μs)	459.94 ± 255.94	connection drop (μs)	1.62 ± 1.18
	Total (μs)	611.32 ± 284.76	Total (μs)	180.24 ± 67.144
Connection request with valid POW	reception (μs)	141.45 ± 47.39	reception (μs)	156.51 ± 46.406
	no POW check (μs)	0.36 ± 0.74	POW check (μs)	52.90 ± 15.41
	path computation (μs)	489.34 ± 172.92	path computation (μs)	488.99 ± 155.26
	Total (μs)	631.15 ± 205.53	Total (μs)	698.40 ± 197.59

Table 5.3 shows the measurement results from evaluating connection setup times on a regular SDN controller and on a CPP-enabled SDN controller. The results show the processing time for packets with and without proofs of work. The regular SDN controller does not perform a POW check and thus performs costly path computation for all connection requests, totaling a connection processing time of over 600μs. The CPP-enabled SDN controller requires around 50μs more processing time for valid connection due to the POW check. However, invalid connection requests, i.e., those without valid POW, that may have been sent by a DoS attacker, can be processed in less than 200μs. In particular, the time after detecting that the POW is not valid is less than 2μs compared to around 500 μs in a conventional controller. Thus, the CPP-enable controller is able to protect its computational resource from DoS attacks. In addition, but not shown here, no resources are used to communicate with switches

or to install flow rules in switch tables after a connection is dropped due to an invalid POW.

These results show that Controller Protection Protocol achieves the desired properties and reduces the workload significantly on SDN controllers when DoS attack connection requests without valid proofs of work are sent.

5.3.2 Performance of controller under attacks in SDN

In this experiment, we created an OpenFlow network environment, including one opneflow virtual switch and three hosts using GENI resourses. The topology is shown in Figure 5.7. Host 1 launches attacks towards host 3 and host 2 is a normal user communicating with host 3. The SYN packets attack is generated using hping3 tool. The network traffic with valid proofs of work is generated using Python API.

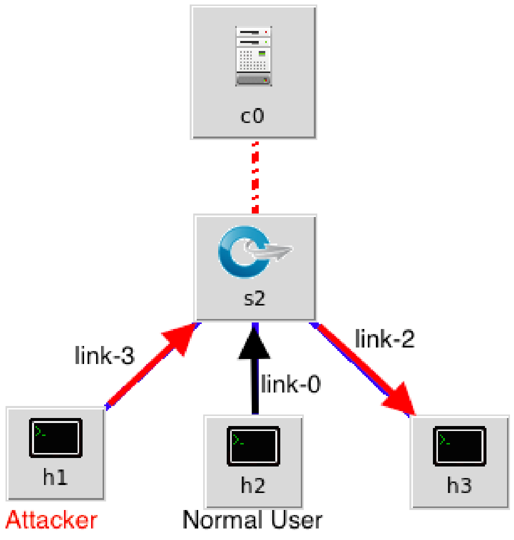


Figure 5.7. Topology of experiment on GENI

Figure 5.8 to Figure 5.14 shows the traffic flows on links from host side in the network. The packets in outgoing traffic are the SYN packets sent from the host, which represents the new connection request the host wants to setup. The packets in incomming traffic contains SYN-ACK packets. Because the SYN packets are generated manually with no actual service request, the SYN-ACK packets also carry FIN flag to terminate the following packets of this connection. Therefore, the incomming traffic in these figures only contains SYN-ACK packets, which means the number of packets comming into the host is the number of connections set up successfully.

Figure 5.8 to Figure 5.11 shows the traffic under the attack with different rate in network using regular controller. The number of incomming packets in host 2 decreases at the point host 1 lauches the attack. In Figure 5.9, the incomming traffic at host 2 is under 10 packet/s and in Figure 5.10, the incomming traffic is under 5 packet/s, which means over 90% of connections cannot be established under 3000 packet/s attack.

Figure 5.12 and Figure 5.13 shows the traffic under the attacks with 1000 packet/s and 10000 packet/s towards CPP-enabled SDN controller. The incomming traffic in host 2 is not affected when attack launches. When host 1 launches attack with 100,000 packet/s, shown in Figure 5.14, over 10 connections per second can be set up under attack.

These results show that Controller Protection Protocol could increase the performance of controllers under attacks, which make the attackers consume more resourses to launch a successful DDoS attack.

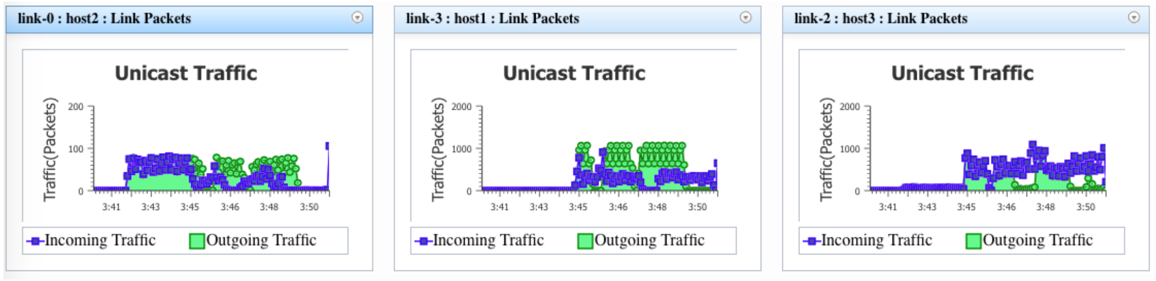


Figure 5.8. Attacks to regular SDN controller (attack with 1000 packets/second)

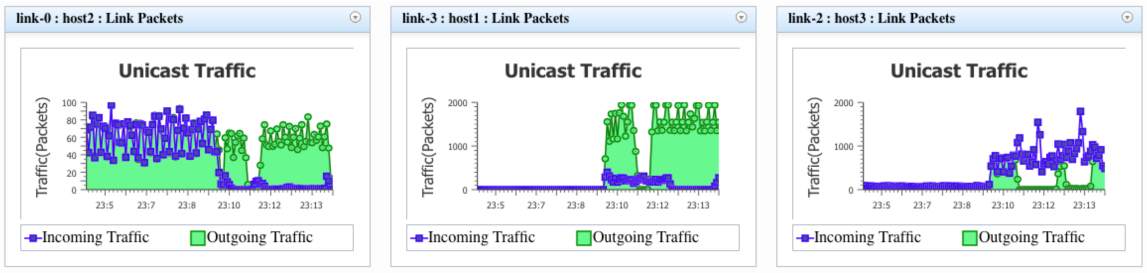


Figure 5.9. Attacks to regular SDN controller (attack with 2000 packets/second)

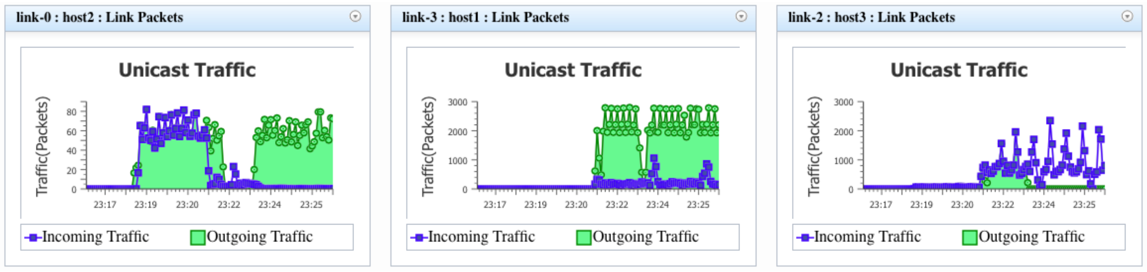


Figure 5.10. Attacks to regular SDN controller (attack with 3000 packets/second)

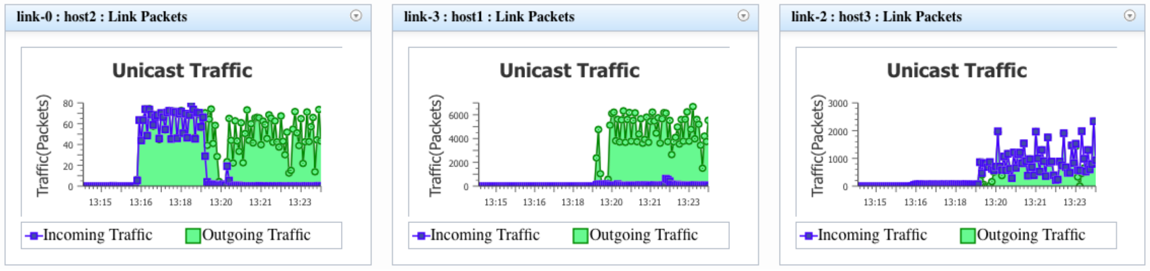


Figure 5.11. Attacks to regular SDN controller (attack with 10000 packets/second)

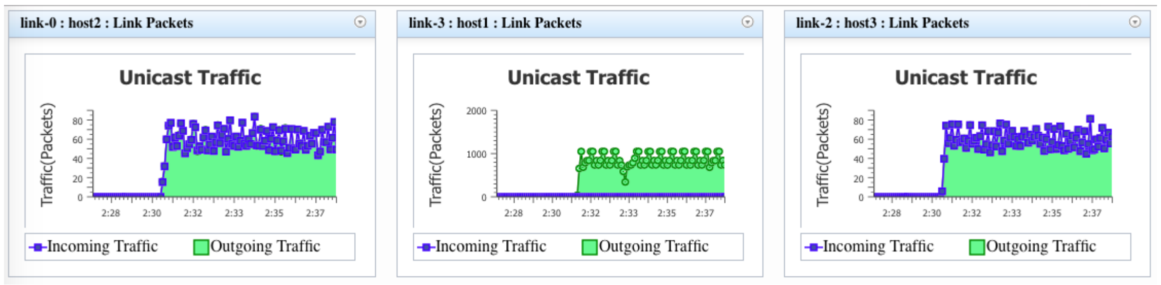


Figure 5.12. Attack to CPP-enabled SDN controller (attack with 1000 packets/second)

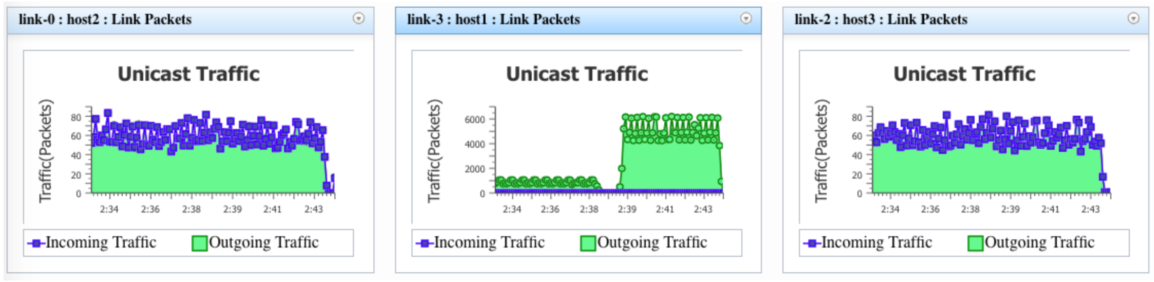


Figure 5.13. Attack to CPP-enabled SDN controller(attacks with 10000 packets/second)

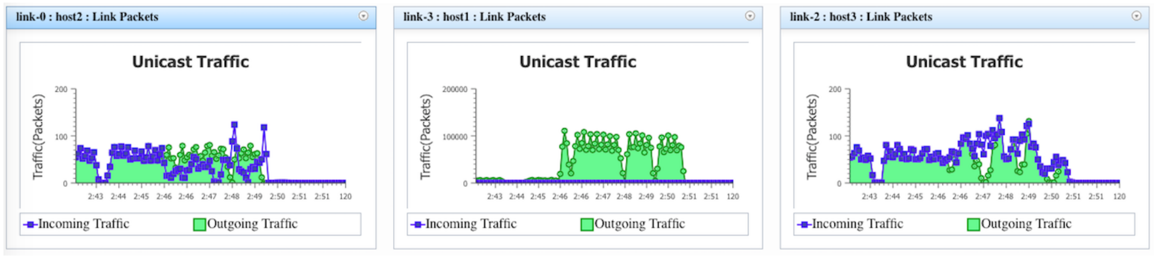


Figure 5.14. Flood attack to CPP-enabled SDN controller

CHAPTER 6
SUMMARY AND FUTURE WORK

Our work addresses the problem of how to protect controllers in software-defined networks from denial-of-service attacks. Since the controller needs to commit a considerable amount of computation, communication, and memory resources in the SDN for each new connection, an attacker can easily cause such denial-of-service by flooding the controller with new connection requests. Our Controller Protection Protocol requires that new connection requests contain a proof of work that demonstrates that the end-system requesting the new connection has already committed considerable computational resources. As a result, an attacker would need to commit prohibitive amounts of computational power for a successful attack.

Based on the security model presented in the thesis and the experimental results, we are able to argue that CPP meets the requirements to protect SDN controllers effectively.

Our Controller Protection Protocol also fulfills practical deployment aspects, such as operating in a one-way mode, not requiring trust among network providers, and being able to adapt the proof of work dynamically to different levels of complexity. In addition, CPP can be implemented in the existing TCP/IP protocol stack without requiring any new headers or header options.

The Controller Protection Protocol Authority, which is a source of a random number (i.e., parameter r_t) that changes over time may be valuable to other protocols that require "freshness." One example is to use this information to avoid replay attacks.

Thus, the presented work may be useful beyond its immediate application to protect SDN controllers from denial of service attacks.

BIBLIOGRAPHY

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, Kyoto, Japan, Aug. 2007, pp. 1–12.

[3] "SDN defined," https://www.opennetworking.org/sdn-resources/sdn-definition.

[4] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, Mar. 2013.

[5] J. Batalle, J. Ferrer Riera, E. Escalona, and J. A. Garcia-Espin, "On the implementation of NFV over an OpenFlow infrastructure: Routing function virtualization," in *Proc. of Workshop on Software Defined Networks for Future Networks and Services (SDN4FNS)*, Trento, Italy, Nov. 2013, pp. 1–6.

[6] X. Chen, T. Wolf, J. Griffioen, O. Ascigil, R. Dutta, G. Rouskas, S. Bhat, I. Baldin, and K. Calvert, "Design of a protocol to enable economic transactions for network services," in *Proc. of IEEE International Conference on Communications (ICC)*, London, UK, Jun. 2015.

[7] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proc. of 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, San Jose, CA, Apr. 2012.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Oct. 2010, pp. 1–6.

[9] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," in *Proc. of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Hong Kong, China, Aug. 2013, pp. 7–12.

[10] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Proc. of Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, E. F. Brickell, Ed. Springer, 1993, vol. 740, pp. 139–147.

[11] S. Shin and G. Gu, "Attacking software-defined networking: a feasibility study," in *Proc. of 2nd ACM SIGCOMM Workshop on Hot topics in Software-Defined Networking*, Hong Kong, China, 2013, pp. 165–166.

[12] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, Apr. 2004.

[13] D. J. Bernstein, "SYN cookies," 1996, http://cr.yp.to/syncookies.html.

[14] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Nov. 2013, pp. 413–427.

[15] V. A. Siris and F. Papagalou, "Application of anomaly detection algorithms for detecting SYN flooding attacks," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, Dallas, TX, Nov. 2004, pp. 2050–2054.

[16] J. M. Estevez-Tapiador, P. Garcia-Teodoro, and J. E. Diaz-Verdejo, "Anomaly detection methods in wired networks: a survey and taxonomy," *Computer Communications*, vol. 27, no. 16, pp. 1569–1584, Oct. 2004.

[17] H. Wang, L. Xu, and G. Gu, "FloodGuard: A dos attack prevention extension in software-defined network," in *Proc. of 45th IEEE/IFIP International Conference on Dependable Systems and Networks*, Rio de Janeiro, Brazil, Jun. 2015, pp. 239–250.

[18] L. Wei and C. Fung, "FlowRanger: A request prioritizing algorithm for controller DoS attacks in software defined networks," in *Proc. of IEEE International Conference on Communications (ICC)*, London, UK, Jun. 2015.

[19] E. Kaiser and W.-c. Feng, "mod_kaPoW: Protecting the web with transparent proof-of-work," in *Proc. of IEEE INFOCOM Workshops*, Phoenix, AZ, Apr. 2008, pp. 1–6.

[20] B. Laurie and R. Clayton, "?proof-of-work? proves not to work," in *Proc. of Third Annual Workshop on Economics of Information Security (WEIS)*, Minneapolis, MN, May 2004.

[21] D. Liu and L. J. Camp, "Proof of work can work," in *Proc. of Fifth Annual Workshop on Economics of Information Security (WEIS)*, Cambridge, UK, Jun. 2006.

[22] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, http://bitcoin.org/bitcoin.pdf, 2008.

[23] J. Becker, D. Breuker, T. Heide, J. Holler, H.-P. Rauer, and R. Böhme, "Can we afford integrity by proof-of-work? scenarios inspired by the bitcoin currency," in *The Economics of Information Security and Privacy*, R. Böhme, Ed. Springer Berlin Heidelberg, 2013, pp. 135–156.

[24] P. Mockapetris, "Domain names - implementation and specification," Network Working Group, RFC 1035, Nov. 1987.

[25] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[26] M. Abadi, M. Burrows, M. S. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," *ACM Transactions on Internet Technology*, vol. 5, no. 2, pp. 299–327, May 2005.

[27] F. Coelho, "An (almost) constant-effort solution-verification proof-of-work protocol based on Merkle trees," in *Proc. of the Cryptology in Africa 1st International Conference on Progress in Cryptology (AFRICACRYPT)*, Casablanca, Morocco, 2008, pp. 80–93.

[28] J. Postel, "Internet Protocol," Information Sciences Institute, RFC 791, Sep. 1981.

[29] J. Postel, "Transmission Control Protocol," Information Sciences Institute, RFC 793, Sep. 1981.