SOFTWARE-DEFINED NETWORKING BASED TESTBED FOR EVALUATING CYBER-ATTACKS AND DEFENSE STRATEGIES USING FPGA

by

Arjun Kumar

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2015

Approved by:

____

Dr. Ron Sass

____

Dr. Asis Nasipuri

____

Dr. James M. Conrad

© 2015
Arjun Kumar
ALL RIGHTS RESERVED

ABSTRACT

ARJUN KUMAR. Software-defined networking based testbed for evaluating cyber-attacks and defense strategies using FPGA.
(Under the direction of DR. RON SASS)

Networks inherently facilitate interconnections of resources and people, even if they are remotely located. While this easily facilitates the communication between physically separated devices, it poses a huge vulnerability. Any remotely connected resource now has access to the network, making network security a necessity. While implementing a defense strategy, it is important for the network administrator to understand the impact an attack could have on the network. Emulators help in understanding how an unforeseen attack can handicap a network. Most of the emulators today are software emulators. While software emulators help to a certain extent in this regard by emulating small scale networks, it is hard to emulate a large scale attack on a large scale network. This thesis aims to design a hardware based solution to this problem which not only overcomes the major drawback of software emulators, but is also controlled, repeatable and fast.

ACKNOWLEDGMENTS

I would like to thank Dr. Ron Sass, for his immense support as my mentor, teacher and advisor. His encouragement and guidance has helped me in each step of my Master's degree. I greatly appreciate all the hard work he has put in to me. I would like to thank the members of my thesis committee, Dr. Asis Nasipuri and Dr. James M. Conrad, for their support and guidance. I am also grateful to Dr. Ehab Al-Shaer, for sharing his expertise which aided me in my thesis work. I would like to extend my gratitude towards to my fellow labmates for all their help in the progression of the thesis. Finally, I would like to thank my family and friends who have always been supportive of my work.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

LIST OF ABBREVIATIONS x

CHAPTER 1: INTRODUCTION 1

1.1 Thesis Statement 3

1.2 Contributions 3

1.3 Outline 4

CHAPTER 2: BACKGROUND 5

2.1 Traditional Networks 5

2.2 The SDN-way 6

2.2.1 SDN Architecture 7

2.2.2 OpenFlow 8

2.3 SDN Network Security — Pros and Cons 8

2.4 Existing SDN Simulators/Emulators 10

2.4.1 Mininet 10

2.4.2 Estinet 10

2.4.3 NS3 11

2.5 FPGA and SDN 11

CHAPTER 3: IMPLEMENTATION 13

3.1 SDN Switch 13

3.1.1 Ingress Multiplexer/Arbiter 16

3.1.2 VLAN Remover 17

3.1.3 Output Lookup Block 17

3.1.3.1 Header Parser 17

3.1.3.2 Exact Match 18

3.1.3.3 Output Processing Logic (OPL) 19
3.1.4 VLAN Adder 20
3.1.5 Egress Demultiplexer 20
3.2 Register Interface 20
3.3 Transmission of Flows Using Microblaze 22
3.4 Emulator Software 23
3.4.1 SDN Controller Software 24
3.4.2 Attack Packets Generator 26
CHAPTER 4: EVALUATION 30
4.1 Evaluation Metrics 30
4.2 Experimental Setup 31
4.2.1 Time to Setup and Execute Experiment 32
4.2.2 Post-processing Scripts 33
4.3 Resource Utilization 33
4.4 Performance 34
4.4.1 Mininet HiFi Performance 35
4.4.2 Performance of the Current Design 36
4.5 Repeatability 37
4.6 Evaluation of the Network Emulator 37
4.6.1 Ping of Death 37
4.6.2 Crossfire Attack Emulation 38
4.6.3 Route Isolation 40
4.6.4 Random Route Mutation 46
4.7 Analysis 49
CHAPTER 5: CONCLUSION 52
5.1 Future Work 52
REFERENCES 54

LIST OF TABLES

TABLE 3.1: Flow table header fields 18

TABLE 3.2: Packet action fields 19

TABLE 3.3: List of counters 22

TABLE 3.4: Event list entry 26

TABLE 4.1: Resource utilization 33

TABLE 4.2: Utilization percentage 34

TABLE 4.3: Line rate for different packet lengths in Mininet HiFi 36

TABLE 4.4: Line rate for different packet lengths 36

TABLE 4.5: Speed up w.r.t Mininet Hifi 37

LIST OF FIGURES

FIGURE 2.1: SDN Architecture 9

FIGURE 3.1: Emulation Architecture 14

FIGURE 3.2: SDN Switch 15

FIGURE 3.3: Sample packet flow 16

FIGURE 3.4: Output Lookup Table 17

FIGURE 3.5: Register interface 21

FIGURE 3.6: Microblaze — Packet drivers interface design 23

FIGURE 3.7: Software Architecture 24

FIGURE 3.8: Controller event handling structure 27

FIGURE 3.9: Flow header calculation 27

FIGURE 3.10: Attack emulation handling structure 28

FIGURE 4.1: Performance measure experimental setup 34

FIGURE 4.2: Ping flooding attack 38

FIGURE 4.3: Ping flooding attack mitigation 38

FIGURE 4.4: Single link crossfire attack 39

FIGURE 4.5: Crossfire attack experiment setup 40

FIGURE 4.6: Crossfire attack: User1 average byte rate 41

FIGURE 4.7: Crossfire attack: Bot group1 average byte rate 41

FIGURE 4.8: Crossfire attack: User2 average byte rate 41

FIGURE 4.9: Crossfire attack: Bot group 3 average byte rate 42

FIGURE 4.10: Crossfire attack: User3 average byte rate 42

FIGURE 4.11: Crossfire attack: Pkt drop in switch 2 42

FIGURE 4.12: Crossfire attack: Pkt drop in switch 3 43

FIGURE 4.13: Route isolation experiment setup 44

FIGURE 4.14: Route isolation: FLOW0 44

FIGURE 4.15: Route isolation: FLOW2 45

FIGURE 4.16: Route isolation: FLOW1 and FLOW6 45

FIGURE 4.17: Route isolation: FLOW3 and FLOW7 45

FIGURE 4.18: Route isolation: Packet Drops 46

FIGURE 4.19: Random route mutation (RRM) experiment setup 47

FIGURE 4.20: RRM: UserA throughput 48

FIGURE 4.21: RRM: UserB throughput 48

FIGURE 4.22: RRM: BotA throughput 48

FIGURE 4.23: RRM: BotB throughput 49

LIST OF ABBREVIATIONS

API	application programming interface
ASIC	application - specific integrated circuit
BRAM	block random access memory
CLI	command line interpreter
CRC	Cyclic redundancy code
DRAM	dynamic random access memory
EDK	Embedded development kit
FPGA	Field programmable gate array
FSL	Fast simplex link
IDS	Intrusion detection system
IP	Internet protocol
LUT	Lookup up table
MAC	Media access control
PCI	Peripheral component interconnect
PLB	Processor local bus
ONF	Open networking foundation
RS-232	Recommended Standard 232
SDRAM	synchronous dynamic random access memory
SDN	Software defined networks
SSH	Secure shell
TCP	transmission control protocol
UART	Universal Asynchronous Receiver Transmitter
VLAN	Virtual local area network
XBS	Xilinx base system
XPS	Xilinx Platform Studio

CHAPTER 1: INTRODUCTION

The science of security is the study of cybersecurity at the multisystem level, i.e. protecting a building or institution IT infrastructure from external attacks. Distributed Denial of Service (DDoS) is the most common of these attacks. Attackers come up with new ways to cause DDoS attacks and with each new attack it gets harder to predict and protect the network. It would help if researchers and network administrators could create such scenarios and observe how the network behaves.

Presently, these investigations are driven by either analytical or simulation studies. However, these approaches are severely constrained. To answer some questions analytically usually requires a number of shaky assumptions to make the problem tractable. This raises questions about the fidelity of the results. Simulations — especially those using virtual machines sharing a single workstation/server — are very popular because they execute the actual code used to manage large IT infrastructure. However, the simulations are extremely slow which limits the simulation time of any test run. Moreover, the experimentalist has to ensure that the act of multitasking does not distort the results which can be very tedious and sometimes, very suspicious.

Further, an attack like DDoS is normally caused by using an army of infected machines sending traffic to the victim. The number of such machines could range from hundreds to thousands. For example, the bandwidth of Network Time Protocol (NTP) amplification attack on CloudFare was as high as 400Gbps. A scenario like that is almost impossible to recreate using analytical or simulation test beds.

In this thesis, we explore the concept of Software Defined Networks (SDN). SDN is a new and emerging technology that has promising applications. It offers a configurable global view of the internal network. Flexibility, one of the most desirable

features of SDN, allows network administrators to improve performance by adapting the routing of individual switches. It also allows security experts to devise tactics to prevent attackers from discovering vulnerabilities in the network.

Traditionally, the way to handle large scale emulation was to downscale the capacity of network links and workload. In SHRiNK [1], the authors demonstrate that queueing delay and drop probability are virtually unchanged in a downscaled IP network containing short- and long-lived TCP and UDP flows; the idea was to relate the behaviour of a faster network to a slower one. Suppose if 100 Mbps packet flows are flowing through a 1Gbps link, then this is downscaled to 10 Mbps packet flows flowing through a 100Mbps link. Works in [2] show the issues associated with downscaling a network in the context of SDN. Specifically, they point out two problems —

- A large-scale deployment with distributed SDN controllers cannot be established in a small scale emulated network.
- Ordering is important while updating flow tables on multiple switches. With a large number of nodes, synchronization becomes a difficult task.

A third approach is a full-scale deployment dedicated to experimentation, but this generally considered cost prohibitive.

In order to solve this, an alternative approach is to use a hardware emulation of an institution's IT infrastructure. Since switches get moved, connections rewired, and IT infrastructure grows over time, the only reasonable hardware platform has to be reconfigurable; i.e. commercially available FPGA devices. This thesis explores this alternative approach. Specifically, it implements an institution-wide network of SDN switches that are managed by a central control processor. Legitimate communication, attack strategies and defense strategies are encoded in data files that are tied to the emulation. This enables a test environment that is fast, controlled and repeatable.

1.1 Thesis Statement

Given the need for hardware emulators, the fundamental question is whether hardware emulation *"is feasible to set up an environment that can rapidly test different attack and defense strategies in a controlled and repeatable environment?"*. Specifically, we need to identify if the hardware emulator —

- is a legitimate tool for the science of security?
- is a superior tool for the science of security?
- is a promising solution for the study of cybersecurity going forward?

To answer the aforementioned questions, we implemented a system containing SDN controller, SDN switches and a processor emulating attack packets. The experiments were performed to test —

- Performance — Can the emulator support high bandwidth, real-time emulation?
- Scalability — Can the emulator support a large number of nodes?
- Flexibility — Can a researcher deploy a network scenario without the knowledge of how the hardware is implemented?
- Fidelity — Do the results accurately model and capture anomalies or faults as a real world network?
- Repeatability — Is the emulator repeatable across multiple runs with same input sets?

1.2 Contributions

The main agenda of this work is to build an FPGA based network emulator and successfully emulate SDN network components, different attack and defense strategies. Below are the list of contributions made by this thesis work—

- Design of a light-weight OpenFlow switch based on NetFPGA 1G OpenFlow Switch [3].

  - Four input and output ports.
  - BRAM based lookup tables supporting 32 flows.
  - Parametrized input buffer which could be changed before synthesis.
  - Maximum line rate achieved was 950 Mbps for 128 byte packets at 100MHz.
  - Tail drop queue management algorithm.

- Feature to provide defense strategy in a CSV file.
  - Specify route management at different timestamps.

- Feature to provide attack strategy in a CSV file.
  - Specify different attack events containing packets of different length at a specific rate.

1.3 Outline

The rest of the thesis is orgranised as follows: chapter 2 describes the concept of SDN, existing emulators for SDN and current work on FPGA and SDN. Chapter 3 presents the implementation details of hardware and software for the emulator. This section also describes the format of the input files. Chapter 4 shows the evaluation results and analysis. Finally, chapter 5 concludes the work of this thesis and plans for future research.

CHAPTER 2: BACKGROUND

This chapter gives an introduction to the concept of Software defined networks (SDN) and the need for it, existing emulators and finally, current works on FPGA and SDN. Section 2.1 describes the traditional way of networking. Section 2.2 introduces the SDN-way of handling networks and section 2.3 discusses the impact of SDN on network security. Section 2.4 lists out the existing emulators. In the end, section 2.5 describes the recent work done on FPGAs in the field of SDN.

2.1 Traditional Networks

In a traditional network, a network device is composed of the following planes —

- Control Plane — Control plane makes the decision as to where to route the packets. This is the brain of the device.
- Data Plane — Performs the required operation on the incoming packets as directed by the control plane.
- Management Plane — Provides management of a device via remote connection through secure shell (SSH) or telnet that is managed through a command line interpreter (CLI).

For example, consider IP routing. The IP datagrams are processed and forwarded by the routers. The various routing protocols run on individual routers to setup the routing table to forward packets to other IP networks and hosts. This forms the control plane. When a packet arrives, the router consults the routing table and forwards the packet accordingly by means of data plane. Router management involves operations like updating an operating system image running on the router, security management, performance management etc.

The typical features of "a traditional network" are —

- Most of the functionality is implemented in dedicated hardware such as an application-specific integrated circuit (ASIC).
- Evolution of hardware is a slow process.
- The configuration of a network device is strenuous.
- The network devices are normally black boxes with a management port and are proprietary. Therefore, this makes the network devices vendor dependent and less compatible with each other.
- Innovations are slow.

2.2 The SDN-way

Software defined networking provides an architecture where the data plane is decoupled from the control plane unlike traditional network appliances like routers, switches etc. The framework contains a logically centralized controller which maintains a global view of the network and dynamically configures and manages all the network devices in the network. For example, a router which belongs to the data plane carries the traffic in the network, but its routing table is updated by a centralized controller. This allows the complexity of managing the network from a single point.

According to the Open Networking Foundation (ONF) [4], Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The OpenFlow protocol is a foundational element for building SDN solutions.

The features of SDN architecture as described by ONF are listed below —

- Directly programmable: Network control is directly programmable because it is decoupled from forwarding functions.
- Agile: Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
- Centrally managed: Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
- Programmatically configured: SDN lets network managers configure, manage, secure and optimize network resources very quickly via dynamic and automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.
- Open standards-based and vendor-neutral: When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols

2.2.1 SDN Architecture

The SDN Architecture is designed such that there is an open interface on which one can write applications that can dynamically control the network devices [5].

- The data plane consists of "dumb" switches which follow the commands from the controller and processes the packets accordingly. Each switch contains a flow table which determines how to forward the traffic.
- The control plane decides on the packet routing and programs the flow tables on the SDN switches. It communicates with the switch via the southbound

application programming interface (API). Currently, OpenFlow [3] is a popular protocol used for southbound API. On the other side, it communicates with the SDN applications via the northbound API. SDN controller handles lower level programming of SDN switches and provides a higher level of abstraction to the applications.

- SDN applications run on the application layer. The applications see an abstracted global view of the network. Some examples of such applications are business applications, firewall, providing end to end quality of service etc.

Figure 2.1 shows all the components of SDN. In short, SDN provides better visibility of the entire network and moves the complexity of controlling the network to a single logical point. Applications like firewall policy management become very simple as having a global view aids in avoiding anomalies in policies. Also, Virtualization has made network management very challenging. As a result, these policies must be changed on the fly every now and then. The SDN controller answers this problem by providing a centralized security and policy information.

2.2.2 OpenFlow

OpenFlow is an instance of SDN. It is a southbound API used for the communication between network controller and an OpenFlow enabled switch. The basic operation of the switch is to collect the header from the packet and compare it against the flow table. The switch then reads the action associated with the matching entry and executes the corresponding action.

2.3 SDN Network Security — Pros and Cons

The concept of SDN can solve various network security issues. However, it has its own challenges and a lot of research is taking place on this end. A centralized controller provides a direct view of the traffic forwarded by the individual switches in the network. This global view aids in analyzing traffic in the complete network, which is almost impossible in traditional networks. S.A Mehdi, J. Khalid and S. A

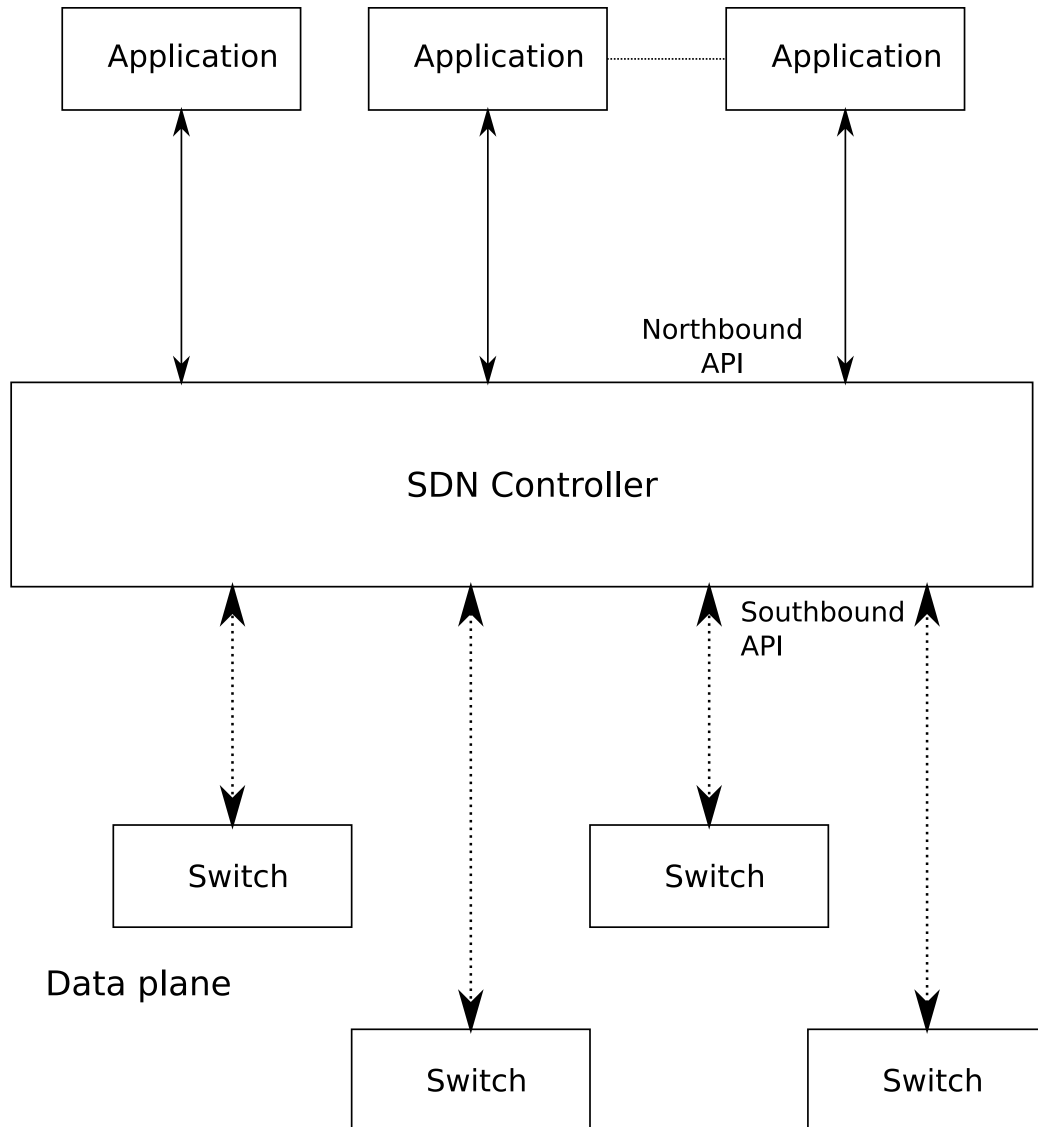


Figure 2.1: SDN Architecture

Khayam [6] implement and evaluate detection of network scans, DoS attacks using the data from the centralized controller. S. Shin et al. [7] present an application which can even detect Zombie machines that are part of the botnet. The data collected from multiple switches allows the controller to detect attacks and act upon it. It can isolate the affected flows from the normal traffic and forward it to security devices like an intrusion detection system (IDS).

However, having a centralized controller poses its own challenges. The first challenge is, as the number of switches in the network increases, the controller becomes a bottleneck because now it has to analyze traffic from so many switches. Kreutz et al. [8], describe two properties of SDN that makes it very popular for malicious users —

- Ability to control the network through software.
- Centralized "network intelligence"

An attack on the controller leads to severe consequences. Thus, complete protection of the controller becomes crucial in SDN.

2.4 Existing SDN Simulators/Emulators

2.4.1 Mininet

Mininet is an open source network emulator which allows one to create a virtual network in software [9]. Mininet runs a set of virtual hosts, switches and links on a single server. This is a simple to use emulator that allows one to create custom topologies, run real applications on individual switches and customize packet forwarding. This is a great tool for research, learning and experimentation. Mininet is meant for small scale experiments with fewer nodes and slower links, as each node has to share CPU and memory resources.

2.4.2 Estinet

Estinet provides simulation and emulation environments for switches and NOX [10], POX [11] or Floodlight [12] controllers [13]. The controllers are real-world con-

trollers without any modifications and control thousands of simulated switches. Estinet's performance results are realistic, accurate and repeatable.

2.4.3 NS3

An open source discrete-event network simulator, NS3 is used primarily for research and educational use [14]. It enables one to study network performance or protocol operation in a controllable or scalable environment. NS3 is capable of interconnecting real or virtual machines. However, NS3 is not as scalable as pure simulation.

2.5 FPGA and SDN

Most of the work done on FPGA with respect to SDN has been to make the hardware more programmable and build a flexible SDN switch.

NetFPGA [15] platform uses an FPGA based approach to prototype network devices. It provides open source hardware and software which can be used for rapid prototyping and mainly intended for teaching network hardware and router design. Multiple switches have been implemented on this platform. Jad, David, G.Adam, Guido and Nick, 2008 [16] have implemented a switch on the NetFPGA platform. The hardware contains a Peripheral Component Interconnect (PCI) card with the following components —

- Xilinx Virtex —II Pro 50
- 4×1 Gbps Ethernet ports
- Two parallel banks of 18 MBit Zero-bus turnaround (ZBT) SRAM.
- 64MB DDR DRAM

This switch is capable of supporting 1Gbps line rate across all 4 ports. This supports 10-tuple matching which can be matched either exactly or using wildcards. This switch supports 64K exact match rules and 32 wildcard rules. The hardware is controlled by the switch management software that runs on the host PC. The software

is open source [17]. The switch was improved by Yabe [18] on NetFPGA 10G board which was capable of supporting 10G traffic across all ports.

The Smart switch developed by Gianni, Andrea, Stefano and Gregorio, 2011 [19] is an OpenFlow compliant architecture on NetFPGA meant to be used for load-balancing. Their implementation supported 100K flow entries while still meeting the line rate of 1 Gbps across all ports.

Asif and Nirav [20] present a FPGA based switch architecture, which can support 10Gbps line rate across all ports. The switch design is modular and is portable across FPGA platforms. The authors use Bluespec SystemVerilog (BSV) which makes it possible to achieve flexibility and portability.

Muhammad, Murtaza, Mukarram and Nick, 2010 [16] present Switchblade, a programmable hardware on which custom data-plane functions can be deployed rapidly. Switchblade offers a set of building blocks that are commonly used in network devices. A subset of blocks could be chosen to implement custom data plane functions. This aids in rapid prototyping which is otherwise hard if a normal cycle of hardware development is considered.

CHAPTER 3: IMPLEMENTATION

This section contains information about the implementation of the SDN switch core, interconnection of multiple SDN switches with the SDN controller (PowerPC) and the attacker nodes (Microblaze). The main functionality of the SDN switch is to modify packet header fields based on the flow table and forward it to the next port(s). The SDN controller is responsible for programming the flow table in each switch and monitor these switches to observe each packet flow. The Microblaze processor, acting as attacker nodes plays the role of an outside network and transmits packets at different programmable rates to the SDN switch network using an array of packet drivers. The big picture showing the connection between different components is given in Figure 3.1.

The chapter is divided into two main sections. Section 3.1 describes the details of hardware implementation and section 3.4 describes the software implementation.

3.1 SDN Switch

The Figure 3.2 shows the implementation of a 4×4 programmable switch implemented with store and forward architecture. It contains multiple internal registers for software access to provide insight about the traffic flowing through the switch. This switch is a simplified version of NetFPGA's 1G Switch [21]. The logic is customized so that multiple switches can fit into one FPGA. This switch provides 10-tuple matching to identify a flow and can support 32 flows at a time. The flow lookup table is implemented in a hash table which is programmed by software.

There are five major blocks in the implementation of the switch as shown in Figure 3.2. The bus width used throughout the design is 64 bits (one word or one line). There is an 8 bit control signal used to identify the last byte in the packet.

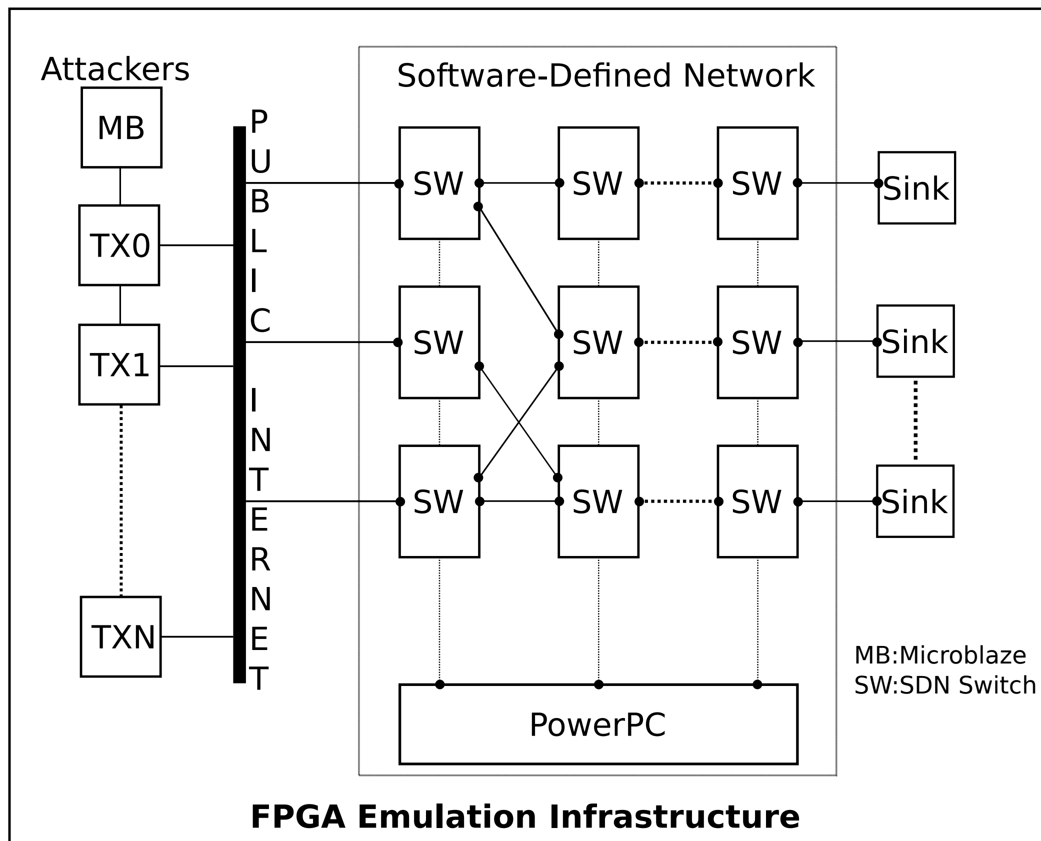


Figure 3.1: The big picture showing the connection between the SDN Controller (PowerPC), Attacker nodes (Microblaze) and SDN Switches

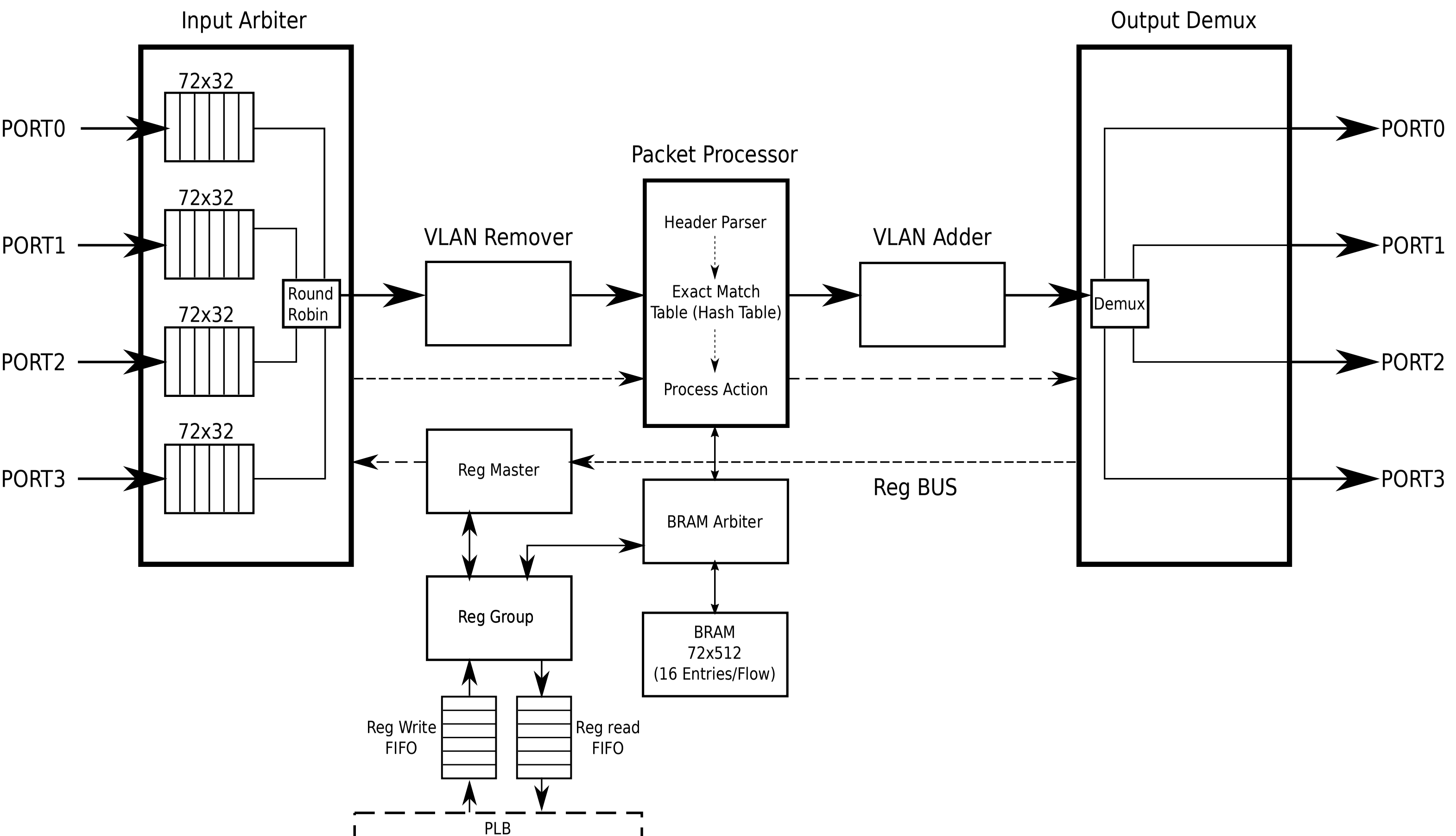


Figure 3.2: Implementation of 4×4 SDN Switch

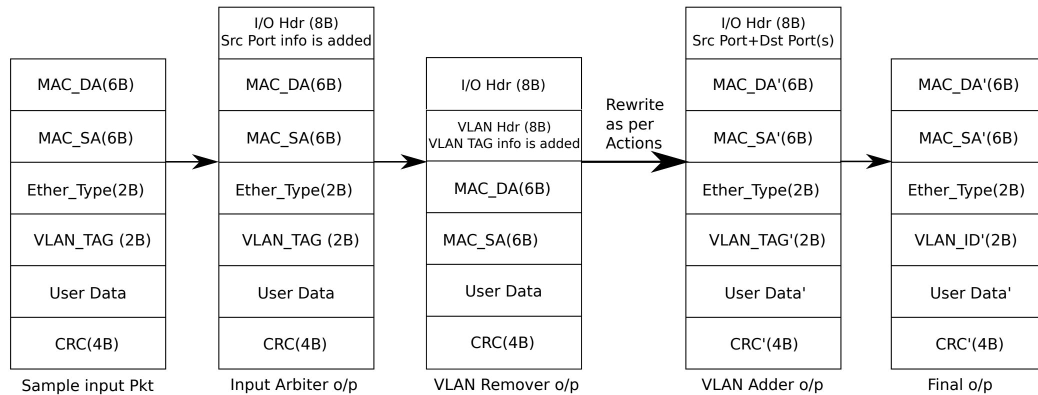


Figure 3.3: Sample packet flow through a switch

If all the control bits are 0, then all bytes are valid. If any of the eight bits in the control signal is set, then the corresponding byte is the last byte of the packet. Only one bit can be set in the control signal at any time. Figure 3.3 shows how a packet is modified in different stages of a switch. The following subsections describe individual blocks.

3.1.1 Ingress Multiplexer/Arbiter

The functionality of this block is to store an entire packet, add a packet header of one line which contains source port and length information and multiplex packets from different ports to the outgoing link. The multiplexing is done using round robin arbitration. There are four per port 32 deep FIFOs. Each FIFO is implemented to drop packets on packet boundaries. That is, if the FIFO buffer gets filled up before a complete packet is pushed in, then the entire packet is dropped by removing the partial packet data which is already present in the buffer. The FIFO buffer sends a back pressure signal to the transmitter. However, if the transmitter continues to transmit, then the packets are dropped. There is an additional buffer that holds the length of each packet that is pushed into each packet FIFO. While reading the packet out of the buffer, the length FIFO is read first to get the length and is used to form the packet header and then the rest of the packet is read out of the data FIFO.

3.1.2 VLAN Remover

The functionality of this block is to remove the Virtual Local Area Network (VLAN) information from the packet and add it as the packet header. The reason for doing this is to prepare the packet for parsing. The NetFPGA parser that is used in the switch expects the packet in this format. If the packet does not contain the VLAN information, then it is unchanged.

3.1.3 Output Lookup Block

This block performs packet processing and packet rewrites. The packet is parsed to extract flow header fields (10 tuples). A flow table lookup is performed using the flow header, the corresponding actions are read and packet is modified accordingly.

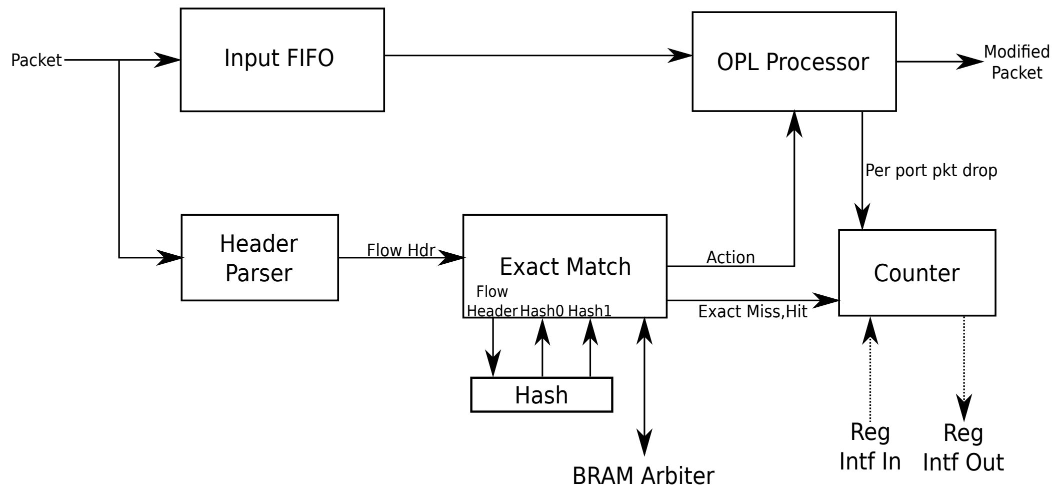


Figure 3.4: Lookup and forwarding mechanism

3.1.3.1 Header Parser

This block parses the incoming packets. The list of all the fields that are parsed is shown in Table 3.1.

A packet can be matched based on the fields in Table 3.1. This provides a great flexibility in the current implementation and is the basis for 10-tuple matching (Ether type and VLAN ID fall under same tuple). The parsed information is combined into a 248 bit header known as a flow header such that it has the same bit order as the match field in the flow table. This is passed to the exact match block with a handshake.

Table 3.1: Flow table header fields

Field	Width(in bits)
TCP destination port	16
TCP source port	16
IP Protocol	8
Destination IP	32
Source IP	32
Ether type	16
Destination MAC address	48
Source MAC address	48
Input port	8
IP TOS	8
VLAN ID	16

3.1.3.2 Exact Match

This block performs the flow table (hash table) lookup, updates the per port packet and byte counters and reads the corresponding actions. If the packet header does not match the corresponding hash table entries, then it is considered as a table miss and the packet is dropped. First, the structure of hash table and hashing function are described and then the functionality of exact match is described.

The hash table is constructed using 512 deep and 64 bit wide block RAM (BRAM). The hash table can support 32 different flows with each flow occupying 16 entries in the hash table. Cyclic redundancy code (CRC) is used as the hashing function. The pseudo code to form the hash indices is shown below

```
//Pad the 248 bit header to form 256 bit data
Data     = {8'd0,flow_hdr}
//Reverse the bits in data
Data_Rev = bit_reverse(Data)
//Consider last 5 bits to identify a flow between 0 and 31
//Polynomial for Hash0 = 0x04C11DB7
Hash0    = CRC32_FUNC_0(Data_Rev)%5;
//Polynomial for Hash1 = 0x1EDC6F41
```

```
Hash1     = CRC32_FUNC_1(Data_Rev)%5;
```

The exact match block reads the flow header present in the address $hash0 \times 16$ and $hash1 \times 16$ (multiplied with 16, as each flow is 16 deep). This flow header is compared with the flow header of the current packet. If it matches, then counters are updated and the corresponding action is read. The software is responsible for programming the hash table. Also, the existing design is based on a proactive SDN controller. That is, the controller should populate the flow table ahead of time for all traffic matches that could come into the switch. The flow action data is 320 bits wide. The fields in action data is show in Table 3.2.

Table 3.2: Packet action fields

Action Fields
Destination Ports
Set VLAN VID
Set VLAN PCP
STRIP VLAN
Set destination MAC
Set source MAC
Set destination IP
Set source IP
Set IP TOS
Set TCP destination port
Set TCP source port

Except for the destination ports, all the information are validated by a set flag. If there is a flow hit, then the corresponding action data is read by the exact match block and it is forwarded to the action processor block. If there is a miss, then an action 'send_to_nowhere' (by setting all the destination port bits to 0) is sent to the action processor block. The action information is sent to the action processor with a handshake.

3.1.3.3 Output Processing Logic (OPL)

This block is reused from the NetFPGA project without any modification. Once an action is ready, the action processor reads the packet from the input FIFO and

executes the action on that packet. Depending on the contents of the action, it modifies the header (the destination ports and VLAN information) and the packet data. The destination ports are given as a bit map and this is used by the egress demultiplexer to forward the packet to the corresponding ports.

3.1.4 VLAN Adder

The functionality of this block is opposite to that of VLAN remove block as shown in subsection 3.1.2. That is, it adds the VLAN field present in the packet header into the packet. If there is no VLAN field, then the packet is unchanged and is forwarded to the next block.

3.1.5 Egress Demultiplexer

This block forwards the packet to the corresponding destination port(s) based on the forwarding bitmask. This block also removes the header containing the port and length information. If all the bits in the bitmask are set to 0, then the packet is dropped. This block is simplified as compared to the NetFPGA implementation. There is no internal buffer present in the block. As a result, if any of the receiving switches sends a flow control, then that is chained to the packet processor module.

3.2 Register Interface

This is the interface between SDN controller and multiple switches on the board. It is used to program the flow table and read registers. The current design supports majority of the required counters as mentioned in the OpenFlow Switch Specification [3]. Each of these counters is unsigned and wrap around with no overflow indicator. Table 3.3 shows the list of counters that are supported in the current design.

Figure 3.5 shows the interface between SDN controller (PowerPC) and other switches present in the design. In each switch, there are two FIFOs — one is to store the commands from the controller until it is read by the switch. The other is to buffer the read data until is read by the controller. The top level register interface in all the switches is connected in a daisy chain fashion. Using the register interface,

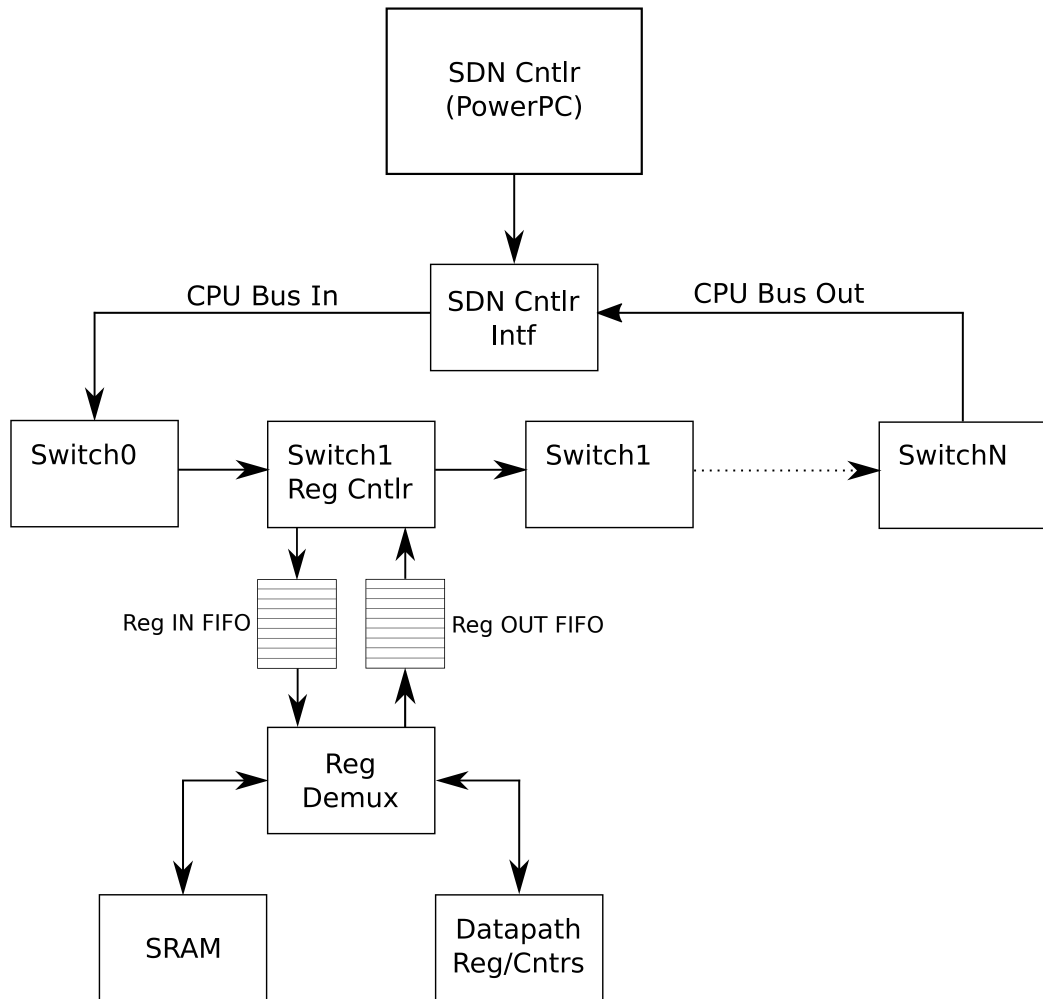


Figure 3.5: Interface between SDN controller and the switches

Table 3.3: List of counters

Counter	Width(in bits)
Per Flow Entry	
Received Packets	24
Received Bytes	32
Last seen(seconds)	8
Per Flow Table	
Lookup Hits	32
Lookup Misses	32
Per Port	
Packet dropped due to overflow	32
Per Switch	
Total Packets	32

the controller performs following operations.

- Access BRAM: Update flow table and read per flow counters.
- Access datapath registers: These are the counters present in the datapath (flow table, per port, packet drops and total packets) and other debug registers.

3.3 Transmission of Flows Using Microblaze

The Microblaze processor is used to send different packets to the network. This acts as the "outer world" to the network of switches. The initial design had Microblaze processor transmitting packets via fast simplex link (FSL) directly to the input interface of multiple switches. However, this resulted in a very low packet rate. The maximum clock speed for Microbalze processor on ML410 is 100MHz and the clock speed for the rest of the design can go as low as 50 MHz. Also, the bus width of the switch is 64 bits and it is accompanied by 8 bit control data. The 32 bit Microblaze processor would take 3 instructions to write to a single packet line (2 for the data and one for the control). Due to clock speed, there is a 2X gain on the processor speed relative to the switch hardware. But, the processor writes 1/3rd of a line in every instruction. Therefore, effectively the switch was getting packets at 33.33MHz (without considering other overheads in the processor). This gets worse when multi-

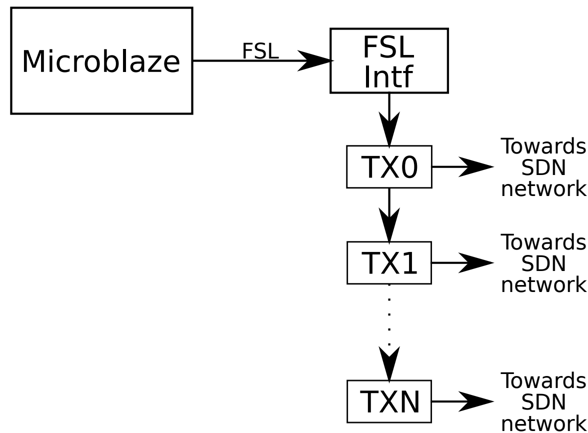


Figure 3.6: Design of the Microblaze and packet drivers interface

ple switches are present and the end result was a very low flow rate. Of course, using multiple Microblaze processors can easily overcome this issue and add more flexibility. But, this would reduce the total number of switches that can be implemented.

The current design has a Microblaze processor accompanied by multiple programmable hardware packet drivers. Each of these drivers contain a small buffer to hold one packet and it transmits this packet to the switch interface at the programmed rate. The processor writes a packet into the buffer, sets the rate and sets the control to start the driver. Similarly, the processor can set the control to stop the driver as well. Figure 3.6 shows the complete design. The Microblaze processor communicates with the FSL interface block using FSL. The processor sends the data and control information in 3 instructions. This is packed into one line by the FSL interface block and forwarded to the corresponding packet driver. All the packet drivers are connected to the FSL interface block in a daisy chain fashion. There is an ID associated with each data line to indicate the receiving driver.

3.4 Emulator Software

The different stages in the software are as shown in Figure 3.7. Currently, the user input is given directly as the elaborated CSV file. In future, a scenario generator can provide a higher level of abstraction which would take user input in a concise form and populate the CSV files in the required format.

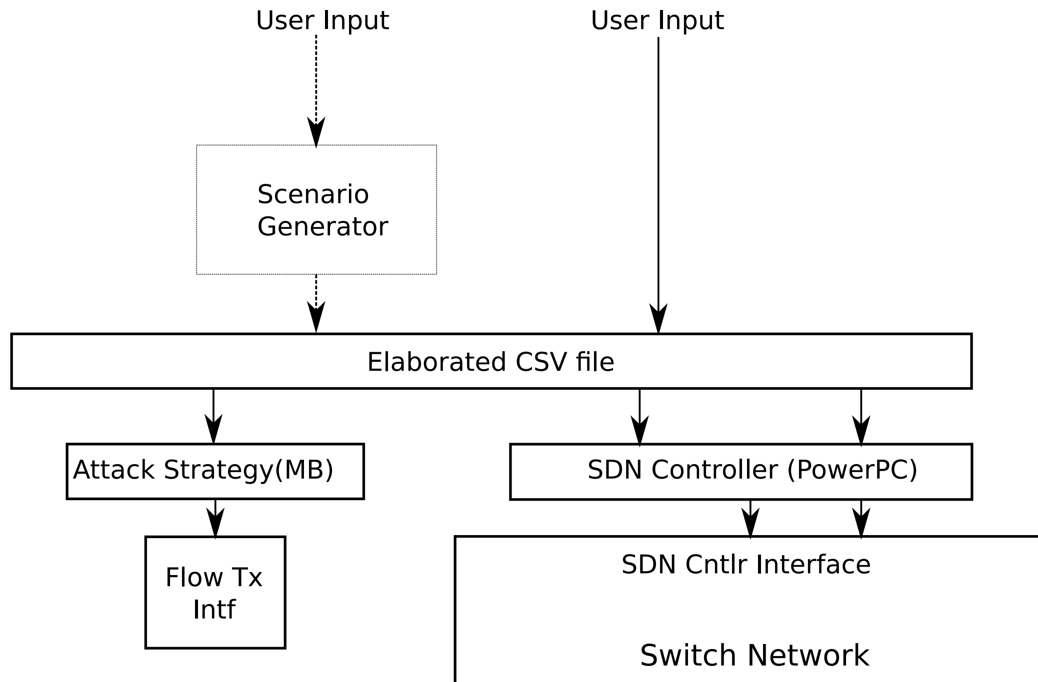


Figure 3.7: Emulation software big picture

The user input is used by (a) the controller to manage the flow tables in different switches, and (b) the packet generator processor to start or stop flows at given times. The controller talks directly with the hardware and is responsible for —

- Adding flows to the flow table on the switches at different timestamps.
- Remove flows from the flow tables at different timestamps.
- Read all the counters periodically and write it to a log, which would be used to perform post processing.

The user input is given as CSV files. Subsections 3.4.1 and 3.4.2 talk about the software running on the SDN controller (PowerPC) and packet generator (Microblaze) respectively.

3.4.1 SDN Controller Software

The user input contains the following information —

- List of packets and the corresponding packet header. CSV format for describing

a packet —

```
<PKT_ID>,<PKT_LEN_IN_BYTES>,<PKT_HEADER>,<PKT_DATA>
<PKT_HEADER> = <MAC_DA[47:32]>,<MAC_DA[31:0]>,
               <MAC_SA[47:32]>,<MAC_SA[31:0]>,
               <VLAN_ID>,<ETHER_TYPE>,<DST_IP>,
               <SRC_IP>,<IP_PROTOCOL>,<IP_TOS>,
               <TCP_DST_PORT>,<TCP_SRC_PORT>,
               <INPUT_PORT>
//Split packet data into 64bit words
<PKT_DATA> = <PKT_DATA0>,<PKT_DATA1>,.....
```

- List of flows. Each flow contains the input packet, the list of switches in the flow's route, the input and output port on each switch and the action that has to be executed on each switch. CSV format for describing a flow —

  ```
  <FLOW_ID>,<PACKET_ID>,<SWITCH_ID>,<INPUT_PORT>,
  <OUTPUT_PORT>,....
  ```

  For each switch in the flow's route, describe the action —

  ```
  <ACTION_FLAG>,<FWD_BIT_MASK>,<VLAN_VID>,<VLAN_PCP>,
  <MAC_DA[47:31]>,<MAC_DA[31:0]>,<MAC_SA[47:31]>,
  <MAC_SA[31:0]>,<IP_DST>,<IP_SRC>,<IP_TOS>,
  <TCP_DST_PORT>,<TCP_SRC_PORT>
  ```

  Action flag is a set of flags, where each flag indicates if the corresponding action must be executed or not. Action flag is described as shown below —

  ```
  <SET_DST_PORT>,<SET_SRC_PORT>,<SET_IP_TOS>,<SET_IP_DST>,
  <SET_IP_SRC>,<SET_MAC_DST>,<SET_MAC_SRC>,<STRIP_VLAN>,
  <SET_VLAN_PCP>,<SET_VLAN_VID>,<OUTPUT>
  ```

- Event list — This contains the scenario of events.

  CSV format to describe a list of events —

  ```
  TIME,<TIME_STAMP>,<NUMBER_OF_FLOWS>
  <FLOW_ID>,<START/STOP>
  ```

  Table 3.4 shows a sample event which indicates the controller to start FLOW2 at 10th second. The CSV format to describe this event is as shown below —

  ```
  TIME, 10, 1
  FLOW2_ID, START
  ```

Table 3.4: Event list entry

TIME	Flow Number	Trigger
10s	FLOW2	START

During each time step (each second), the controller checks if it has to perform any action. If so, it will read the corresponding flow and program the flow table on the switches in the flow's route. The data structure used to hold all the information is shown in Figure 3.8. Consider an example where the controller has to install FLOW0 in switches 0, 1 and 2. The user input contains the action to be taken in each switch and the flow header for the first switch (match field). For all other switches, the controller has to compute the flow header. The computation of the flow header is as shown in Figure 3.9. The controller then updates the flow header and action on switches 0, 1 and 2.

3.4.2 Attack Packets Generator

The main requirement of emulating an attack strategy is to give a flexibility such that it can accurately establish the different ways in which an attacker might prosecute an attack. A Mircroblaze processor is used for this purpose. The processor utilizes two data structures — one contains the list of all packets that would be used in

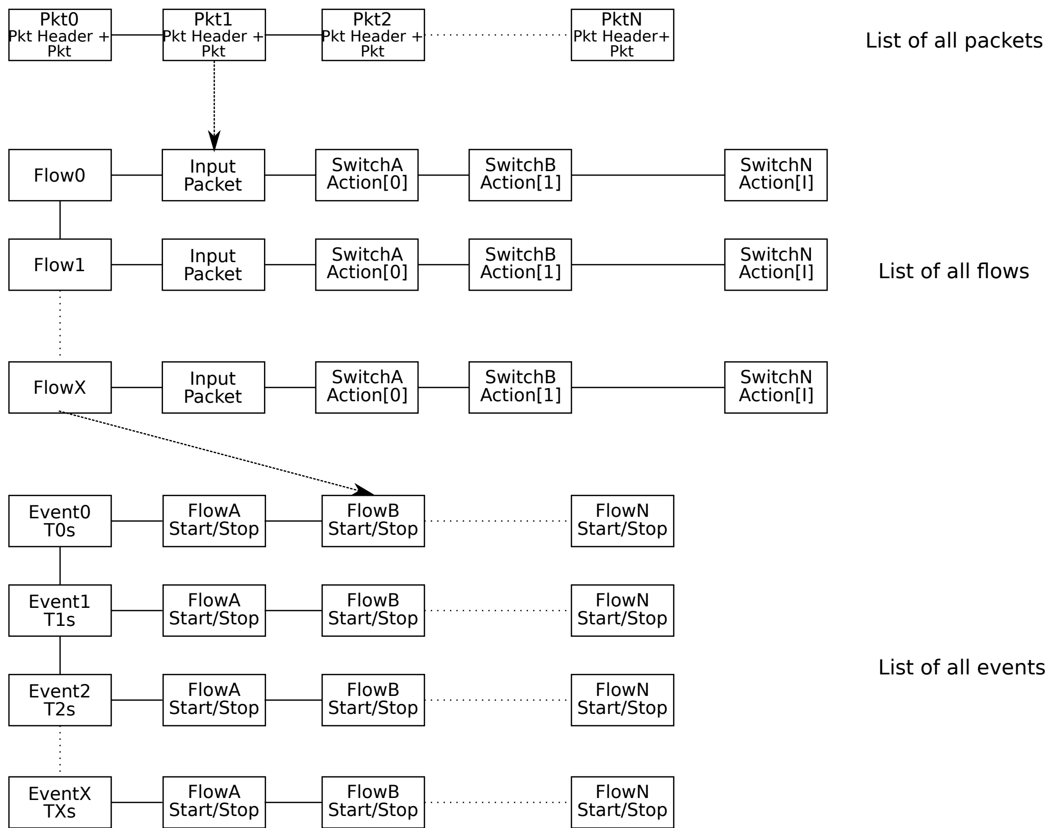


Figure 3.8: Controller event handling structure

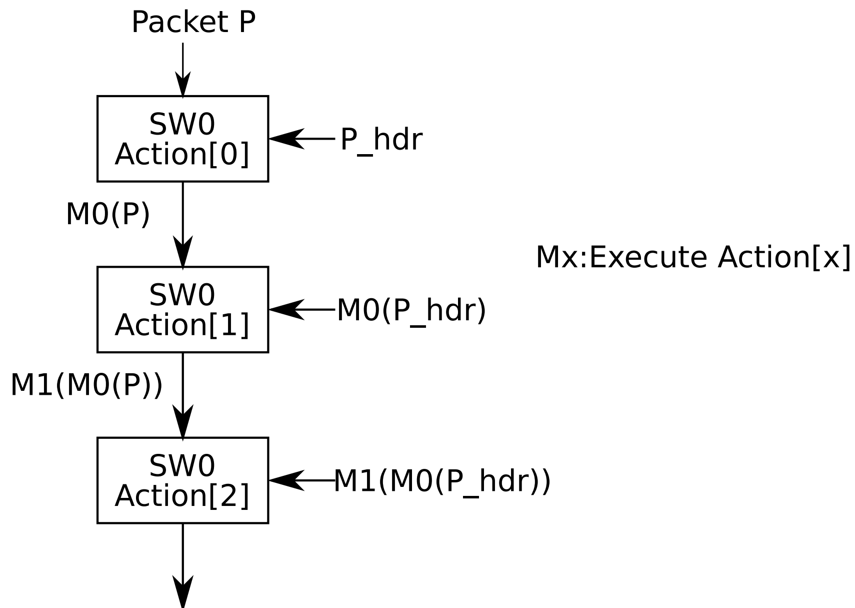


Figure 3.9: Header calculation for installing flows in flow table

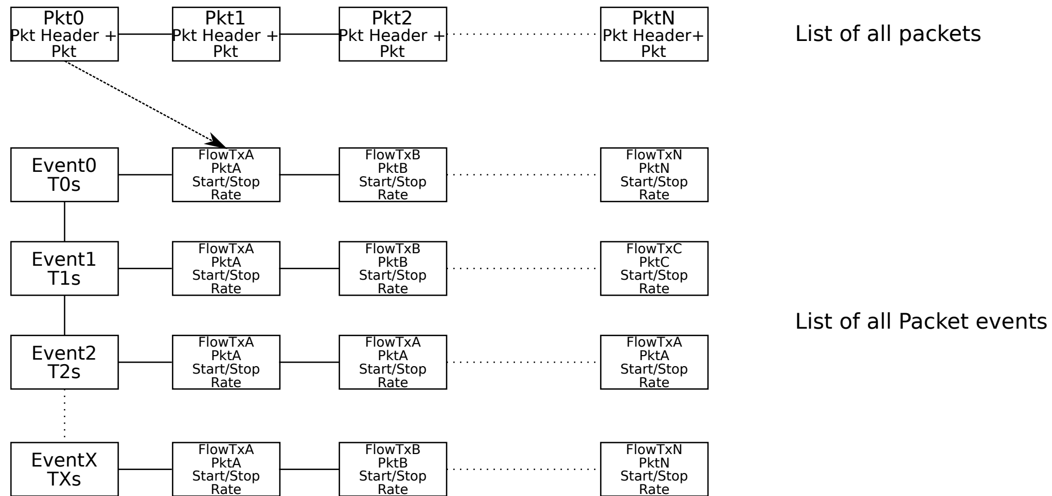


Figure 3.10: Attack emulation handling struture

emulation and the other contains the list of event samples and the corresponding actions to be performed.

Figure 3.10 describes the relation between these two structures. The format for both structures are shown below —

CSV file format —

```
TIME, <TIME_STAMP>, <NUMBER_OF_EVENTS>
<PKT_ID>, <TX_ID>, <START/STOP>, <BURST_ON>, <BURST_OFF>
```

In the above format, burst_on and burst_off is used to determine the packet rate. Burst_on indicates the number of cycles for which the 64 bit word must be transmitted. Burst_off indicates the number of idle cycles between two valid words. For example, suppose at the 50th second if a user wants to start transmission of the packet with ID=10 to the network using a rate of 1Mbps to packet drivers with ID=1 and ID=2, then the event list can be written like this —

```
//First compute the values for Burst_ON and Burst_OFF.
//Setting the Burst_ON size to 16, then calculating
//Burst_off -
```

$$\text{Time required to send 16x64bits at 1Mbps} \ = \frac{16 \times 64}{10^6} \text{ s}$$

$$\text{Number of clock cycles in the above time} = \frac{\frac{16 \times 64}{10^6}}{10^{-8}}$$

$$\text{Burst_OFF} \ = 102400 - 16 = 102384$$

```
TIME, 50, 2
10, 1, START, 16, 102384
10, 2, START, 16, 102384
```

The file format for list of packets is similar to what as shown in subsection 3.4.1.

During each time step, the processor checks if there is any event to be executed at that time. If so, it will read the list of packets and the corresponding action. The action contains the trigger START or STOP, the ID of the packet driver that would transmit the packet and packet rate. If the action is "START", then it programs the internal buffer of the corresponding packet driver with packet data and rate and sets its control bit to start transmitting. If the action is "STOP", then it clears the control bit in the packet driver.

CHAPTER 4: EVALUATION

This chapter is divided into seven parts. Section 4.1 lists the important metrics to monitor network health. Section 4.2 describes the components required for experimental setup. The resource utilization and the performance of the switch are discussed in section 4.3 and section 4.4 respectively. Section 4.5 mentions the repeatability of the current implementation. A set of experiments to test the feasibility of the design are discussed in section 4.6. Finally, an analysis of the data is presented in section 4.7.

4.1 Evaluation Metrics

The following metrics are important to monitor the health of the network —

- Throughput — This is a measure of the rate of successful message delivery and is measured in bits/second or packets/second. The current implementation can support both the metrics.
- Packet loss (Packet drops) — This measure helps in identifying congestion in a particular switch. The packet drops take place due to the buffer overflow in a particular switch. The current implementation supports per port packet drop for each switch using which the rate of packet drops can be measured. However, it does not give the number of bytes dropped per port.
- End to end latency — This is a measure of the amount of time required to transmit a packet from source to destination. This includes the transmission delay, propagation delay, processing delay and queueing delay. The behaviour of the network has a direct impact on the latency. Suppose, if a packet travels through a number of congested switches in its path, then the queueing delay

increases thus increasing the latency. This is not supported in the current implementation. It requires some hardware modification and will be supported in future.

- Traffic between SDN controller and data plane — This metric is specific to SDN. This is a measure of the number of new flows that are coming into the switch. Whenever a switch does not find a packet in the flow table, it consults the controller to update the flow table for that particular flow. A malicious attacker can craft packets such that the switches in the infrastructure layer consult the controller frequently, which rapidly increases the number of control packets in the network. Current implementation contains a proactive controller. That is, the controller updates the flow in the flow table of all switches and there is no communication between the switches and the controller to add unknown flows. The future work involves implementing a reactive controller and this metric would be supported in that design.

4.2 Experimental Setup

All the experiments designed to answer the thesis question were implemented on Xilinx ML410 board that has the following core components —

- Xilinx Virtex-4
  - 2 PowerPC processor blocks
  - 4176 Kb Block RAM
  - 396 Kb distributed RAM
- 512MB DDR2 SDRAM

ModelSim was used for simulation of hardware design and verification. Version 14.5 of ISE/EDK tools was used to create and synthesize the design and for software implementation. The entire design was implemented using 100MHz clock. The core components of the design included the SDN switch as described in section 3.1,

PowerPC processor acting as the SDN Controller and Mircroblaze processor to emulate the attack scenario as described in section 3.4. The design also involved SDN controller interface block, which is the interface between PowerPC and the network of switches, and packet drivers to drive the packets into the network. PowerPC was connected to the SDN controller interface via the PLB bus. Microblaze uses FSL to communicate with the packet drivers. A custom hardware timer was implemented which counts every second and is 32 bits wide. The timer value was read by PowerPC and Microblaze via the PLB bus. 512 MB of DDR2 SDRAM memory was included and logically shared between PowerPC and Microblaze. A RS-232 core operating at 115200 bps with no parity was included to display the output from PowerPC and Microblaze on terminal via UART. The bitstream and the elf files were downloaded onto the board using XMD. Minicom, a text-based modem control and terminal emulation program was used for setting up a remote serial console. This provides a way to log the outputs from PowerPC and Microblaze processors and analyze the results. The on-board verification was done using Xilinx Chipscope Pro. The effectiveness of the emulator is determined by its performance and the breadth of experiments that could be performed on this setup.

4.2.1 Time to Setup and Execute Experiment

This involves connecting the network components on Xilinx XPS, synthesizing the design and generating the bit stream. This step takes most of the time. Once the hardware bitstream is ready, the CSV file must be written to generate different scenarios as described in subsection 3.4.1 and subsection 3.4.2. Then the software is compiled in Xilinx XSDK. Finally, the bitstream and the elf files must be downloaded onto the board using Xilinx XMD to perform real time emulation. The output log obtained from Minicom is passed through the python scripts to generate different graphs.

4.2.2 Post-processing Scripts

Python scripts were written to analyze the data output from PowerPC and Microblaze. The output information was used to plot various graphs —

- Total packets per second in all switches.
- Per port, per second packet drops in each switch.
- Flow rate in packets/second and bytes/second.
- Average flow rate in packets/second and bytes/second.

4.3 Resource Utilization

Number of switches that can be fit onto an FPGA is a key requirement for the researchers to perform experiments. The Virtex4 ML410 board has 50,560 registers and 4-input LUTs. There are 25,242 slices. Each slice is composed of 2 registers and 2 LUTs. The board also contains 129 BRAM and 6670 LUTRAM. The resource utilization of individual components is as shown in Table 4.1.

Table 4.1: Resource utilization

Component	Number of LUTs	Number of slice registers	BRAM/ LUT RAM
SDN Switch	8,934	5,849	1,484
Microblaze processor	2,414	1,065	471
SDN Controller Interface	1,759	1,172	—
FSL interface	89	157	—
Packet Drivers	129	125	2
Custom Timer	266	256	—
Total resource on board	50,560	50,560	6,799

As seen from the Table 4.1 and total resources available on the board, the SDN switch block utilizes about 16.7% of the LUTs and 11.56% of the Slice registers. The packet driver block occupies only about 0.25% of LUTs and slice registers. These are the two components that vary in number based on user requirements. The rest of the blocks occupy about 9% of LUTs and 5% of slice registers. The maximum number of switches that could be implemented on Virtex - 4 ML410 board were 4. Table 4.2

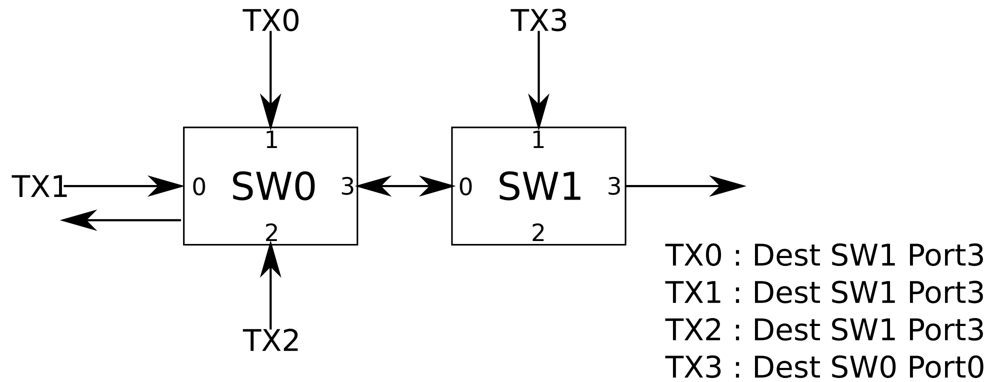


Figure 4.1: Performance measure experimental setup

shows the summary of board resource utilization with 4 switches and 4 packet drivers.

Table 4.2: Utilization percentage

	Resource Used	Utilization
Number of slice registers	32135	63%
Number of 4-input LUTs	46130	86%
RAMB16 (16 x 2 memory)	129	55%

Clearly, the number of SDN switches that can be used depends on the amount of logic resources that are available on the board. The experiment was done on a 10 year old hardware. More recent FPGAs contain 10 times more resources and thus enable the ability to use a large number of switches and packet drivers.

4.4 Performance

The performance was evaluated by the forwarding capacity of individual switches. The experimental setup used to measure the forwarding rate is as shown in Figure 4.1. Packet drivers — TX0, TX1, TX2 and TX3 were connected as shown in the figure and made to transmit packets at full rate. Thus, switch SW0 had streams of packets coming in from all ports. The forwarding capacity of switch SW0 indicates the measure of performance. The results obtained from Mininet Hifi are discussed first, then the results obtained from the current implementation are discussed.

4.4.1 Mininet HiFi Performance

A topology was created as shown in Figure 4.1. The switches used in the experiment were Open vSwitch [22]. Packet drivers TX0, TX1, TX2 and TX3 were replaced by hosts 0, 1, 2 and 3. Link capacity was set to 700Mbps and the max queue length was set to 1000. The max queue length does not matter in the current experiment because the packet lengths are relatively small and testing was done on UDP traffic. The experiment was done on a 4 core, 3.4 GHz Intel I-7 2600 processor.

Topology:

The command used to setup the topology —

```
sudo mn --custom ~/mininet/custom/custom-topo.py \
        --topo perf_measure --switch ovsk        \
        --link tc,bw=700,max_queue_size=1000     \
        --controller remote --mac
```

The topology was specified in the file custom-topo.py and the file was provided while creating the topology. The option '−−mac' makes MAC equal to the IP and the option '−−controller remote' is used to specify that the controller is on the local host.

Controller:

Hub controller from Pox library was used. A hub simply forwards the packet to all the ports except the incoming port. The command to start the controller —

```
python ./pox.py forwarding.hub
```

Measuring Performance:

Iperf [23] was used to measure performance. UDP server was started at the receiving hosts (hosts connected to SW0 Port0 and SW1 Port3). Each host was made to send UDP traffic at different lengths for 100 s and the results obtained are shown in Table 4.3.

Iperf commands used in the experiment —

```
//To start server(-s) and use UDP (-u)
iperf -s -u


//Client
iperf -c <server_ip> -b 700m -l <64/128/256> -t 100
```

Table 4.3: Line rate for different packet lengths in Mininet HiFi

Packet Length	Bit rate (Mbps)
64B	9.95
76B	18.7
128B	23.1
256B	45.3
1470B	180

4.4.2 Performance of the Current Design

Table 4.4 shows the forwarding rate for packet lengths of 64B, 76B and 128B.

Table 4.4: Line rate for different packet lengths

Packet Length	Bit rate (Mbps)	Packet rate(Pkts/sec)
64B	760	1.17M
76B	890	1.17M
128B	950	.78M

The results in Table 4.4 showcase the reason for using hardware emulators. Table 4.5 show the speedup obtained by using hardware emulator with respect to Mininet HiFi at different lengths. The speedup reduces as the packet length is increased. However, even for a packet length of 1470B, the maximum forwarding rate in Mininet HiFi was lesser than that of the forwarding rate for 128B in the hardware (around 5times). Unfortunately, the current hardware design can support a maximum of 256B and the current packet driver can support a maximum of 128B. Therefore, the tests weren't performed for greater lengths. Also, a simple packet generator was used in all the experiments, that sent X word(s) (1 word = 8B) in a window Y cycle(s), where X

and Y are the inputs given by the user. Using a packet generator as shown in [24] would yield a consistent line rate at all packet lengths.

Table 4.5: Speed up w.r.t Mininet Hifi

Packet Length	64	76	128
Speed up	76.3	47.5	38.5

4.5 Repeatability

In the current design, the output is repeatable as there are no events which would vary in different runs. For a given set of inputs, the results obtained are exactly the same.

4.6 Evaluation of the Network Emulator

In order to evaluate the network emulator, four experiments were performed. For each experiment, the attack strategy (set of input packets, there rate and start/stop time stamp) and flow handling specification were given as inputs. The network topology was designed and synthesized using XPS and is fixed for different runs.

4.6.1 Ping of Death

A small scale network of two switches was setup on hardware. S, a single packet driver acting as the attacker was setup to transmit traffic at increasing bandwidth into the network. This type of attack, where there is a single source sending high volume traffic is less effective today due to high network capacity and processing capabilities. But this approach can be easily extrapolated to Smurf attack [25] or reflector based denial of service attack [26]. The task of mitigating this type of attack depends on the flow handling specification. For the sake of this experiment, the emulator software was tweaked to add an additional trigger action —

```
If flowA rate > Threshold -> Remove flowA & Insert flowB
```

Figure 4.2 shows the attack traffic without the above action. In Figure 4.3 as soon the incoming traffic bandwidth exceeds the threshold ($= 3.5 \times 10^8 bytes/second$) the flow is blocked and we do not see any traffic in switch1 (SW1). In a more

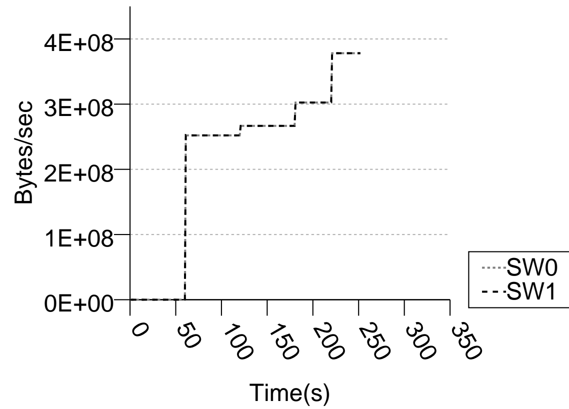


Figure 4.2: Shows pings from an attacker overwhelming the server

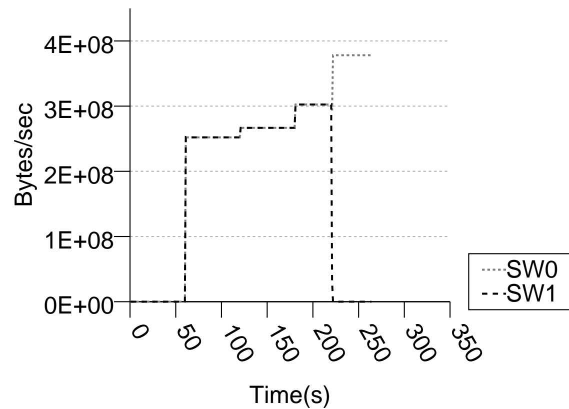


Figure 4.3: Shows pings from an attacker being blocked by the intermediate switch SW0

complex implementation as shown in subsection 4.6.3 the traffic could be isolated and forwarded to another network device like IDS where it could be inspected and then forwarded.

4.6.2 Crossfire Attack Emulation

Crossfire attack [27], a type of DoS link-flooding attack, is a large scale attack which cuts off internet to a particular geographic area (Target area). As observed from the power law for internet routes [28], there is a narrow waist (small set of high bandwidth links) to carry the traffic to the target area and its geographical neighbors. In order to attack the target area, the attacker first identifies a set of links (Target links), which are shared between the target area and the public server around the

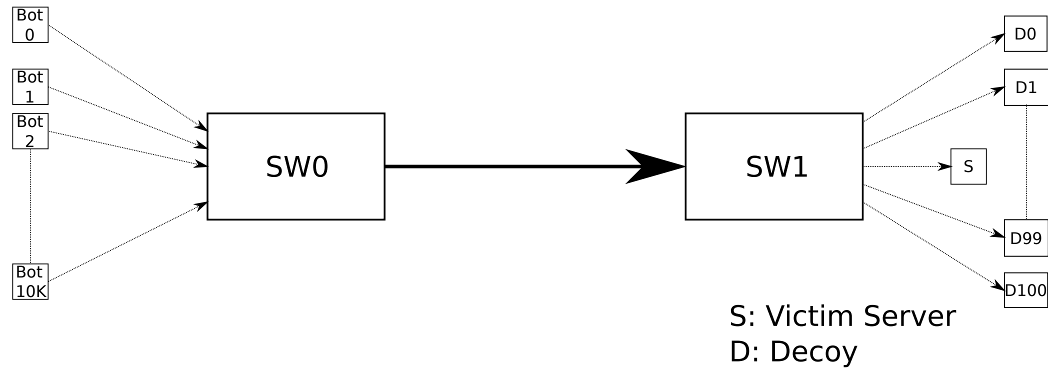


Figure 4.4: Scenario showing crossfire attack

target area called the Decoy servers using tools such as traceroute. Then, a large collection of bots [29] send low intensity traffic to the decoy servers, such that the target links are flooded and the target area becomes the victim. This attack is hard to mitigate since the target area cannot observe the flood directly, yet it is disconnected from the internet.

Consider an example where the aim is to attack server S and there are 100 decoy servers, D0 — D99 as shown in Figure 4.4. Further, consider a link of 10Gbps carrying traffic to all the servers, called the critical link. Now in order to attack server S, the attacker must flood this link by sending low intensity traffic to all the neighboring servers. Suppose if there are 10K bots. Then, 10K x 100 x 10kbps = 10Gbps. Thus, traffic of 10Kbps per bot from a set of 10K bots is sufficient to flood the critical link. Further, if the number of bots available or the number of decoy servers increase, then the traffic rate per bot would go down even more.

To demonstrate this attack a scenario was setup as shown in Figure 4.5. Here, Bot Group (BG) represents a cumulative traffic of all bots sending traffic to that link. Users 1, 2 and 3 are the legitimate users sending traffic to the server S, which is the victim server. The bots are made to send traffic to the decoy sever; there could be multiple decoy servers. The link between SW2 and SW3 becomes the critical link. A constant rate was maintained for the user traffic and a packet of length 128B was used. The bot traffic was started at 0 and then increased at time intervals 60s and

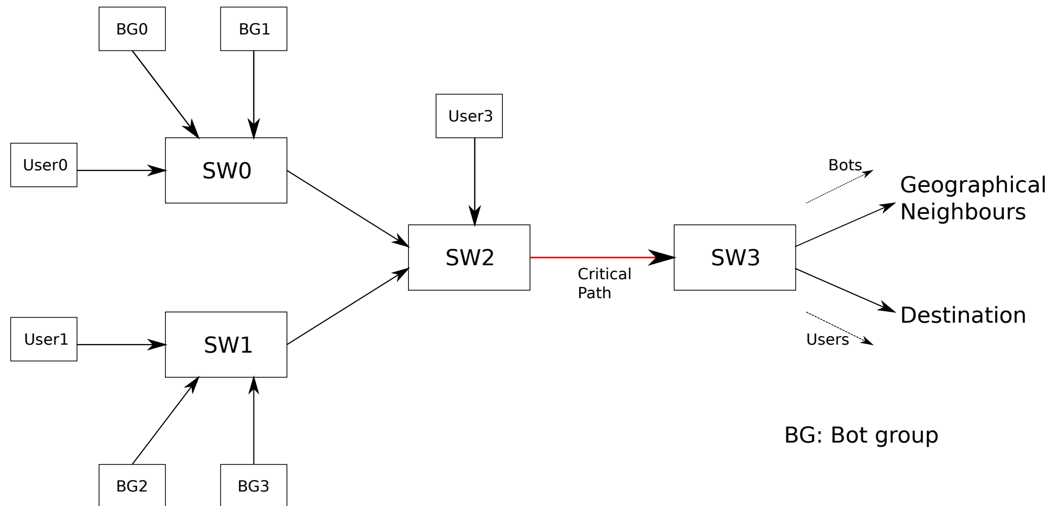


Figure 4.5: Experimental setup to demonstrate crossfire attack

120s.

Figure 4.6, Figure 4.8 and Figure 4.10 show the average byte rate of the traffic from user 1, 2 and 3 respectively. Figure 4.7 and Figure 4.9 show the average byte rate of the traffic from bot group 1 and bot group 3 respectively. The bot traffic increases with time and floods the critical link between SW2 and SW3 and blocks user1 and user2. But, user3 is not affected by the bot traffic because the round robin arbiter in SW2 provides 1/3rd of the outgoing bandwidth to user3. Most of the traffic from user1 and user2 gets dropped at the input of SW2. Flooding of the links can be clearly observed with high packet drop rates at port0 and port1 for SW2 as shown in Figure 4.11 and the high packet drop rate at port0 for SW3 as shown in Figure 4.12

4.6.3 Route Isolation

SDN architecture provides the ability to modify the routes on the fly. This is a key property which provides an inherit security to the SDN based network. Some examples of the usages of this property are shown below —

- An application running on the controller can identify some malicious activity based on the data that is collected across the network. In response, routes of these malicious flows could be reconfigured through security devices for further

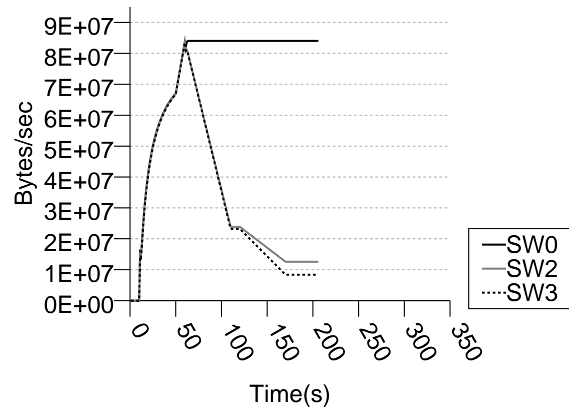


Figure 4.6: User1 average byte rate

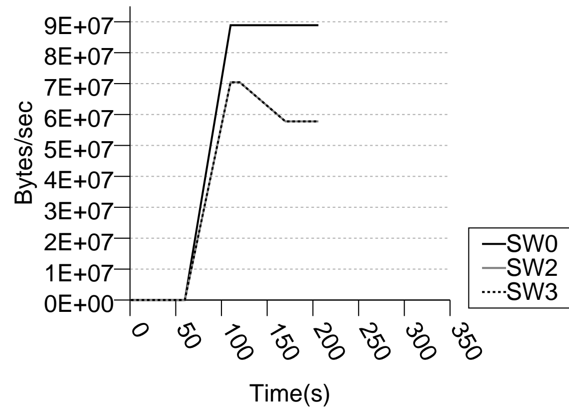


Figure 4.7: Bot group2 average byte rate

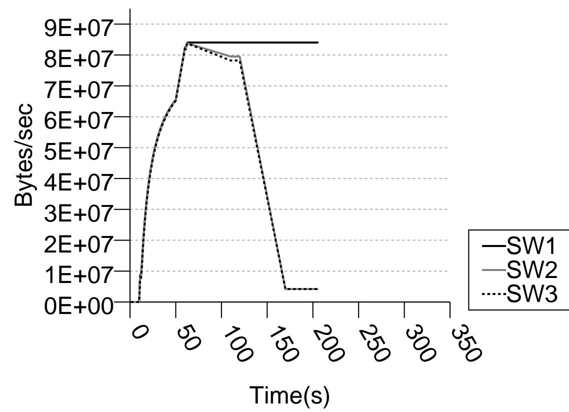


Figure 4.8: User2 average byte rate

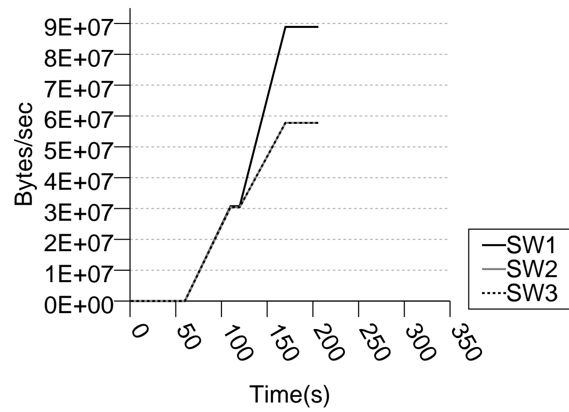


Figure 4.9: Bot group 3 average byte rate

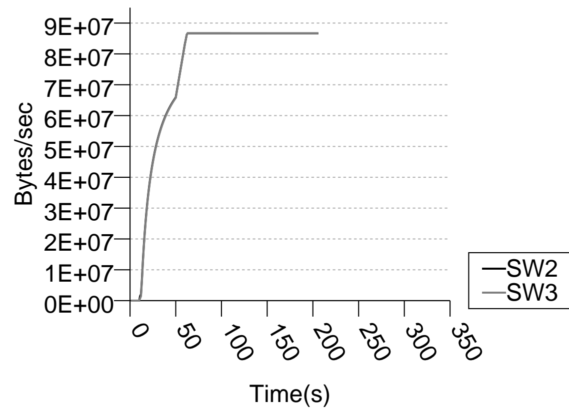


Figure 4.10: User3 average byte rate

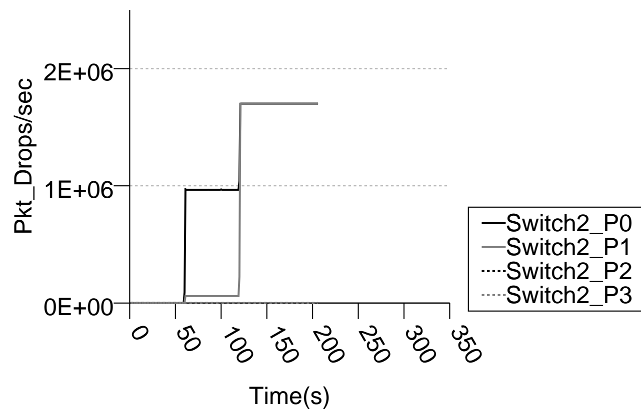


Figure 4.11: Per port packet drops in Switch2

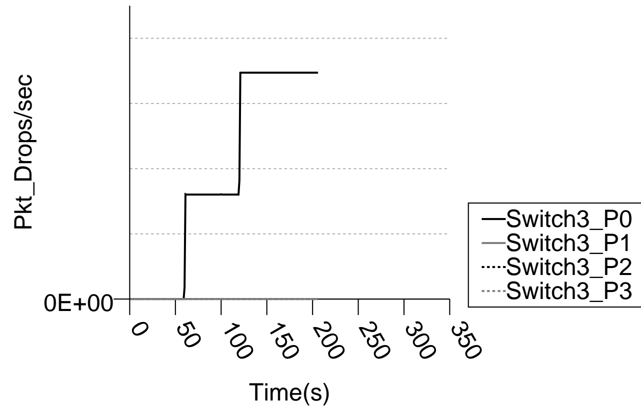


Figure 4.12: Per port packet drops in Switch3

inspection.

- Similarly, a flow could be routed on the fly through a low risk path to provide additional security.
- To identify and mitigate a large scale attack. The authors in [30] use this property against crossfire attack.

In this experiment, on the fly route modification was tested. The setup was as shown in figure Figure 4.13. The routes of flows are given below.

```
FLOW0, FLOW1: SW0 --> SW2 --> SW3
FLOW2, FLOW3: SW1 --> SW2 --> SW3
```

FLOW1 and FLOW3 are considered as malicious traffic; FLOW0 and FLOW2 are considered as legitimate traffic. With this, the attack scenario and the flow handling scenario are described below.

Attack scenario: Send high bandwidth traffic to the network.

Flow handling: Reroute the malicious traffic. Traffic reroute could be done to pass the malicious traffic through a security device or completely block the traffic.

New routes for FLOW1 and FLOW3 are shown as FLOW6 and FLOW7. Both the flows are redirected via port 1 on the respective switches. Figure 4.14 and Figure 4.15

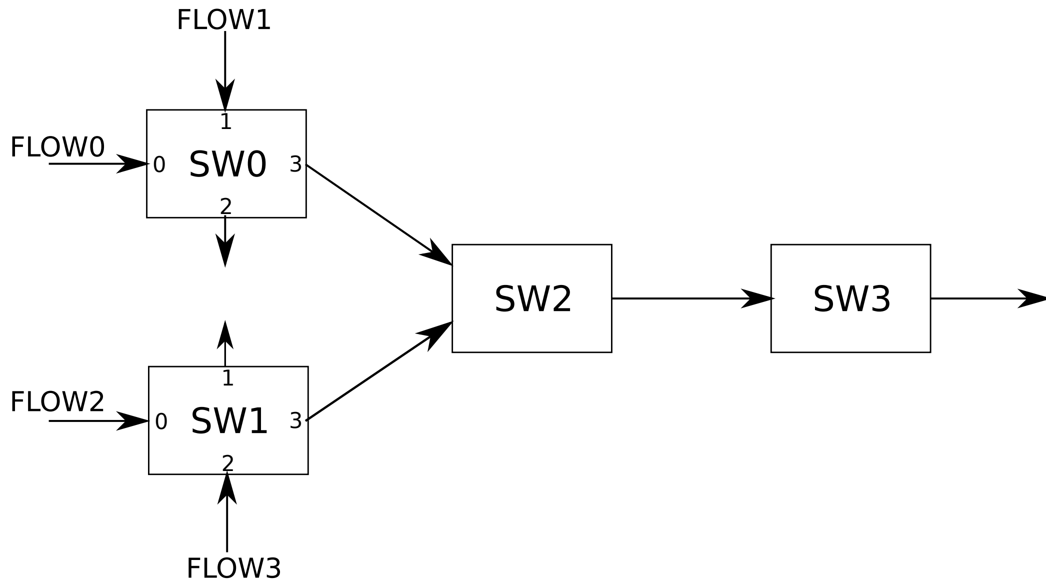


Figure 4.13: Experimental setup to demonstrate route isolation

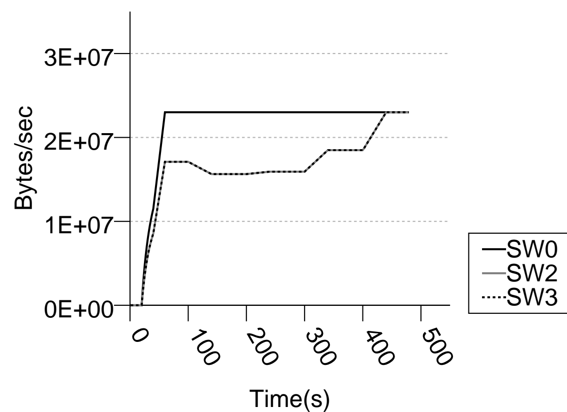


Figure 4.14: FLOW0 — legitimate traffic

show average byte rates of the legitimate traffic. As long as this traffic is mixed with the malicious traffic, the rate is less. After rerouting the malicious traffic, the rate increases. Figure 4.16 and Figure 4.17 show the rate of malicious traffic before and after rerouting through SW0 and SW1 respectively. Figure 4.18 clearly indicates the reduction in the packet drop rate in SW2 after rerouting the malicious traffic.

This experiment demonstrates the flexibility in the preset design to allow researchers to conduct experiments, which requires on the fly isolation of intrusions. However, the flow handling must be specified at the compile time. In future, it could

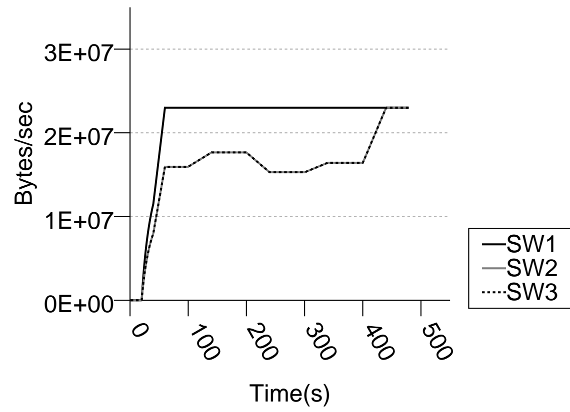
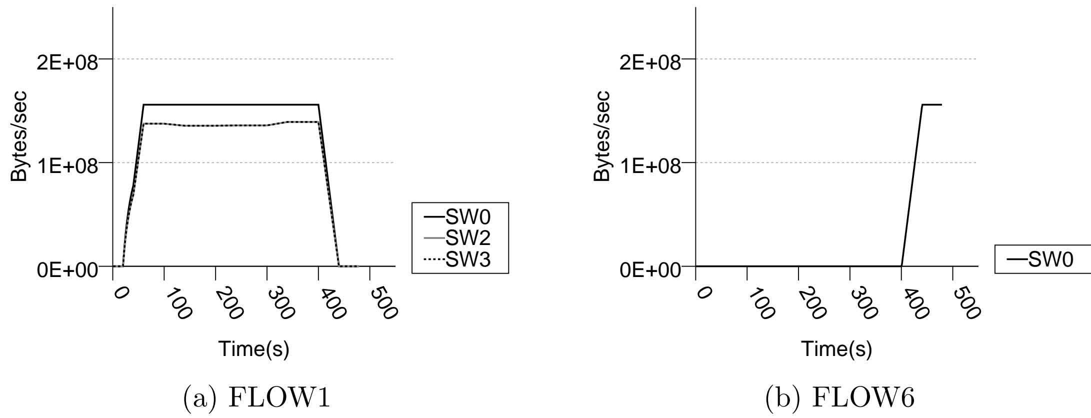


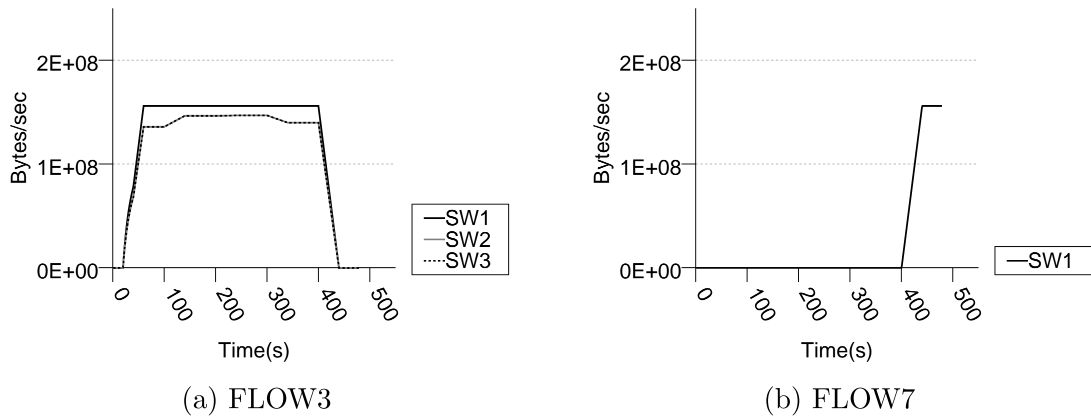
Figure 4.15: FLOW2 — legitimate traffic



(a) FLOW1

(b) FLOW6

Figure 4.16: FLOW1 and FLOW6 — malicious traffic



(a) FLOW3

(b) FLOW7

Figure 4.17: FLOW3 and FLOW7 — malicious traffic

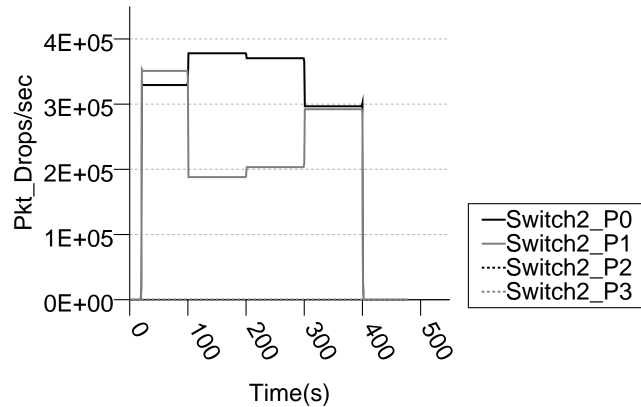


Figure 4.18: Packet Drops in Switch2

be made dynamic, where an application running on the controller would get the data from the switches and dynamically modify the flow.

4.6.4 Random Route Mutation

This experiment is a simple case study of the work shown in [31]. Most of the forwarding routes in the internet today are static. Dynamic route handling is used only for load-management or reliability in case of equipment failures. These methods are predictable and offer significant advantage for the attackers to eavesdrop or launch DoS attacks. Random route mutation (RRM) describes a technique in which the routes of multiple network flows are changed randomly in order to defend against reconnaissance, eavesdrop and DoS attacks.

In order to test the feasibility of implementing RRM on the current emulation design we consider a simple scenario as shown in Figure 4.19. This network is an extension of the network shown in Figure 4.13. There are two packet flows from legitimate users (UserA and UserB) and two packet flows from the attackers (BotA and BotB). Due to high bandwidth of the attack traffic, if the user and bot share a common link, then the user traffic would get dropped by a large extent. For example, if FLOW0 (user1 traffic) and FLOW1 (BotA traffic) are the statically configured routes to reach the destination for UserA and BoatA respectively. Then, UserA throughput would be almost 0. In this experiment, it could be shown that a higher

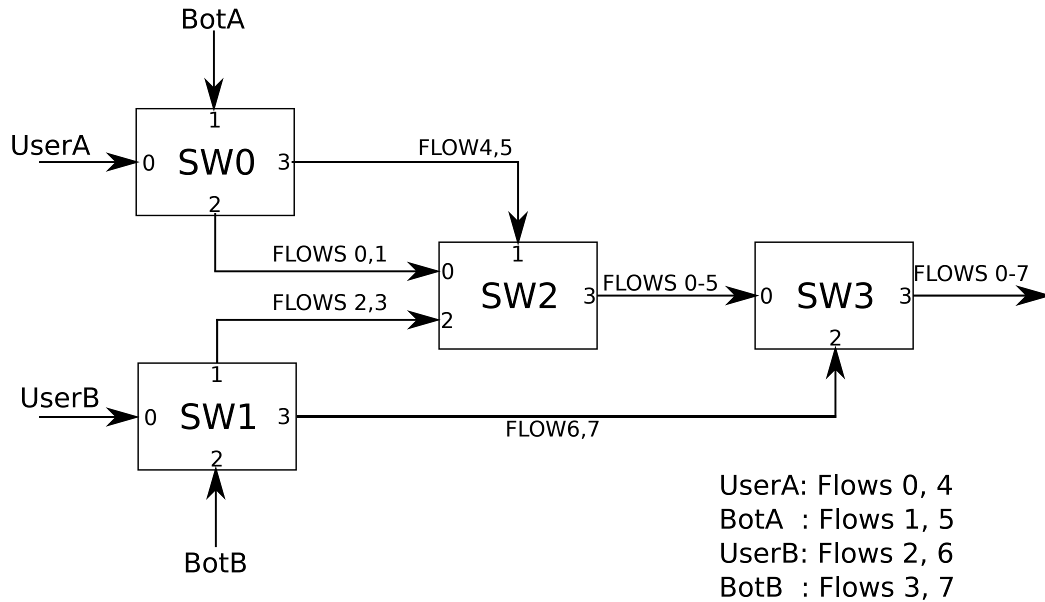
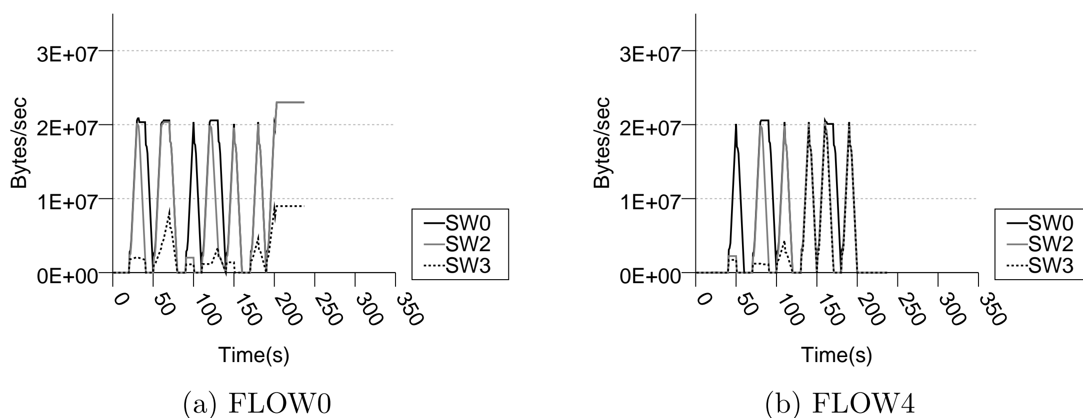


Figure 4.19: Experimental setup to demonstrate random route mutation

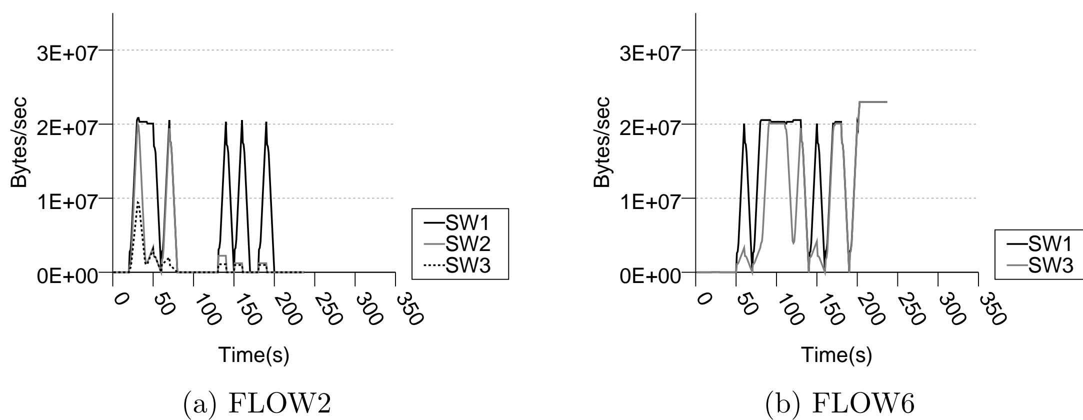
throughput of the user traffic is achieved due to randomizing the flow between source and destination as opposed to having static routes. This works well when the network administrator is unable to identify the attack flows, but is aware that the network is under attack.

For each packet flow, there are two routes to reach the destination as shown in the Figure 4.19. In the experiment, for every 10s, the route between source and destination for a flow was randomized. The flows were started on the 20th second and the flow randomization was done till 200s with a 10s interval. For each flow, the bandwidth at switch3 (SW3) indicates the total bandwidth that is reaching the destination. Evaluation of the throughput for User1 — the bandwidth of user traffic arriving at the network is equal to the bandwidth in switch 1 (SW1). For UserA, this is around 160 Mbps. The attack traffic from BotA arriving in to the network is around 2Gbps. Plots for UserA traffic — FLOW0 and FLOW4 is shown in Figure 4.20. Due to randomization, the UserA traffic is not completely dropped. Infact, between 130s and 170s the UserA throughput in switch 0 (SW0) and switch 3 (SW3) are equal. Similar observations can be made on UserB traffic in switch 3.



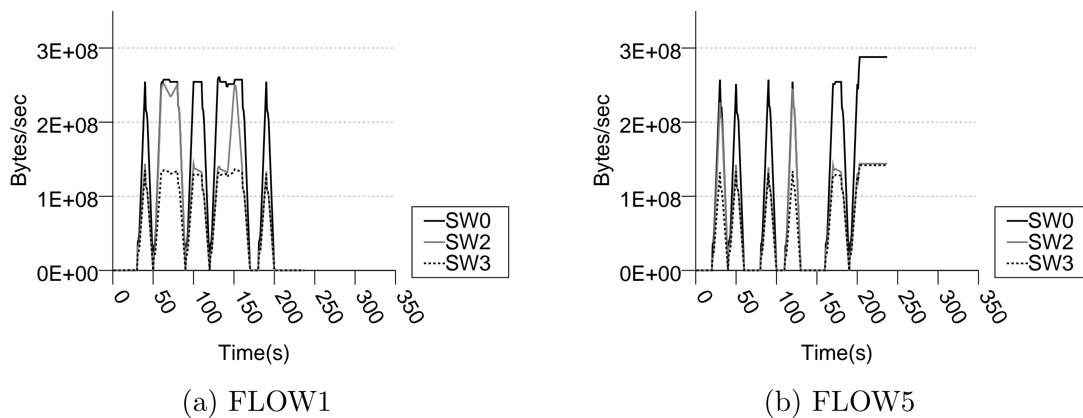
(a) FLOW0 (b) FLOW4

Figure 4.20: UserA throughput



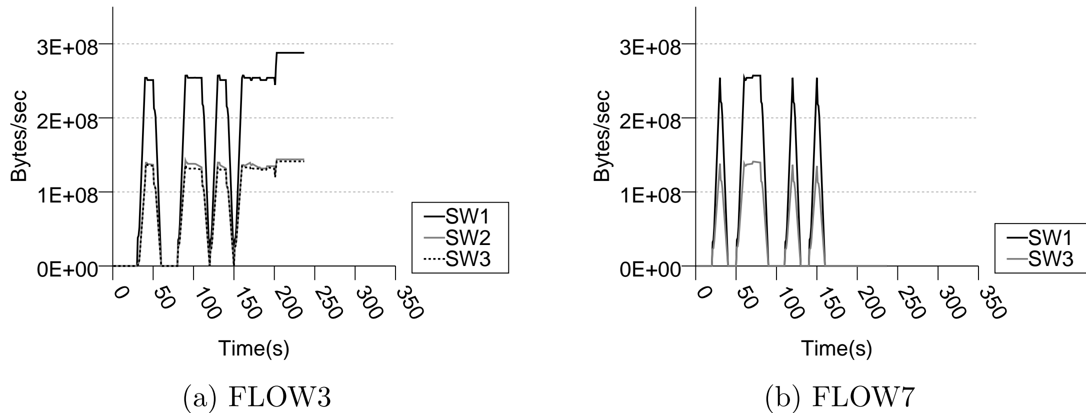
(a) FLOW2 (b) FLOW6

Figure 4.21: UserB throughput



(a) FLOW1 (b) FLOW5

Figure 4.22: BotA throughput



(a) FLOW3 (b) FLOW7

Figure 4.23: BotB throughput

In this experiment, randomizing the flows aids in assuring some user throughput between the source and the destination. With larger FPGA boards, more number of nodes would be available and full implementation of the work shown in RRM can be tested easily; this is a part of the future work.

4.7 Analysis

In this section, we analyse the results obtained from the experiments to identify if our solution answers the thesis questions, i.e., whether the hardware emulator is a legitimate tool which provides a promising solution for the future of science of security. We address the performance of the current emulator, scalability, repeatability and fidelity of the existing tool.

- Performance: Table 4.5 shows the speedup factor of the current implementation with respect to Mininet. From our experiment in section 4.4, we can see that the hardware can support a maximum bandwidth of 950Mbps for a packet length of 128B. The experiment was done with a clock frequency of 100MHz. On a modern board we would be able to increase the clock frequency, further increasing the bandwidth that the design can support. The results clearly indicate that the hardware can support high bandwidth emulation.
- Scalability: The existing emulation contained a maximum of 4 switches, which

is very less for a large scale emulation. However, a modern board like Xilinx Virtex-7 [32] contains ten times more FPGA resources. Using a board like that would mean we can easily implement around 40 switches. Further, if we consider a cluster of FPGAs, say, 16 FPGAs — then the number of switches would be 640. Also, a single Microblaze processor can emulate multiple entry points and multiple attackers indicating that a single Microblaze can support a large collection of switches on the board. Therefore, scalability is just a question of how much hardware resources are available. For a very large scale emulation, we might have to consider a cluster containing a large set of FPGAs. Of course, there is a limitation on how many boards can be used at once due to inter-FPGA communication. This is a part of the future work.

- Repeatability: All the experiments in the current design are controlled and are completely reproducible. For a given input set, same results are obtained.
- Fidelity: We showed different experiments to establish the breadth of the current design. Crossfire attack emulation in subsection 4.6.2 was an example of a large scale attack. The graphs obtained from our sample design is synchronous with the theory explained in [27]. We implemented an attack scenario and defense strategy for ping of death (subsection 4.6.1), route isolation (subsection 4.6.3) and random route mutation (subsection 4.6.4). The experiments were conducted for high input data rate and were "near" real-time. The data collected accurately described the nature of each of these attacks. The effectiveness of the defense strategy used can also be measured from the data collected by the controller.
- Accessibility: The existing design requires the users to enter information into the CSV files. The input format mainly includes the time stamp and the set of events that has to be executed at that time stamp. Apart from that, the hardware switches must be connected in the required topology. This can be

done using Xilinx XPS GUI. This provides a good level of abstraction for any user without having hardware background to use the tool.

CHAPTER 5: CONCLUSION

The initial prototype of the hardware emulator with SDN network components, attack and defense strategy shows promising results for a large scale and high bandwidth emulation. We emulate four attack scenarios. In three experiments, the defense strategy was also emulated. These experiments prove the high capacity, high fidelity and real-time properties of hardware emulator. Now, the only restriction was the size of the board to perform the same experiments with large number of nodes. With a newer board and having a cluster of FPGAs, we can easily overcome this restriction. From researcher's point of view, this work will provide a platform to test different network properties, attack scenarios, defense tactics and observe real-network behavior. Eventually, providing an effective tool for the science of security and aiding in making the networks rigid against the adversaries.

5.1 Future Work

The work presented in this thesis was an initial prototype to evaluate the feasibility of hardware emulators. The immediate work on the current design would be to provide wildcard matching functionality in the hardware, i.e., mask out some of the fields in the match field (flow header) and then perform the match. This would increase the flexibility of the existing design. On the software side, a scenario generator could be implemented, that provides a higher level of abstraction for the user and populates the CSV files as required by the hardware. Along with this, the existing controller could be made reactive where — (a) some event in the collected results could trigger additional events, and (b) a switch can communicate with the controller if a new flow is received.

A major enhancement of the current design that would really aid network pro-

fessionals would be to provide the flexibility to run controllers like POX, NOX or flow-visor. The controller could be made to communicate with hardware switches on the board. Since, there is a lot of research and development happening on these controllers, this seems to be a viable solution for long-term use. Finally, implementing the emulator on a cluster of FPGAs would help in emulating a very large scale network.

REFERENCES

[1] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik, “Shrink: a method for enabling scaleable performance prediction and efficient network simulation,” *IEEE/ACM Transactions on Networking (TON)*, vol. 13, no. 5, pp. 975–988, 2005.

[2] A. Roy, K. Yocum, and A. C. Snoeren, “Challenges in the emulation of large scale software defined networks,” in *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 2013, p. 10.

[3] O. S. Specification-Version, “1.4. 0,” 2013.

[4] O. N. Fundation, “Software-defined networking: The new norm for networks,” *ONF White Paper*, 2012.

[5] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, “Software-defined networking (SDN): Layers and architecture terminology,” Tech. Rep., 2015.

[6] S. A. Mehdi, J. Khalid, and S. A. Khayam, “Revisiting traffic anomaly detection using software defined networking,” in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 161–180.

[7] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, “Fresco: Modular composable security services for software-defined networks.” in *NDSS*, 2013.

[8] D. Kreutz, F. Ramos, and P. Verissimo, “Towards secure and dependable software-defined networks,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 55–60.

[9] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “Nox: towards an operating system for networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[11] J. Mccauley, “Pox: A python-based openflow controller,” 2014.

[12] “Floodlight openflow controller,” projectfloodlight.org/floodlight.

[13] S.-Y. Wang, C.-L. Chou, and C.-M. Yang, “Estinet openflow network simulator and emulator,” *Communications Magazine, IEEE*, vol. 51, no. 9, pp. 110–117, 2013.

[14] G. Carneiro, "Ns-3: Network simulator 3," in *UTM Lab Meeting April*, vol. 20, 2010.

[15] "Netfpga official web site," netfpga.org.

[16] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster, "Switchblade: a platform for rapid deployment of network protocols on programmable hardware," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 183–194, 2011.

[17] "The open flow switch consortium," opennetworking.org.

[18] "Netfpga 10g openflow switch," github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch.

[19] G. Antichi, A. D. Pietro, S. Giordano, G. Procissi, and D. Ficara, "Design and development of an openflow compliant smart gigabit switch," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE.* IEEE, 2011, pp. 1–5.

[20] A. Khan and N. Dave, "Enabling hardware exploration in software-defined networking: A flexible, portable openflow switch," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on.* IEEE, 2013, pp. 145–148.

[21] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems.* ACM, 2008, pp. 1–9.

[22] "Open vswitch, an open virtual switch," openvswitch.org.

[23] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The tcp/udp bandwidth measurement tool," *htt p://dast. nlanr. net/Projects*, 2005.

[24] G. A. Covington, G. Gibb, J. W. Lockwood, and N. Mckeown, "A packet generator on the netfpga platform," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on.* IEEE, 2009, pp. 235–238.

[25] C. C. Center, "Cert advisory ca-1998-01 smurf ip denial-of-service attacks," 1998.

[26] V. Paxson, "An analysis of using reflectors for distributed denial-of-service attacks," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 3, pp. 38–47, 2001.

[27] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 127–141.

[28] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[29] R. Vogt, J. Aycock, and M. J. Jacobson Jr, “Army of botnets.” in *NDSS*, 2007.

[30] D. Gkounis, V. Kotronis, and X. Dimitropoulos, “Towards defeating the crossfire attack using sdn,” *arXiv preprint arXiv:1412.2013*, 2014.

[31] Q. Duan, E. Al-Shaer, and H. Jafarian, “Efficient random route mutation considering flow and network constraints,” in *Communications and Network Security (CNS), 2013 IEEE Conference on.* IEEE, 2013, pp. 260–268.

[32] “Xilinx virtex-7 family,” xilinx.com/products/silicon-devices/fpga/virtex-7.html.