Hindawi
Security and Communication Networks
Volume 2018, Article ID 9562801, 14 pages
https://doi.org/10.1155/2018/9562801

Research Article

An Efficient High-Throughput and Low-Latency SYN Flood Defender for High-Speed Networks

Duc-Minh Ngo, Cuong Pham-Quoc, and Tran Ngoc Thinh

Ho Chi Minh City University of Technology, VNU-HCM, 268 Ly Thuong Kiet Street, District 10, Ho Chi Minh City, Vietnam

Correspondence should be addressed to Cuong Pham-Quoc; cuongpham@hcmut.edu.vn

Received 16 October 2018; Accepted 9 December 2018; Published 24 December 2018

Guest Editor: Eiji Kamioka

Copyright © 2018 Duc-Minh Ngo et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

As one of the main types of Distributed Denial of Service (DDoS) attacks, SYN flood attacks have caused serious issues for servers when legitimate clients may be denied connections. There is an essential demand for a sufficient approach to mitigate SYN flood attacks. In this paper, we introduce an efficient high-throughput and low-latency SYN flood defender architecture, carefully designed with a pipeline model. A mathematical model is also introduced with the architecture for estimating SYN flood protection throughput and latency. The first prototype version based on the architecture with Verilog-HDL can function as standalone to alleviate high-rate SYN flood attacks and can be integrated into an OpenFlow switch for handling network packets. Our experiments with NetFPGA-10G platforms show that the core can protect servers against SYN flood attacks by up to 28+ millions packets per second that outperforms most well-known hardware-based approaches in the literature.

1. Introduction

Along with the rapid development in technology as well as network architectures, cybersecurity becomes a primary issue for organizations such as commercial trades, banks, military networks. Symantec [1] shows that cybercriminals could exploit the communication channels among IoTs (Internet of Things) devices to perform an increasing of 600% in overall IoTs attacks in 2017. According to Cisco Cybersecurity Reports [2], even though global web traffic enhances security by using encryption techniques, there still exist 42% of organizations that have been faced with DDoS attacks. Among them, TCP SYN flood attacks [3] are one of the most popular DDoS attack types and widely used because SYN packets are not likely to be rejected by default. Therefore, a robust SYN flood defender approach is an essential demand.

In the last decades, the expansion of IoTs and local network systems result in the evolution of traditional network architectures. Software-Defined-Networking (SDN) [4] whose control plane is decoupled from the data plane has introduced advantages compared to tradition network architectures such as centralized network provisioning, holistic enterprise management, and lower operating costs [5]. However, SDN architecture suffers from security vulnerabilities in the control plane as well as in the data plane and the communication channel. SDN systems can easily be broken down since TCP SYN attacks flood the communication channel. To overcome this critical security issue, strengthening the processing power of the control plane using software approaches, such as work in [6–8], is well researched. However, the main drawback of these studies is the use of the Controller resources for performing security functions while controllers are targets for saturation attacks. In recent years, bringing some intelligent processes from control planes to data planes becomes a trend in SDN [9, 10] especially when data planes are built in hardware platforms. With smarter data planes, controllers and communication channels are protected from attacks because threats can be prevented.

In this work, we propose an efficient high-throughput and low-latency SYN flood defender for high-speed network. The SYN flood defenders are designed to process incoming packets in a pipeline model for increasing throughput and reducing latency. The proposed architecture is technology-independent so that our SYN flood defender can be implemented by different hardware technologies such as ASIC or FPGA. Moreover, we aim to integrate the proposed SYN flood defender core into SDN data planes; therefore the proposed architecture takes the SDN Protocol and architecture into

account. We implement the proposed SYN flood defender architecture by Verilog-HDL without using any Intellectual Property (IP) core. The core is integrated into an FPGA SDN-based OpenFlow switch built in our previous work [11]. A number of testing scenarios are conducted with both synthetic data and real network data using the switch to verify and evaluate the proposed architecture and core. The main contributions of this work are summarized in two folds.

(1) We propose an efficient high-throughput SYN flood defender hardware architecture that can analyze network packets to prevent high-rate SYN flooding attacks. The SYN flood defender architecture can be executed in pipeline to improve throughput and performance. We also present a mathematical model for evaluating performance as well as bandwidth of the core. This estimation model is then validated by our experiments.

(2) We implement the proposed architecture with Verilog-HDL without using any IP core so that the SYN flood defender core can be built on different technologies. The core is integrated into our FPGA SDN-based OpenFlow switch which is built on the NetFPGA-10G board [12]. To the best of our knowledge, this is the first SDN-based OpenFlow switch with SYN flood defender integrated that could prevent SYN flood attacks with attack rates by up to more than 28 million packets per second with maximum 0.248 ms latency.

The rest of this work is organized as follows. Section 2 presents background and discusses related work. Section 3 introduces our proposed SYN flood defender architecture and a mathematical model for evaluating system performance. Section 4 illustrates implementation of the proposed SYN flood defender core and our FPGA SDN-based OpenFlow switch. We evaluate and analyze the system in Section 5. Finally, conclusion and future work are discussed in Section 6.

2. Background and Related Work

In this section, we first present background of TCP SYN flood attacks based on that we propose our efficient high-throughput and low-latency SYN flood defender for high-speed network. We then introduce related work in the literature which mainly focuses on preventing SYN flood attacks.

2.1. Background. The TCP SYN flood attacks mechanism exploits the TCP three-way handshake Protocol to acquire resources of target servers and to prevent legitimate clients from provided services. Figure 1(a) describes the three-way handshake process of a normal server while Figure 1(b) represents a SYN flood defender system. Different from normal forwarding devices, a SYN flood defender represents the protected server to feedback SYN-ACK packets with SYN cookies technique applied [16]. The defender then authenticates ACK packets with a proper mechanism (RST packets) discussed later in Section 4. The validated clients after receiving RST packets make the next SYN packets to access the protected system.

For preventing SYN flood attacks, the defense systems could be placed at source sides, victim sides, or network sides [17]. The weakness of common SYN flood prevention systems deployed at source sides is the massive consumption of resources during high-rate attacks. Software-based approaches, such as [16, 18, 19], aim to minimize stored information but increasing latencies of network packets. In contrast, hardware-based approaches, comprising Field Programmable Gate Arrays (FPGA) [20] or Application-Specific Integrated Circuit (ASIC) [21] for parallel processing, have been used as efficient platforms for building SYN flood defense systems. The main advantages of hardware approaches are parallel processing and low latencies, suitable for protecting against high-rate SYN flood attacks. However, there still exist some limitations such as low scalability, high implementation cost, and high complexity which have not optimized the design yet.

2.2. Related Work. In the literature, there exist some studies that focus on TCP SYN flood attack prevention. Hop-count filtering [22, 23] creates an IP-addresses-mapping table and observes the hop-count (TTL) of packets to detect attacks when the hop-count is not matched with the expected value. Another mechanism, IP puzzles [24], makes the server send a puzzle to source clients to request solving these puzzles for authentication. TCP probing for reply argument packets [25] recognizes spoofing packets by requesting sources to change the window size value of packets. The existing TCP SYN flood attack prevention mechanisms always require protected systems to store clients information while waiting for the authentication process completed; thus reducing memory capacity of systems. In another aspect, Bernstein proposes a SYN cookies technique [16], which is widely used because of the ability to reconstruct the Packet just based on the available information. The work in [26] makes an experimental study on the application of SYN cookies and also reviews two SYN cookies implementations on Linux and FreeBSD operating systems. SYN cookies on Linux encode an initial Cookie number using a timestamp and a cryptographic hashing value while in FreeBSD, a combination of SYN cache and SYN cookies technique is applied. These two operating systems use different kinds of hash function to generate the cryptographic hash value; the Linux one even inserts a secret key to strengthen this value.

Software-based approaches such as SYNkill [18] prevent the waste of resources by generating RST and ACK packets based on client requests. Ingress Filtering [27] blocks attacks by comparing source addresses with the stored white-list to forward packets, while STONE [28] aggregates all the addresses and compares with regular traffic. SYNMON [29] uses a network processor and CUSUM method to detect attacks. D-Ward [30] builds three units including observation, traffic policing, and rate limiting and then performing a rate limiting method to mitigate attacks. The work in [31] applies the Bloom filter to store traffic information

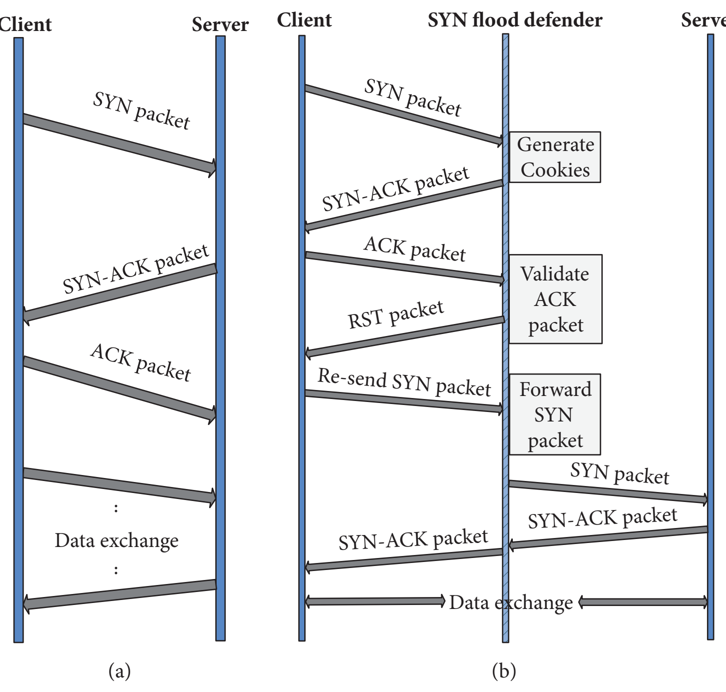


Figure 1: TCP three-way handshaking Protocol of (a) traditional system and (b) SYN flood defender system together with protected server.

and to detect attacks using the CUSUM method and then mitigating attacks using both ingress filter and rate limiting mechanism. The Bloom filter is used in [32] to detect attacks while clients have to retransmit requests to the server during attacks.

Other mechanisms to prevent TCP SYN flood attacks are to exploit network architectures. VSSF [17] requires clients to send ACK packets before forwarding packets through a registration table on an FPGA platform, thus consuming hardware resources and producing latency during high-rate attacks. SLICOTS [13] monitors Packet rate and installs temporary forwarding rules until the client is authenticated to mitigate attack packets in the Controller. Avant-Guard [14] builds a connection migration module to migrate SYN flood attacks. This module represents the server and uses the SYN cookies technique to complete the TCP three-way handshake. Instead of sending RST packets after authenticating connections like our proposed mechanism, the connection migration reproduces SYN packets and performs TCP three-way handshake to the server. As a result, Avant-Guard needs to synchronize sequence numbers of connections by storing differences between two sequence numbers and has to modify packets belonging to these connections. This behavior will produce high overhead in terms of Response time. Research in [15] combines the SYN cookies technique and changes flow tables of SDN data planes against SYN flood attacks. The authors propose to take advantages of flow tables in SDN to store some switching rules, which contain unknown initial Cookie numbers, then using the ability of OpenFlow Protocol to modify packets. This mechanism will reduce computing resources but consuming more space in flow tables.

3. System Design

In this section, we present our hardware-based SYN flood defender architecture that could be used as an extension in other systems such as Intrusion Detection Systems (IDS) and OpenFlow Switch, to prevent high-rate SYN flood attacks. In contrast to other approaches in the literature, the proposed core will not reduce Packet processing performance of the protected systems due to pipelining processing model. We also present a mathematical model for estimating both throughput and latency of our proposed hardware SYN flood defender architecture.

3.1. Overall Architecture. For integrating purpose, our SYN flood defender core is able to receive data from two different sources: networks and an associated host Controller. Data coming from the host Controller will configure the core as well as collect statistical information. Data coming from networks are packets with different fields which need to be processed. Figure 2 introduces our proposed hardware-based SYN flood defender architecture which can function in five pipeline stages to improve throughput. The stages are grouped into five processes; each of them is responsible for one particular step according to the SYN flood defense algorithm. The following sections discuss main processes of the core.

3.1.1. Header Extraction. The Header extraction, done by the `Header Extractor` module, is mainly responsible for analyzing incoming packets to collect required Header fields. Those extracted fields are forwarded to three modules in the next stage for further scanning. Full packets are also stored

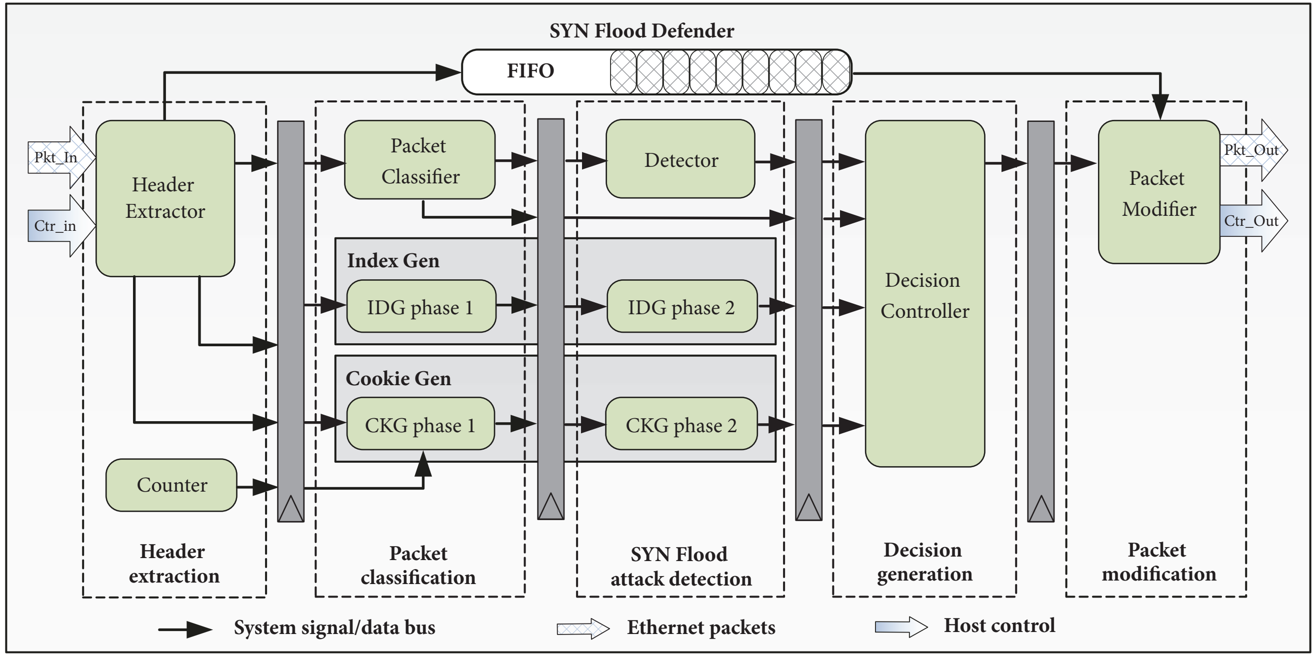


Figure 2: Hardware architecture of SYN flood defender in a five-stage pipeline.

in FIFO to further process after examined. In this process, a 32-bit Counter module plays an important role in generating numbers that will be used to create Cookie numbers according to the TCP three-way handshaking Protocol. These Cookie numbers must be different from cycle to cycle. The generated number is forwarded to the next pipeline stage.

3.1.2. Packet Classification. After receiving Header fields of incoming packets generated by the Header extraction process in the first pipeline stage, the Packet classification process categorizes packets into three different groups including SYN packets, ACK packets, and other packets, done by the Packet Classifier module. Classified results are then forwarded to the Detector module, located in the third stage, to recognize SYN flood attacks. Along with categorizing packets, the Packet classification process also starts generating *Index* and *Cookie* numbers by using our proposed hashing algorithm. The two subprocesses *Index Gen (IDG)* and *Cookie Gen (CKG)* span two stages, the second and the third stages, by using two modules for each, IDG phase 1 and IDG phase 2 for the former and CKG phase 1 and CKG phase 2 for the latter. While IDG phase 1 and CKG phase 1 receive specific Header fields extracted by the *Header extraction* process, the two other modules generate numbers by using hashing techniques.

3.1.3. SYN Food Attack Detection. This process, performed by the Detector module, is mainly responsible for detecting SYN flood attacks based on information from the Packet Classifier module. This module can be considered as the heart of the proposed SYN flood defender core because further processing to deal with SYN flood attacks depends on determined results of the module. Results generated by this module, called SYNFlood signal, alert other modules in the core if there exist SYN flood attacks.

In this work, we propose to assert the SYNFlood signal when one of the following conditions happened. (1) *Condition 1:* the number of SYN packets in an interval exceeds a threshold; (2) *Condition 2:* the number of SYN packets is greater than the number of ACK packets a threshold in a particular interval. Both *threshold* and *interval* values are determined by network administrators through the associated host Controller. In other words, these values are reconfigurable.

More details of this checking process are presented in Algorithm 1. In this algorithm, the number of packets (SYN and ACK, called S and A, respectively) in a particular interval (called $INTERVAL$) is updated accordingly with Packet types represented in TCP_flag. At the end of an interval ($I = INTERVAL$), the two aforementioned conditions are checked to determine the status of the system (under attacks or not). The counter variables for SYN and ACK packets are also reset at that time. When one of the conditions is satisfied, the SYNFlood signal is asserted at least in one next interval. This signal is deasserted at the end of an interval when both conditions are not true for the current interval.

3.1.4. Decision Generation. This process, executed by the Decision Controller module in the fourth stage of the pipeline architecture, is responsible for synthesizing results from the Packet Classifier, Detector, IDG phase 2, and CKG phase 2 modules for making decisions on incoming network packets. Based on information from the SYN flood attack detection process, the Decision Controller module issues one of the following results to handle the Packet modification process in the next stages: (1) converting

```
Input: TCP_flag
Parameter: ACK_DIFF, INTERVAL and SYN_PKT
Output: SYNFlood
1  S = 0; // the number of SYN packets
2  A = 0; // the number of ACK packets
3  I = 0; // Interval
4  while true do
5      if incoming packet then
6          if TCP_flag = SYN_flag then
7              S = S + 1;
8          if TCP_flag = ACK_flag then
9              A = A + 1;
10     if I = INTERVAL then
11         I = 0;
12         if (S ≥ SYN_PKT) || (S ≥ A+ACK_DIFF) then
13             SYNFlood = true; // attacks exist
14         else
15             SYNFlood = false; // attacks do not exist
16         S = 0;
17         A = 0;
18     else
19         I = I + 1;
20 return SYNFlood
```

Algorithm 1: SYN flood attack detection.

SYN packets into SYN-ACK packets to handshake with clients when there exist SYN flood attacks; (2) generating reset (RST) packets to recognized clients in order to make connections of these clients and the protected server; (3) bypassing SYN packets to the protected server when they belong to legitimate clients. Decision results together with Header fields are then forwarded to the next process, Packet modification for further processing. Along with the SYN flood attack detection process, the decision generation process plays a key role in protecting the server against SYN flood attacks, but still, allowing truth clients to connect to the protected server. While the `Detector` module keep scanning SYN packets to identify attacks, the `Decision Controller` module, on behalf of the server, performs the 3-way TCP handshaking Protocol with clients in case attacks are happening. When attacks do not exist (the `SYNFlood` signal is deasserted), SYN packets are bypassed directly to the protected server so that connections between clients and the server can be made without delay. More details of this process are presented in Algorithm 2.

The decision generation process, at first, examines results generated by the `Detector` module. While there exists at least a SYN flood attacking flow, a number of behaviors need to be performed to protect the server, i.e., preventing illegitimate SYN packets arrive the server (lines 2-19 in Algorithm 2). In other words, the SYN flood defender core is activated to verify clients on behalf of the server. In contrast, when the `Detector` module does not identify any attacks, SYN packets are switched directly to the server to reduce latency (lines 20-22 in Algorithm 2). In other words, the core during this period only monitors and forwards incoming SYN packets without making communication to servers.

TCP SYN flood attacks, as presented in Section 2.1, usually exploits the 3-way TCP handshaking Protocol by sending huge amount of SYN packets to target servers so that these servers need to Response by SYN-ACK feedback packets. Because of the huge amount of incoming SYN requests, servers usually are not able to serve other clients. To overcome attacks, i.e., preventing attacking SYN packets arrive the protected server, our core takes over the server to feedback attacking clients (sending SYN-ACK packets to Response SYN packets) while attacks are recognized. Therefore, the server can serve other connected clients. According to the SYN flood attacks method, attackers will not answer to SYN-ACK packets sent by the target server or SYN defender core while normal clients will respond by sending back ACK packets.

During SYN flood attacks period, there may exist safe clients that also want to connect to the server. Therefore, the defender core should be able identify truth clients. This behavior is clearly described in Algorithm 2 (lines 5-9), named *Response* phase. In the Response phase, a SYN Packet coming, the process checks history to see if the Packet arrives from a known client or new client (line 5). A known client means that this client already finished a 3-way TCP handshaking with the core and received an RST Packet from the core before sending the second SYN Packet. SYN packets from known clients are forwarded directly to the protected server (line 7) while SYN packets from unknown clients are converted to SYN-ACK packets to verify unknown clients (line 9). However, after connecting to the target server, a known client may become a SYN flood attacker, for example, due to virus infection. Therefore, the known client is flushed from history to become an unknown client right after connecting to the target server so that new connections made by this client in the future will be verified again (line 6).

To be categorized as a truth client, a client after receiving a SYN-ACK Packet from the target server should reply an ACK Packet with a particular ACK number to the server. This ACK number must be calculated by increasing one from a number, called Cookie number, attached in the SYN-ACK Packet. Based on this ACK number, the target server can verify the client and add it into known clients. We implement this verifying behavior from lines 12 to 16 in Algorithm 2, named *Authentication* phase. After verifying the client (Cookie number for this client is increased and attached in the feedback ACK Packet), the core resends an RST Packet to the client (line 14). The client then starts the second handshaking to connect to the server, since the client already became a known client.

3.1.5. Packet Modification. This is the final process which is mainly responsible for performing decisions determined by the `Decision Controller` module. Similarly to *Header extraction*, this process also takes one stage for processing packets, executed by the `Packet Modifier` module. Full packets at the head of the FIFO are forwarded to the protected server when `bypassing` decisions are issued. Please note that these packets can be either SYN packets from known clients or other Packet types such as payload

```
Input: SYNFlood, Protocol, TCP_flag, ACK_number, Index, Cookie
Output: Decision, Cookie
1 Decision = Bypassing;
2 if (incoming packet) && (SYNFlood = true) then
3     if Protocol = TCP then
4         if TCP_flag = SYN_flag then
5             Response phase:
6             if packet ∈ known clients then
7                 Forget the client;
8                 Decision = Bypassing;
9             else
10                Decision = Converting SYN to SYN-ACK;
11        else if TCP_flag = ACK_flag then
12            Authentication phase:
13            if Cookie = ACK_number - 1 then
14                Add to known clients;
15                Decision = Converting ACK to RST;
16            else
17                Decision = Bypassing;
18 return Decision, Cookie
```

Algorithm 2: Decision generation.

packets from connected clients or control packets generated by the associated host. In contrast, packets at the head of the FIFO are reconstructed with new information and sent back to clients when either `Converting SYN to SYN-ACK` or `Converting ACK to RST` decisions are released. These packets are SYN packets from unknown clients or ACK packets from clients in the verifying process.

3.2. Performance Analysis. Although the SYN flood defender core is designed carefully with a five stages pipeline model and FPGA platforms can be used for implementing the core, system performance can still suffer from a downgrade due to latency in packets examining. In this section, we analyze performance of the proposed architecture in terms of *Latency* and *Bandwidth*. In this work, *Latency* is the percentage (%) of extra packets processing time compared to the original processing time without SYN flood protection. Meanwhile, *Bandwidth* is considered as the maximum throughput (bits per second) that the SYN flood defender core can handle when implemented in an FPGA platform.

3.2.1. Latency. To estimate the average latency of the core, we assume that incoming packets rates do not exceed maximum throughput of the hardware platform used to build the core. In other words, all incoming packets are accepted without any drop. Consider a set of N packets with various Packet length (bits per Packet); these N packets are categorized into m groups according to their size. Let us call n_i the number of packets in group i_{th} ($N = \sum_{i=0}^{m-1} n_i$) and L_i the length of packets in this group. Assume that T_{max} is the maximum throughput of channels connecting clients and the target server, without the SYN flood defender core. The minimal time for all N packets arriving at the target server is calculated by

$$t_{original}^{min} = \frac{\sum_{i=0}^{m-1} n_i \times L_i}{T_{max}} \quad (1)$$

When the proposed core is added to protect the target server, packets are examined by the core before forwarded to the server. Therefore, additional processing time (latency) will be introduced by the core. However, due to the pipeline model, after the first Packet arrives at the server, one Packet is processed every clock cycle. Assume that, K among N packets are SYN packets arriving at the core from unknown sources. The interval for all N packets coming to the server protected by our core is estimated by (assume that these packets are sent continuously to the server).

$$t_{protected} = \frac{\sum_{i=0}^{m-1} n_i \times L_i}{T_{max}} + \frac{D}{f} + C \times K \quad (2)$$

where D represents the number of clock cycles for the first Packet to go through our core (D also is equal to the number of pipeline stages) while f is the clock rate of the hardware platform used to deploy our SYN flood defender. The constant C is the overhead value introduced by the handshaking Protocol between the core and unknown clients. More details of this value are further discussed in Section 5.

The latency issued by the core is calculated based on estimations of $t_{original}^{min}$ and $t_{protected}$ value as shown in

$$\text{Latency} = \frac{t_{protected} - t_{original}^{min}}{t_{original}^{min}} \times 100\% = T_{max} \times \frac{D/f + C \times K}{\sum_{i=0}^{m-1} n_i \times L_i} \quad (3)$$

In (3), the frequency f, number of stages D, and the constant value C depend on system design. Hence, the latency value is

mostly determined by the number of packets N, Packet sizes L_i, and the channel maximum throughput T_{max}.

3.2.2. Throughput. To estimate the maximum throughput that the system is able to accept incoming packets without drops, we assume that the core is deployed in a particular hardware platform. The platform can receive at maximum W bits per clock cycles for processing due to the W-bit bus-width inside the platform. Therefore, incoming packets are fragmented into W-bit frames. It means that, for a particular L-byte Packet, the core requires R clock cycles for receiving as estimated in

$$R = \left\lceil \frac{(L+H)\times 8}{W} \right\rceil \quad (4)$$

where the constant value H value is the number of bytes called the interpacket gap [33] between two packets that is defined by the Protocol used. For the TCP Protocol in IPv4, this value is 4 bytes [33].

The minimum interval $t_{receive}^{min}$ in which the core can receive all N packets as discussed in the previous section, when these packets are streamed continuously to core, is estimated by

$$t_{receive}^{min} = \frac{1}{f}\sum_{i=0}^{m-1} n_i \times R_i = \frac{1}{f}\sum_{i=0}^{m-1} n_i \times \left\lceil \frac{(L_i+H)\times 8}{W} \right\rceil \quad (5)$$

Thanks to pipeline design, the total time for the core to receive and scan all packets is estimated by

$$\begin{aligned} t_{process} &= t_{receive}^{min} + \frac{D}{f} \\ &= \frac{1}{f}\sum_{i=0}^{m-1} n_i \times \left\lceil \frac{(L_i+H)\times 8}{W} \right\rceil + \frac{D}{f} \end{aligned} \quad (6)$$

Therefore, the throughput of the core T_{core} can be estimated as in

$$\begin{aligned} T_{core} &= \frac{\text{Amount of data}}{t_{process}} \\ &= \frac{\sum_{i=0}^{m-1} n_i \times (L_i+H)\times 8}{(1/f)\sum_{i=0}^{m-1} n_i \times \lceil ((L_i+H)\times 8)/W \rceil + D/f} \end{aligned} \quad (7)$$

When DDoS attacks like SYN flooding happen, attackers send extremely large number of packets continuously to shut down the system. In this case, bandwidth of the core estimated in (7) can be simplified as

$$T \approx f \times \frac{\sum_{i=0}^{m-1} n_i \times (L_i+H)\times 8}{\sum_{i=0}^{m-1} n_i \times \lceil ((L_i+H)\times 8)/W \rceil} \quad (8)$$

4. System Implementation

In this section, we introduce our first prototype version of the proposed SYN flood defender core deployed on a particular FPGA platform. We then show the integration of the core into the forwarding plane of an OpenFlow switch for an SDN network.

bit offset	0-3	4-7	8-15	16-18	19-31
0	MAC dst (Destination MAC Address)				
⋮	⋮				
112	EtherType				
144	Version	IHL	Type of Service	Total Length	
176	Identification			Flag	Fragment Offset
208	Time to Live		Protocol	Header Checksum	
240	IP src (IPv4 Source Address)				
272	IP dst (IPv4 Destination Address)			IP dst	
304	TCP/UDP src port			TCP/UDP dst port	
336	Sequence Number				
368	ACK (Acknowledgement) Number				
400	Data Offset	Reserved	U A P R S F	Window Size	
432	Checksum			Urgent Pointer	
464	Options (MSS, ...)				Padding

Header Extractor 1 | Header Extractor 2

Figure 3: Header extraction in two phases of SYN flood defender.

4.1. FPGA-Based SYN Flood Defender Core Implementation. To develop the first prototype version of the proposed SYN flood defender core, we use a NetFPGA-10G board which includes a Xilinx Virtex 5 xc5vtx240t device. To implement the architecture proposed in the previous section, Verilog-HDL is used manually for developing modules so that the core is platform-independently; i.e., it is possible to port the core to other FPGA platforms. For this version, the PCIe interface is used for communicating with the associated host Controller while the four SFP+ ports with maximum half-duplex throughput at 10Gbps per port are used for incoming network packets. Standard Xilinx AXI4 lite [34] is used as the main interconnect inside the core (connecting the input ports with the modules inside the core). Therefore, the frame size (W) in this prototype version is 256 bits (32 bytes). Below, we highlight primary points of the modules inside the core in the following paragraphs.

4.1.1. `Header Extractor`. This module is responsible for extracting Ethernet [35] and TCP/IP Header fields [36] of incoming packets. Figure 3 shows the extracted fields and their positions in a network Packet. Although `Header Extractor` in the generic architecture in Section 3.1 requires only one stage pipeline, we need to divide this module into submodules (`Header Extractor 1` and `Header Extractor 2`, each takes one cycle in one pipeline stage) due to limitation of the NetFPGA-10G platform. The platform only supports 32-byte bus-width [37] while extracted Header fields consist of 64 bytes. These Header fields are then forwarded to the next stages for other modules.

4.1.2. `IDG Phase 1` *and* `IDG Phase 2`. As explained in Section 3.1, the Index Generation process takes two steps,

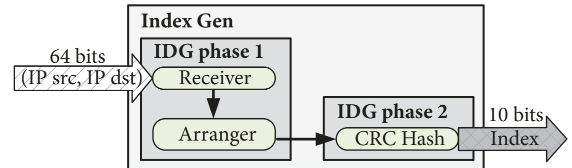


Figure 4: Index generation process with CRC hash algorithm.

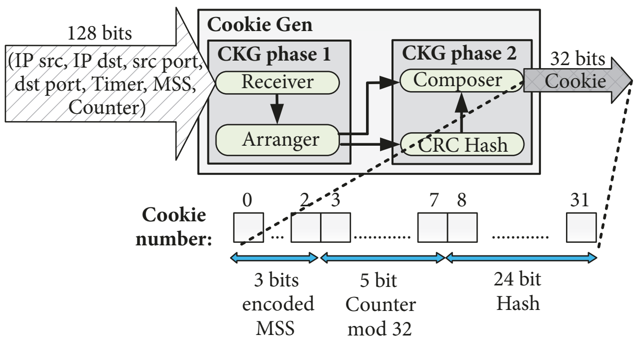


Figure 5: Cookie generation process using CRC hash algorithm and the construction of Cookie number.

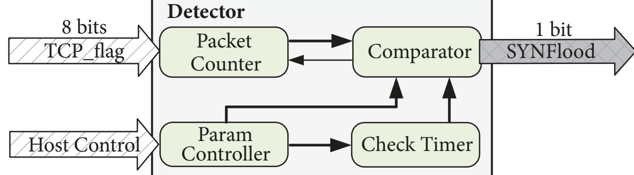


Figure 6: The construction of SYN flood Detector module.

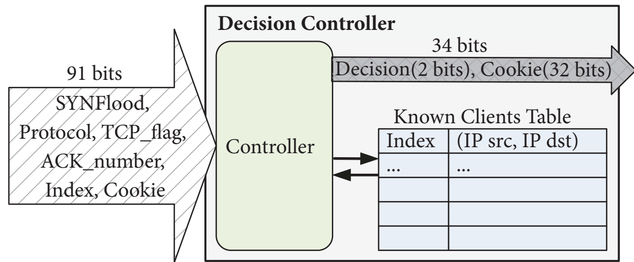


Figure 7: The Decision Controller module with authentication mechanism using Known Clients Table.

processed by two separated modules (`IDG phase 1` and `IDG phase 2`) to issue an index number for a particular client based on IP source and destination fields. Figure 4 illustrates detail behaviors of the process. According to the figure, a CRC hash algorithm is used to issue a 10-bit index number for a client so that the core can identify the client. With the 10-bit index, the core can manage up to 1024 different clients that want to connect to the protected server. In other words, the list of known clients can be 1024 items at maximum. However, after verified according to the 3-way TCP Protocol, the client will be removed from the known clients list.

4.1.3. `CKG Phase 1` *and* `CKG Phase 2`. Similarly, to the Index Generation process, the Cookie generation process also takes two stages (two clock cycles), done by both the `CKG phase 1` and `CKG phase 2` module, to issue 32-bit Cookie numbers so that the core can verify clients. Figure 5 represents detail of the two modules and the way 32-bit Cookie numbers generated. Based on 32-bit counter, 64-bit IP source and destination, and 32-bit TCP/UDP source and destination ports fields extracted by the previous stages, the modules produce 24-bit CRC hash numbers and construct 32-bit cookies. These Cookie numbers are sent back to unknown clients that would make connections with the server.

4.1.4. `Detector`. This module mainly implements the detection algorithm presented in Algorithm 1 to recognize SYN flood attacks. Figure 6 introduces hardware structure of the `Detector` module. The module receives an 8-bit flag that describes the being scanned Packet. Based on this flag, the module counts the number of SYN and ACK packets, done by the `Packet Counter` submodule. Along with this counting behavior, a timer is set according to the *INTERVAl* value determined by administrators. The submodule `Param Controller` keeps all parameters *INTERVAL* and thresholds (*SYN_PKT* and *ACK_DIFF*). The `Check Timer` submodule is responsible for triggering the `Comparator` module every *INTERVAL* period to making comparisons between counted values from `Packet Counter` and predefined thresholds. Based on the results of comparisons, the `SYNFlood` signal is asserted or deasserted to announce the `Decision Controller` module about SYN flood attacks in the system.

4.1.5. `Decision Controller`. This module is mainly responsible for executing the decision algorithm proposed in Algorithm 2. Detailed structure of the module is illustrated in Figure 7. The module consists of 2 submodules, named `Controller` and `Known Clients Table`. According to the algorithm, the module receives an index number (10 bits) and a Cookie number (32 bits) for the being scanned Packet, the Protocol flag (8 bits), and TCP flag (8 bits) that are extracted from Header fields of the Packet, ACK number (32 bits) of the ACK Packet sent from the client, and the `SYNFlood` signal from the `Detector` module.

The `Controller` submodule checks the `SYNFlood` signal and other information to issue corresponding decisions according to Algorithm 2. When the `SYNFlood` signal is false, i.e., SYN flood attacks do not exist, all arriving packets are forwarded to the server to reduce latency. When `SYNFlood` signal is asserted, SYN packets coming from unknown clients activate the verifying process; i.e., converting SYN to SYN-ACK is issued. SYN packets coming from known clients (existing in the `Known Clients Table`) are forwarded to the protected server and information of these clients is removed from the table as explained in Section 3.1.

4.1.6. `Packet Modifier`. As explained in Section 3.1, this module is responsible for constructing SYN-ACK and RST

Table 1: Header modification in two cases: converting SYN to SYN/ACK packet and converting ACK to RST packet.

Header fields	SYN ⟶ SYN/ACK	ACK ⟶ RST
IP src	IP dst	IP dst
IP dst	IP src	IP src
TCP/UDP src port	TCP/UDP dst port	TCP/UDP dst port
TCP/UDP dst port	TCP/UDP src port	TCP/UDP src port
Flag	SYN/ACK	RST
ACK number	Sequence number + 1	Reset to zero
Sequence number	Cookie number	ACK number
Window size	Keep	Reset to zero
TCP header checksum	Recalculated	Recalculated

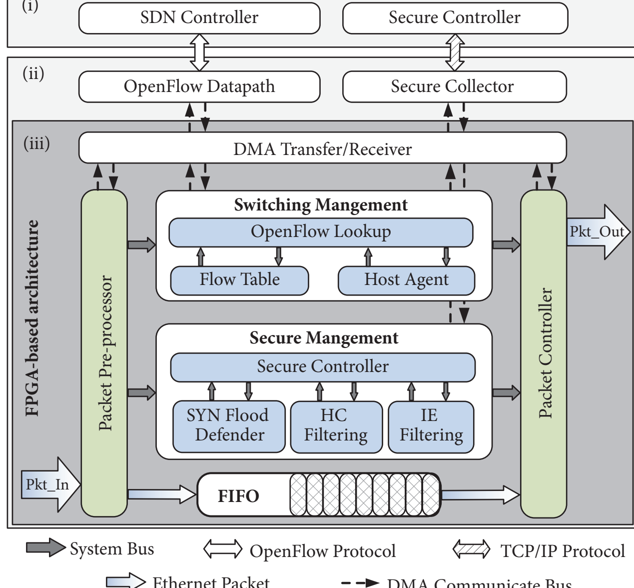


Figure 8: System architecture is organized as follows: (i) host Controller, (ii) OpenFlow data plane, and (iii) FPGA-based forwarding device.

packets to reply to clients that would make connections to the protected server. SYN-ACK packets are issued when the core receives SYN packets from unknown clients. Meanwhile; RST packets are sent to verified clients, after ACK packets from these clients are received. However, it is simple to construct these two types of packets, by modifying some Header fields of arriving packets. Table 1 summarizes the fields that are adjusted to create corresponding packets.

4.2. Combination of SYN Flood Defender and OpenFlow Switch. We introduced our Secured OFS, where a number of DDoS countermeasure mechanisms are integrated into an OpenFlow-based switch, in our previous work [11]. We now integrate the SYN flood defender core to this OpenFlow switch architecture as a security core to conduct more realistic tests. Figure 8 illustrates the high-level structure of our integrated system and can be categorized into three layers according to the OpenFlow architecture, including (i) host Controller, (ii) OpenFlow data plane, and (iii) FPGA-based forwarding device.

The *host Controller* layer consists of two particular controllers. While SDN controller is responsible for manipulating SDN-based forwarding rules, Secure Controller handles, and monitors the security functions of the forwarding plane. Each Controller uses one specific type of Protocol to communicate with the data plane layer: SDN controller uses the traditional OpenFlow Protocol and Secure Controller uses TCP/IP Protocol.

The *OpenFlow data plane* layer consists of services functioning as interfaces for communication between controllers at software level and the FPGA-based forwarding device. The *FPGA-based forwarding device* accommodates our secured OpenFlow switch consolidated by our proposed SYN flood defender. However, to efficiently use hardware resources we make some enhancements as follows.

(1) Both OpenFlow switching behaviors (managed by the Switching Management block) and Secure scanning behaviors (managed by the Secure Management block) require Packet Header fields for their processing. Therefore, instead of implementing each Header extraction process for each block, we develop a `Packet Pre-processor` module for extracting Header fields of incoming packets.

(2) The FIFO memory is used to store raw packets can be removed from SYN flood defender because the OpenFlow switch also has a FIFO memory connected between the Packet Preprocessor and Packet Controller.

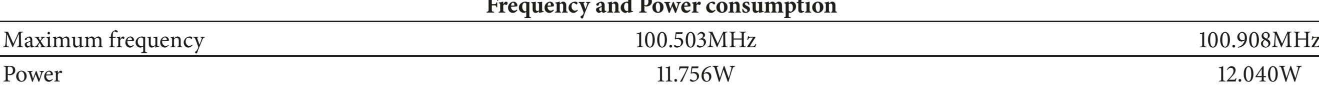
(3) By removing the FIFO memory, the `Packet Modifier` module in SYN flood defender is moved to the Packet Controller block to modify Packet in FIFO memory.

5. Evaluation

In this section, we present our experimental results for the proposed SYN flood defender. The core is evaluated in two different experiments, standalone and integrated into our secured OpenFlow switch. In each experiment, throughput and latency of the core are reported by using results from various testing scenarios.

Table 2: Synthesis results for the proposed SYN flood defender core implemented in the NetFPGA 10G platform.

Hardware resources usage		
Resources	**SYN flood Def.**	**OpenFlow**
Register	4,435 *(2.96%)*	80,080 *(53.47%)*
LUT	4,411 *(2,95%)*	70,825 *(47.29%)*
BlockRAM	18 *(5.56%)*	200 *(61.73%)*
Frequency and Power consumption		
Maximum frequency	100.503MHz	100.908MHz
Power	11.756W	12.040W



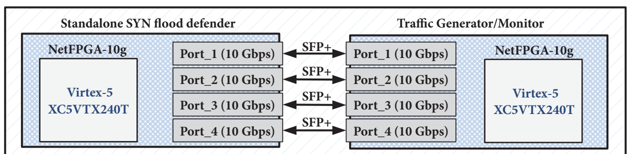


Figure 9: Performance evaluation model for standalone SYN flood defender on NetFPGA-10G boards.

5.1. Synthesis Results. Both the standalone SYN flood defender core and the OFS with our core integrated are developed on the NetFPGA-10G platform containing a Xilinx Virtex-5 xc5vtx240t device. The device hosts 149,760 Registers, 149,760 LUTs, 324 BlockRAMs, and 37,440 Slices. The board provides 4 high-speed half-duplex SFP+ ports with 40 Gbps maximum throughput. Both the core and the OFS are implemented by Verilog-HDL. While the standalone core does not use any IP core for improving flexibility, the OFS with our core integrated uses some IP core provided by Xilinx such as AXI4 lite. Both the standalone core and the integrated OFS are synthesized with Xilinx XPS 14.7 [38] and further optimized by Xilinx PlanAhead [39].

Table 2 shows the hardware resources usage of the proposed core and the OpenFlow switch with SYN flood defender core integrated. As shown in the table, the proposed core is hardware-efficiency compared to the full OFS with integrated secured cores (the core uses only 2.96% Registers, 2.95% LUT, and 5.56% BlockRAM of the FPGA device while the full OFS requires 53.47% Registers, 47.29% LUT, and 61.73% BlockRAM). The table also provides synthesis results that are maximum frequency and power consumption of the core and the full OFS. As shown in the table, both the core and the OFS can almost function at the same maximum frequency.

5.2. Standalone Experimental Results. We develop two comprehensive scenarios to evaluate performance and throughput of the core. In first scenario, we simulate SYN flood attacks by sending all SYN packets to the core at an extreme high rate. The scenarios simulate a realistic situation when a set of network packets ranging from 62 bytes to 1500 bytes in length are sent to the core. To perform the test, two different

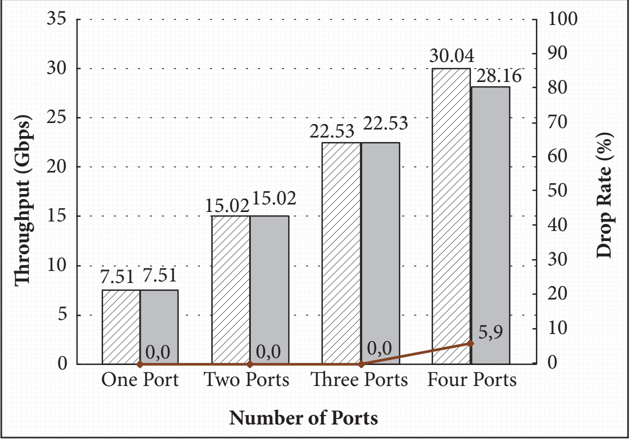


Figure 10: Processing throughputs of the core with 62-byte SYN packets.

NetFPGA-10G boards are used with connection topology shown in Figure 9. One board ported the OSNT (Open Source Network Tester) [40] to generate and send packets to the other board hosting the core.

As shown in Figure 9, each NetFPGA-10G board consists of 4 SFP+ high-speed ports. Therefore, to fully investigate the core, we handle different situations by sending packets to the core through 1 port, 2 ports, 3 ports, and 4 ports. For evaluating the throughputs of the core, the OSNT monitors reply packets generated by the core, also through SFP+ ports. In other words, the board hosting the OSNT tool plays two roles, the protected server and clients.

With the first testing scenario where all packets are 62-byte SYN packets, the OSNT tool is able to generate packets up to 7.51 Gbps per port. Therefore, we examine our core with packets throughput by up to 7.51 Gbps, 15.02 Gbps, 22.53 Gbps, and 30.04 Gbps for 1 port, 2 ports, 3 ports, and 4 ports, respectively. Figure 10 shows the throughput of our core working standalone in this scenario. According to the figure, the core is able to process incoming throughputs by

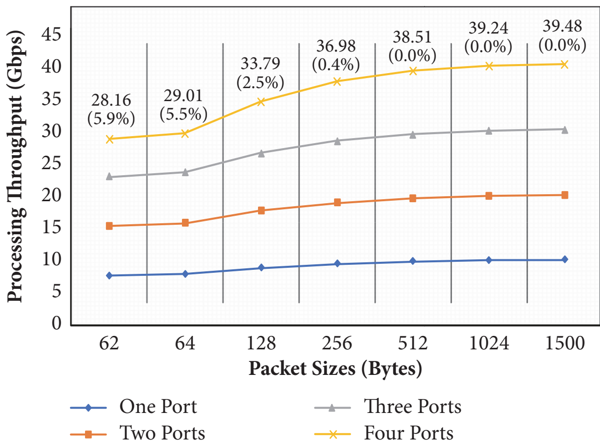


Figure 11: Processing throughputs (Gbps) with various Packet sizes and drop rates (%).

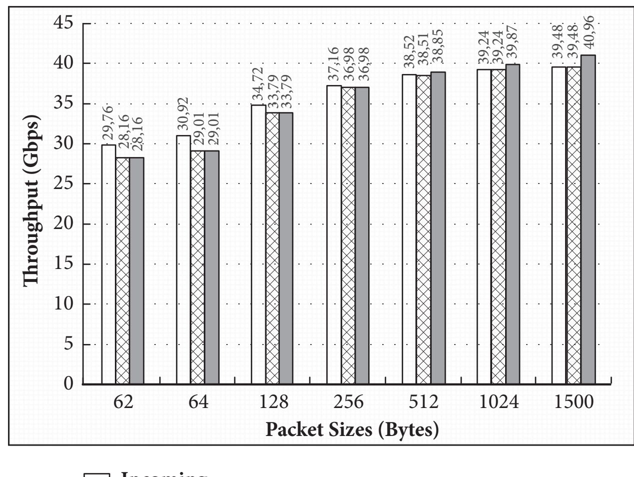


Figure 12: Comparison of throughputs measured by experiments and calculated by our estimated model.

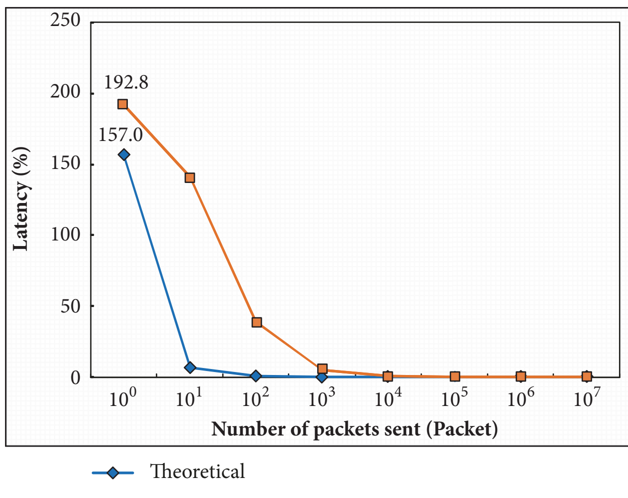


Figure 13: The latency comparison for different numbers of packets.

up to 28.16 Gbps. In other words, packets dropping happens when the core is attacked by SYN flood at throughput 28.16 Gbps. When packets are dropped, safe clients cannot connect to the protected server because requests from these clients are dropped.

In the second testing scenario, we measure throughputs of the core with packets at various sizes ranging from 62 to 1,500 bytes. The testing method is the same with the first scenario. Figure 11 shows throughput results of our core in this scenario. In this figure, the horizontal axis represents sizes of packets while the vertical axis is processing throughputs of the core corresponding to sizes of packets. Similarly to the first testing scenario, we request the OSNT tool to send packets to the core at maximum throughput capacity. When only one port is used, 62-byte packets and 1,500-byte packets can be processed at maximum throughputs by up to 7.51 Gbps and 9.87 Gbps, respectively. Both are the maximum throughputs that the OSNT tool can issue per port. When all SFP+ ports are used, maximum throughputs for packets at various sizes from 512 bytes to 1,500 bytes are four times compared to one port. However, drops occur for 62-byte, 64-byte, 128-byte, and 256-byte packets at rates 5.9%, 5.5%, 2.5%, and 0.4%, respectively, due to a large number of interpacket gaps inserted.

Based on results collected from the aforementioned testing scenarios, we verify our performance estimation model (theoretical throughput) presented in Section 3.2. Figure 12 compares incoming and processing throughputs (the first and the second columns, respectively) measured by experiments with theoretical throughputs calculated by (8) (the last columns) with various packet sizes. Incoming throughput is the maximum throughput that the OSNT tool is able to be sent to our core. Processing throughput represents how many bit the core can process when packets arrive at the incoming throughput rate. When processing throughput is smaller than incoming throughput, a number of packets are dropped due to the limitation of the core. As discussed above, the AXI4 lite interface is used for interconnect for the proposed core and the SFP+ ports of the platforms. Therefore, the frame size W in the equation is 32 bytes while the interpacket gap H is 4 bytes for the Ethernet II Protocol. The results show that performance evaluation model produces almost the same values with experiments. Processing throughputs measured with packets smaller than 512 bytes in size match well with theoretical results. For 512-byte, 1,024-byte, and 1,500-byte packets, we overestimate the core due to the limitation of the OSNT tool. In other words, the maximum generating capacity of the OSNT tool is limited at 38.51 Gbps, 39.24 Gbps, and 39.48 Gbps for 512-byte, 1,024-byte, and 1,500-byte packets, respectively.

Finally, we compare latency introduced by the core in our experiments with results estimated by (3). According to simulation results, our core takes 28 clock cycles to process the first frame ($D = 28$). Figure 13 illustrates the latency when different numbers of packets are sent to our core. The vertical

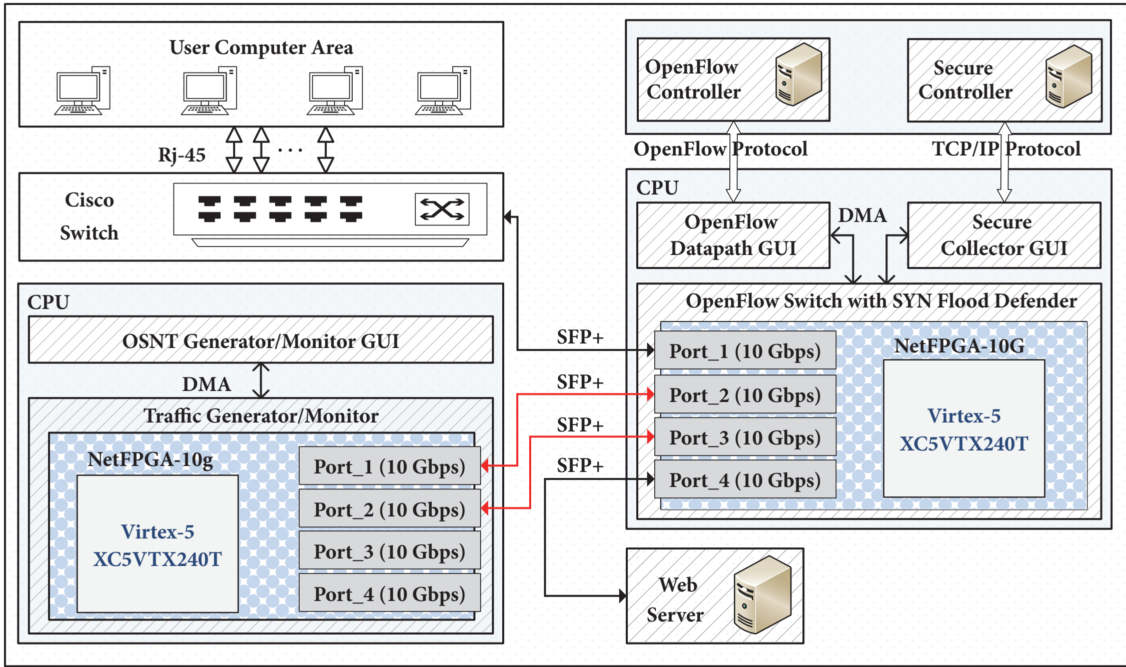


Figure 14: Testing model for verifying an integration of the proposed core and our OFS.

axis shows latency in percentage compared to a system without our core while the number of packets sent to the core is represented in the horizontal axis. According to the figure, our latency estimation model is matched well to experimental results when a large number of packets arrive at our core. Thanks to the pipeline architecture, the latency is decreased exponentially when the number of packets increased (0.07% with 10^5 packets in practical).

5.3. Integrating with Secured OFS Experimental Results. For a comprehensive validation, our SYN flood defender core is integrated into our OFS switch to protect a Web Server deployed in a local area network. In a real network, the parameters presented in Algorithm 1 need to be considered for an efficient SYN flood detection process. Setting the three parameters *ACK_DIFF*, *INTERVAL*, and *SYN_PKT* should be suitable for real-time network traffic. Researchers in [41] concluded that almost half the attacks had an estimated rate of 500+ packets per second. Therefore, we set *SYN_PKT* to 500 seconds and *INTERVAL* to 1 second. The third parameter, *ACK_DIFF*, is affected by transfer times and round-trip times [42] of packets. Since round-trip times are small in our testing local network, we set this parameter to be smaller than 100 (a larger *ACK_DIFF* can introduce high false negative rates). Figure 14 shows our testing model including various components as follows.

(i) An *OpenFlow Controller*, deployed on a traditional CPU, handles the OpenFlow switch using *OpenFlow Protocol*.

(ii) A *Secure Controller*, also deployed on a traditional CPU, handles the Secure functions using *TCP/IP Protocol*.

(iii) Our OpenFlow switch with an integrated SYN flood defender core implemented on a NetFPGA-10G board. The switch is connected to the host CPU Controller through the PCIe interface (DMA).

(iv) A *Web Server* provides simple web services for users on the local network.

(v) A group of computers represents users to retrieve services from the Web Server.

(vi) A *Cisco Switch* to connect user computers and also provide a bridge between the RJ-45 interface and the SFP+ standard interface.

(vii) An *OSNT Generator/Monitor* on a NetFPGA-10G board is integrated to a CPU to generate high-speed SYN flood attacks.

In this testing scenario, the NetFPGA-10G board with the OSNT tool deployed plays a role as attackers that continuously send 62-byte SYN packets at maximum speed to the protected server via two ports (at total 15.02 Gbps that approximate to more than 28,000,000 packets per second). While being attacked, the protected server also receives and responses requests from users. Our experimental results show that, during these peak attacks, the protected server is still able to serve users while the server without protection of our core cannot respond to users under the same attacking rate.

To compare with related works on SYN flood attacks prevention in the literature, we measure the Response time of the server by observing packets arrival time using the Wireshark toolchain [43]. Table 3 compares well-known proposals and our SYN flood defender core in terms of *attack rates* (packets per second), *Response time* (*ms*), and *Latency* (%, when compared to unprotected servers). In each

Table 3: Comparison between SYN flood defender ability and other proposals in the literature.

System	Attack rate (pps)	Response time (ms)	Latency (ms)	Latency (%)
SLICOTS [13]	0	240	110	84.62
	350	1900	1770	1361.54
Avant Guard [14]	0	399	7.3	1.86
	800	400.1	8.4	2.14
NASSD [15]	0	N/A	0.033	N/A
	200,000	N/A	0.034	N/A
Our core	0	1.047	0.112	11.98
	28,410,000	1.183	0.248	26.52

protection approach, the first row with 0-pps attacks shows Response time for SYN packets without attacks while the second row represents Response time when attacks occur.

According to the table, SLICOTS can protect a server against attacks only at 350 pps while introducing latency around 1700-1900 ms, thus giving the latency value up to 1361.54%. Avant-Guard can respond to safe client requests during SYN flood attacks rate at up to 800 pps with only 7.3 ms latency issued. NASSD is able to filter SYN flood attacks by up to 200,000 pps with only 0.034 ms latency. However, the authors do not show Response time in their system. Compared to these approaches, we introduce smaller latency than SLICOTS and Avant-Guard while NASSD outperforms us a bit in terms of latency. Especially, our SYN flood defender core built on the NetFPGA-10G platform outperforms all these SYN flood defender approaches in terms of attack rates when we are able to protect our server against SYN flood attacks by up to 28,410,000 pps, extremely higher than other aforementioned approaches.

6. Conclusion and Future Work

In this paper, we proposed an efficient hardware architecture for high-throughput and low-latency SYN flood defender. The architecture was carefully designed with a pipeline model to improve system performance and reduce latency. We also provided a mathematic model for evaluating performance and maximum bandwidth of the proposed SYN flood defender. The architecture was then implemented by using Verilog-HDL. For validating the core, we conducted multiple experiments with standalone core as well as integration with our Secure OpenFlow switch, proposed in our previous work. For standalone testing, two scenarios already performed for estimating maximum SYN flood attack preventing ability. According to our experimental results, we are able to protect a server against SYN flood attacks up to 28.16 Gbps. We then integrated the proposed core into our OFS for testing in a more realistic model. The OFS now functions not only as a switch for routing packets according to the OpenFlow Protocol but also as a defender core to prevent SYN flood attack. According to this testing, the switch with our SYN flood defender core integrated outperforms most well-known SYN flood defender proposals in the literature. We are able to protect our server against attacks at up to 28,410,000 pps with only small latency in Response time.

Data Availability

Data can be requested by sending an email to the corresponding authors.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research is funded by Vietnam National University, Ho Chi Minh City (VNU-HCM), under Grant no. B2016-20-02.

References

[1] Symantec, “Executive Summary - 2018 Internet Security Threat Report,” 2018, https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-executive-summary-en.pdf.

[2] Cisco, “Cisco Cybersecurity Reports,” 2018, https://www.cisco.com/c/en/us/products/security/security-reports.html.

[3] CISCO, “Defenses Against TCP SYN Flooding Attacks - The Internet Protocol Journal - Volume 9, Number 4,” https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-34/syn-flooding-attacks.html.

[4] Opennetworking, “Software-Defined Networking (SDN) Definition,” 2018, https://www.opennetworking.org/sdn-definition/.

[5] Advisor, “Advantages of Software Defined Networking,” 2018, http://www.ingrammicroadvisor.com/data-center/7-advantages-of-software-defined-networking.

[6] 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.

[7] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, “A security enforcement kernel for OpenFlow networks,” in *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks, HotSDN 2012*, pp. 121–126, Association for Computing Machinery, Helsinki, Finland, August 2012.

[8] R. Braga, E. Mota, and A. Passito, “Lightweight DDoS flooding attack detection using NOX/OpenFlow,” in *Proceedings of the*

35th Annual IEEE Conference on Local Computer Networks (LCN '10), pp. 408–415, Denver, Colo, USA, October 2010.

[9] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, "A Survey on the Security of Stateful SDN Data Planes," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1701–1725, 2017.

[10] C. Pham-Quoc, B. Nguyen, and T. N. Thinh, "FPGA-based Multicore Architecture for Integrating Multiple DDoS Defense Mechanisms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 14–19, 2017.

[11] B. Ho, C. Pham-Quoc, T. N. Thinh, and N. Thoai, "A Secured OpenFlow-Based Switch Architecture," in *Proceedings of the 2016 International Conference on Advanced Computing and Applications (ACOMP)*, pp. 83–89, Can Tho City, Vietnam, November 2016.

[12] NetFPGA, "NetFPGA-10G Information," 2018, https://netfpga.org/10G_specs.html.

[13] R. Mohammadi, R. Javidan, and M. Conti, "SLICOTS: An SDN-based lightweight countermeasure for TCP SYN flooding attacks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 2, pp. 487–497, 2017.

[14] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 413–424, 2013.

[15] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with SDN data plane," in *Proceedings of the IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9, Atlanta, GA, USA, May 2017.

[16] D. J. Bernstein, "Syn cookies, 1996," 2016, http://cr.yp.to/syncookies.html.

[17] S. Ghanti and G. Naik, Defense techniques of SYN flood attack characterization and comparisons,.

[18] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni, "Analysis of a denial of service attack on TCP," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 208–223, May 1997.

[19] J. Lemon, "Resisting SYN Flood DoS Attacks with a SYN Cache," in *BSDCon*, vol. 2002, pp. 89–97, 2002.

[20] M. Yasir, "Introduction to FPGA Technology," 2018, https://www.fpgarelated.com/showarticle/17.php.

[21] Techopedia, "Application-Specific Integrated Circuit (ASIC)," 2018, https://www.techopedia.com/definition/2357/application-specific-integrated-circuit-asic.

[22] H. Wang, C. Jin, and K. G. Shin, "Defense against spoofed IP traffic using hop-count filtering," *IEEE/ACM Transactions on Networking*, vol. 15, no. 1, pp. 40–53, 2007.

[23] B. Xiao, W. Chen, and Y. He, "An autonomous defense against SYN flooding attacks: Detect and throttle attacks at the victim side independently," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 456–470, 2008.

[24] W.-C. Feng, E. Kaiser, and A. Luu, "The design and implementation of network puzzles," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, pp. 2372–2382, Miami, Fla, USA, March 2005.

[25] L. Kavisankar and C. Chellappan, "A mitigation model for TCP SYN flooding with IP spoofing," in *Proceedings of the International Conference on Recent Trends in Information Technology, ICRTIT 2011*, pp. 251–256, India, June 2011.

[26] J. J. Echevarria, P. Garaizar, and J. Legarda, "An experimental study on the applicability of SYN cookies to networked constrained devices," *Software: Practice and Experience*, vol. 48, no. 3, pp. 740–749, 2018.

[27] D. Senie and P. Ferguson, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing," Network RFC2267, 1998.

[28] V. Gulisano, M. Callau-Zori, Z. Fu, R. Jiménez-Peris, M. Papatriantafilou, and M. Patiño-Martínez, "STONE: A streaming DDoS defense framework," *Expert Systems with Applications*, vol. 42, no. 24, pp. 9620–9633, 2015.

[29] . BoonPing Lim and M. Uddin, "Statistical-Based SYN-Flooding Detection Using Programmable Network Processor," in *Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05)*, pp. 465–470, Sydney, Australia.

[30] J. Mirkovic and P. Reiher, "D-WARD: a source-end defense against flooding denial-of-service attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 3, pp. 216–232, 2005.

[31] W. Chen and D. Y. Yeung, "Throttling spoofed SYN flooding traffic at the source," *Telecommunication Systems*, vol. 33, no. 1–3, pp. 47–65, 2006.

[32] C. Sun, J. Fan, L. Shi, and B. Liu, "A novel router-based scheme to mitigate SYN flooding DDoS attacks," *IEEE INFOCOM (Student Poster)*, 2007.

[33] S. Haddock, "InterPacket Gap and Start of Packet Lane Alignment," 2018, http://www.ieee802.org/3/ae/public/jul00/haddock_1_0700.pdf.

[34] Xilinx, "AXI4-Lite IP Interface (IPIF)," 2018, https://www.xilinx.com/products/intellectual-property/axi_lite_ipif.html.

[35] D. Fifield and M. Baxter, "TCP/IP Reference," 2018, https://nmap.org/book/tcpip-ref.html.

[36] dcs.gla.ac.uk, "EthernetFrame 802.3 AT 10Mbps," 2018, http://www.dcs.gla.ac.uk/lewis/networkpages/m04s03EthernetFrame.htm.

[37] Github, "Standard IP Interfaces," 2018, https://github.com/NetFPGA/NetFPGA-public/wiki/Standard-IP-Interfaces.

[38] Xilinx, "Xilinx Platform Studio (XPS)," 2018, https://www.xilinx.com/products/design-tools/xps.html.

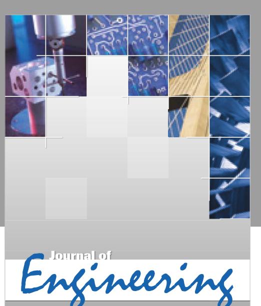
[39] PlanAhead, "PlanAhead Design and Analysis Tool," 2018, https://www.xilinx.com/products/design-tools/planahead html.

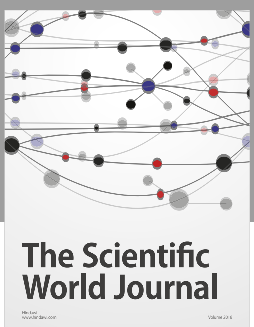
[40] Github, "OSNT 10G Home," https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-10G-Home.

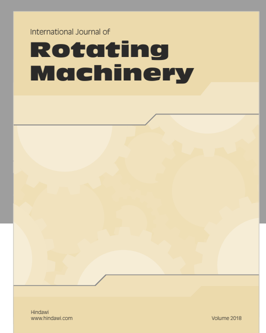
[41] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, "Inferring internet denial-of-service activity," *ACM Transactions on Computer Systems*, vol. 24, no. 2, pp. 115–139, 2006.

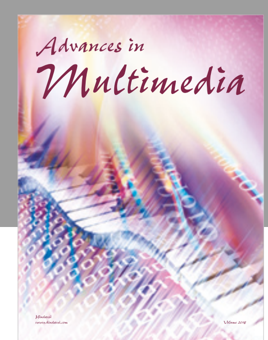
[42] A. Günther and C. Hoene, "Measuring round trip times to determine the distance between wlan nodes," in *Proceedings of the International conference on research in networking*, vol. 3462 of *Lecture Notes in Computer Science*, pp. 768–779, Springer, Berlin, Germany.

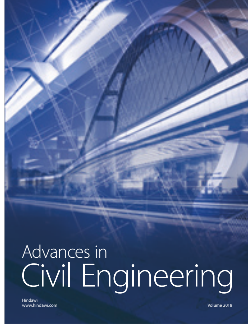
[43] Wireshark, "Wiresharke Go Deep," 2018, https://www.wireshark.org/.











Submit your manuscripts at
www.hindawi.com