Scalable Network Function Virtualization for Heterogeneous Middleboxes

Xuzhi Zhang, Xiaozhe Shao, George Provelengios, Naveen Kumar Dumpala, Lixin Gao, and Russell Tessier
University of Massachusetts, Department of Electrical and Computer Engineering, Amherst, MA 01003

***Abstract*—Over the past decade, a wide-ranging collection of network functions in middleboxes has been used to accommodate the needs of network users. Although the use of general-purpose processors has been shown to be feasible for this purpose, the serial nature of microprocessors limits network functional virtualization (NFV) performance. In this paper, we describe a new heterogeneous hardware-software approach to NFV construction that provides scalability and programmability, while supporting significant hardware-level parallelism and reconfiguration. Our computing platform uses both field-programmable gate arrays (FPGA) and microprocessors to implement numerous NFV operations that can be dynamically customized to specific network flow needs. As the number of required functions and their characteristics change, the hardware in the FPGA is automatically reconfigured to support the updated requirements. Traffic management and hardware reconfiguration functions are performed by a global *coordinator* which allows for the rapid sharing of middlebox state and continuous evaluation of network function needs. To evaluate our approach, a series of software tools and NFV modules have been implemented. Our system is shown to be scalable for collections of network functions exceeding one million shared states.**

I. Introduction

As the Internet has evolved, increasingly diverse network functions, or middleboxes, have been deployed to accommodate business and social needs. Typical network functions, such as firewalls, network address translations (NATs), load balancers, packet classification, and proxy caches, process packets in sophisticated ways, so as to ensure reliability and improve performance in enterprise, service provider, and cloud provider networks. Recently, operators have expressed interest in replacing dedicated ASIC-based appliances with software-based network functions running on generic commodity hardware—a trend known as network function virtualization (NFV). These generic commodity hardware components are typically virtualized into multiple network function instances, each of which supports different network functions. NFV enables operators to enforce high-level policies expressed by enterprise or service networks by directing flows through appropriate network function instances, and further enables isolation among high-level policies performed for different customers.

The customization of existing classification and management blocks to support network functions is challenging. In general, the serial nature of microprocessors limits the achievable performance of NFV implementations while ASICs limit real-time configurability. To achieve the parallelism and flexible classification and management performance required, CoNFV, a network function platform based on FPGAs, microprocessors, and supporting software running on commodity hardware, has been developed. This distributed and scalable network function virtualization platform allows for the sharing of state across middleboxes and the rebalancing of NFV functions using FPGA reconfiguration and microprocessor virtual machine thread creation, as needed. A library of programmable modules has been constructed based on specialized SQL attack detection, distributed denial-of-service (DDoS) detection, flow classification, and network address translation (NAT). These function modules, implemented in either FPGA hardware or processor software, are swapped into middleboxes in response to customer needs and network traffic. To support system operation, a real-time NFV state sharing and resource allocation tool has been implemented. The tool periodically identifies required classification and management functions, assembles the components from specified libraries, and dynamically reconfigures the component FPGA(s) that implement(s) the network function. Our prototype network function virtualization environment is assessed using Altera DE5 FPGA boards, microprocessor-based middleboxes, and network switches. The system is shown to be scalable both in middlebox count and quantity of shared state.

Section II presents NFV and the use of FPGAs in networking functions. In Section III, we present our scalable hardware and software system. Implementation details are provided in Section IV and our experimental methodology is detailed in Section V. Section VI quantifies the benefits of our dynamic reconfiguration approach. Section VII concludes the paper and offers directions for future work.

II. Related Work

A. Network Function Virtualization

NFV is a concept that virtualizes an entire class of network node (or middleware) functions into building blocks that may be connected, or chained, together to create communication services. These network nodes include border controllers (such as firewalls, load balancers, and wide-area network (WAN) accelerators) that protect a network. Traditionally, a network border controller consists of a collection of custom hardware appliances, each of which is designed for a specific network function. With the advance of server virtualization technology, it is possible to decompose traditional network border controller functions into virtual machines running different software. When designing and developing the software that provides virtual network functions, it is possible to break software into components and package those components into one or more functions. To provide isolation among network functions customized for each customer, it is important to install each software component into a virtual machine. Virtual machines are hosted in one or more physical nodes consisting of commodity hardware. They are connected by tunnels to satisfy the requirements of a customer.

Recent work on network function virtualization has mainly focused on the control and management of middlebox functions. Qazi et al. [1] employed Software Defined Networking (SDN) principles to enforce policies for traffic steering. Sherry et al. [2] proposed to use cloud services to perform network functions. Gember et al. [3] aims to provide mechanisms for tenants to specify their middlebox needs, and automatically deploy and scale middleboxes that maximize performance. A number of studies [4], [5] have focused on designing software-based programmable middleboxes in a virtualized environment.

B. FPGA-based Networking Functions

Reconfigurable logic provides an ideal platform for network functions due to the parallelism, specialization, and adaptability offered by FPGA devices [6]. These characteristics match well with the multi Gigabit-per-second (Gbps) throughput constraints frequently imposed on networking infrastructure and the need for frequent updates required by changing packet analysis and filtering metrics. As FPGAs continue to be integrated into cloud computing environments [7] and data centers [8], their use in network and application processing will continue to grow.

A number of FPGA-based platforms have been deployed for network applications involving performance improvement, load balancing, and reliability. A packet classifier [9] was used in a decision-tree-based, 2-D multi-pipeline architecture in a Virtex 5 device to obtain up to 80 Gbps throughput. A wide range of FPGA-based network intrusion detection systems have been implemented using CAMs [10], finite automata [11], and Bloom filters [12]. FPGA logic allows for the implementation of a massive number of parallel matching FSMs and Bloom filter hash functions that can be customized to a changing set of matching rules, including the entire SNORT NIDS ruleset [11]. Hardware-based FSMs for ruleset matching can easily be synthesized from a high-level language, such as C. In general, these network functions operate in isolation on separate boards in a subnetwork.

State and configuration management for subnetwork FPGAs has been limited by a lack of global state coordination support and the inability to swap their functions using network-wide information. Although a recent NFV system using FPGAs [13] is a step in the right direction, the project focuses more on NFV programmability than on the ability to perform on-the-fly reconfiguration and state sharing. Our primary contribution in this work is the development of such a scalable and automated system in a networked environment.

III. System Operation

A. System Overview

It is common for middleboxes positioned across a subnetwork to deploy distributed functions using commodity hardware, custom hardware, virtual machines (VM), or reconfigurable hardware. Information from multiple packet flows must often be utilized for these stateful, distributed functions. Information is collected locally during packet processing from flows that pass through the middlebox. For a variety of applications, such as NAT and SQL injection (SQLi) attack detection, a distributed approach allows for parallel analysis

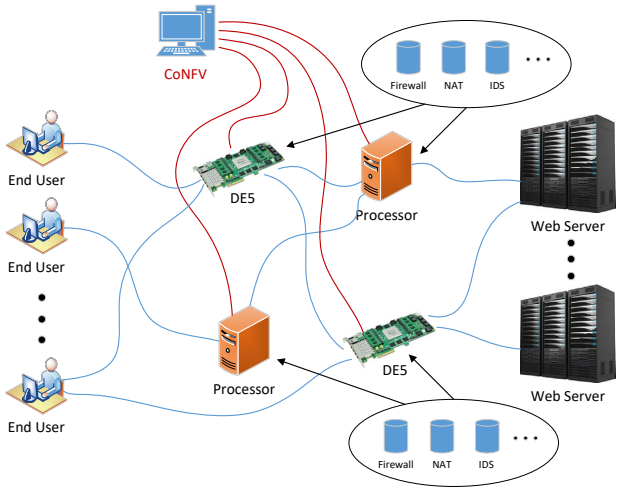

Fig. 1: Overview of the CoNFV configurable network function virtualization system using processor- and FPGA (DE5)-based middleboxes

of multiple flows, each collecting correlated information. The scalable CoNFV system collects global state information and shares this information among distributed FPGA and microprocessor packet processors. The CoNFV *coordinator* gives each middlebox access to global state information using programmable interfaces. Subsets of this information are cached in the middleboxes for some applications.

Middlebox and coordinator functionality can be quickly updated as network function needs change. For example, many NFV operations can initially be assigned to software for low and moderate traffic loads. As network traffic and computational workload increase for a function, instances can be migrated to FPGA-based hardware. A traffic and workload decrease for a specific function can have the opposite effect. The allocation of functions to middleboxes is dynamically assessed and orchestrated by the coordinator as state-based network conditions are processed. The coordinator automatically reallocates resources as needed.

An overview of our global state-sharing system for heterogeneous middleboxes is shown in Figure 1. Microprocessor- and FPGA-based (DE5) middleboxes are distributed across the network. The middleboxes share state information through TCP connections to the CoNFV coordinator. As shown in Section VI, the coordinator is able to handle state for a scalable set of middleboxes, with minimal packet processing slowdown. The network setup represents a number of interconnect configurations, including those found in data centers.

Figure 2 shows the framework of the system. The coordinator stores global state values in a table as a set of key-value pairs. Each middlebox can access global state using a key. The *state manager*, a software module which can be configured for each application, can both retrieve and update state. The *resource evaluator* assesses the current utilization of middlebox resources in response to messages and state variables and can choose to perform middlebox resource rebalancing. The *configuration manager* coordinates the resource assignment in collaboration with the resource evaluator.

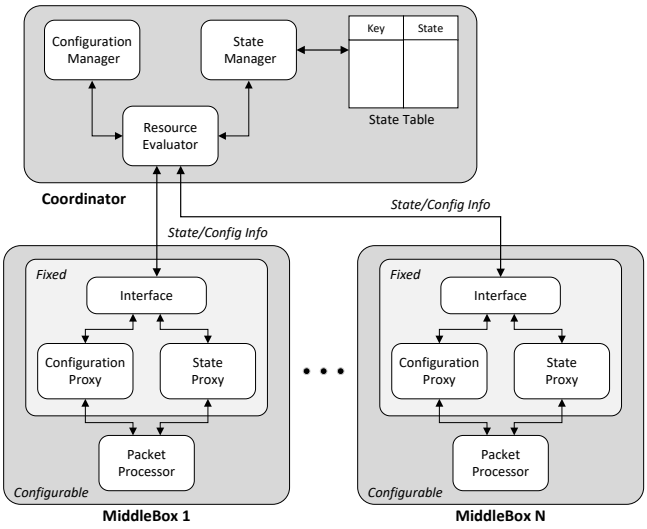

Fig. 2: Middlebox and global coordinator interaction. Middleboxes can be either processor- or FPGA-based.

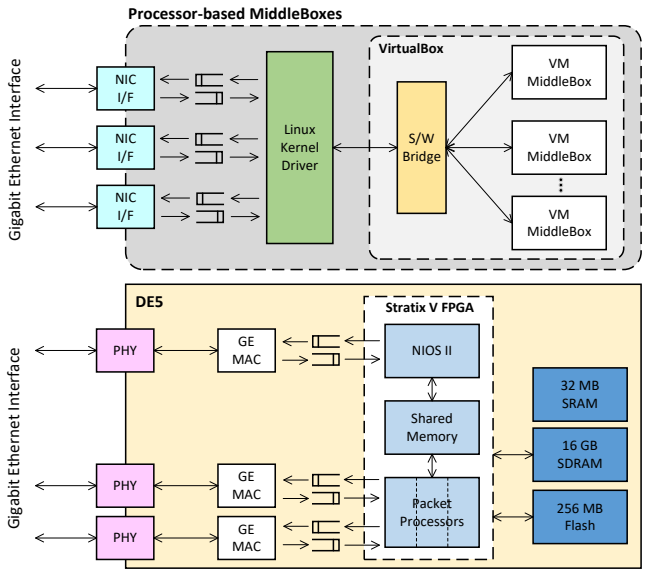

Fig. 3: High-level overview of processor- and FPGA-based middleboxes in CoNFV

Each middlebox contains a *packet processor* and an associated *state proxy* module. After a state request originates in the packet processor, the state proxy module generates and sends state requests to the coordinator, and receives state updates from the coordinator. The *configuration proxy* module coordinates either software thread activation/deactivation for packet processors or hardware reconfiguration for FPGA packet processors. A control *interface* allows for interaction with the coordinator. The specific functions of these modules for three applications is detailed in Section V.

B. Cross-Middlebox State Sharing

Our system relies on state sharing for two types of actions: function triggering and state retrieval. Inspection functions evaluate network traffic and examine packets for monitoring, intrusion detection, and identification of other invasive attacks. Manipulation functions examine and modify flows by dropping, updating or creating new packets. State sharing for these two types of flows proceeds as follows:

Trigger state: For inspection functions, data packets are passively inspected as they enter a middlebox for specific characteristics of attacks such as DDoS or SQLi. If an event is observed that requires a global state update, state information both in the middlebox and in the coordinator are updated. As the state is updated in the centralized state table on the coordinator, it is checked by the resource evaluator to determine if remediation elsewhere in the network is needed. In Section V, we describe how CoNFV can be used to address distributed DDoS and SQLi attacks. A firewall or packet filter can be enabled at one or more points in the network in response.

Retrieval state: For manipulation functions, global states are updated during packet processing. Middleboxes that require retrieved state generally manipulate packets. In the case of state retrieval, individual packet processors request state information if it is not available locally. The coordinator provides a global repository for state information and can update state as needed. A common use of state retrieval is for network address translation (NAT). When NAT receives the first packet of a flow it creates state which determines the translation from an external (IP, port) pair to an internal (IP, port) pair on the local subnetwork. This information must be shared across all middleboxes performing NAT translation for the subnetwork to avoid (IP, port) assignment overlap. In CoNFV, translation information (global state) is stored in the coordinator. If a middlebox receives a packet and its translation information is not stored locally, the information can be obtained from the centralized repository.

C. Dynamic Resource Management

NFV resources must be managed using a global view of function deployment. In response to changing threats or monitoring goals, resources are reallocated under the control of the configuration manager in the coordinator. This unit coordinates the migration, creation, and destruction of functions in real-time to meet functional needs. For processor-based middleboxes, virtual machines (VM) threads are created or destroyed in response to stimuli from the coordinator. For FPGA-based systems, portions of the FPGA circuitry are swapped to change functionality. As shown in Figure 2, FPGA resources are split into fixed resources that manage function interfaces and packet processing resources that can be dynamically reconfigured. For example, in response to the configuration proxy, portions of the FPGAs can be swapped.

IV. Framework Implementation

A. Framework Overview

Our coordinator and middlebox framework includes both commodity processor-based components and FPGA boards (Figure 3). The coordinator is implemented using a processor-based Intel Duo server (2.66 GHz, 4 GB). Processor-based

middleboxes are implemented using a hexad-core Intel Xeon workstation (2.4 GHz, 32 GB SDRAM, and six 1 Gbps NICs). FPGA-based middleboxes are implemented using Altera DE5 boards that include Stratix V FPGAs. TCP sockets are used to enable middlebox/coordinator interactions. The communication between the coordinator and the middleboxes is sufficiently frequent that the coordinator maintains a live connection for each middlebox since it is costly to initialize a new connection for each state operation.

A high-level view of FPGA- and processor-based middleboxes appears in Figure 3. In this configuration, network functions with the highest throughput and lowest latency are assigned to the FPGA on the DE5 board. The DE5 contains 16 GB SDRAM, 256 MB flash, a Stratix V 5SGXEA7N FPGA, and four 1 Gbps Ethernet ports. One port each is used for data input and output and a third port is used for communication with the coordinator via a network switch.

When the number of hardware middleboxes in the subnetwork exceeds available FPGA hardware, additional middleboxes can be generated in software on PCs. A PC server is sliced into virtual machines (VMs) using VirtualBox[1] which virtualizes the server at the operating system level. Each virtual machine operates like a stand-alone server. Software middleboxes are effectively isolated from each other in separate VirtualBox containers that guarantee a fair share of CPU cycles and physical memory to each middlebox. Hardware and software middlebox functions can be customized based on the designer's specifications.

B. State Sharing

The step-by-step behavior of trigger and state retrieval operations is described in the following.

1) Trigger States: A state table of trigger states is located in the coordinator. Middleboxes update trigger states during packet processing. Inside the coordinator, the state manager updates or creates trigger states according to the received state from middleboxes. When a packet comes into a middlebox, the packet processor inspects the packet and sends it out. According to the semantics of the network function, the inspection result might lead to a state update. Whenever the state manager updates or creates a trigger state, a state checker in the *resource evaluator* is triggered to detect malicious activities based on the new state. If a malicious activity is detected, the associated reactions, such as logging or notification, are engaged.

Trigger states do not directly affect the packet processing. They are maintained to detect malicious activities. The semantics of detections are determined by the network function designer.

2) State Retrieval: Asynchronous state operations used in our system allow a packet processor to process other packets without blocking while state is retrieved from the coordinator. However, asynchronous state operations might put packets out of order. For example, if the processing of a packet does not need a state operation, the packet can be processed immediately without waiting for the state return. Network functions that satisfy this condition are not uncommon. For example, for NAT, every packet in a flow requires the same mapping from one (IP, port) pair to another (IP, port) pair. Packets with known translations can proceed while others wait for translation information. During asynchronous state operation, the middlebox is able to process, for instance, the next incoming packet first. When the state is returned from the coordinator, the middlebox continues the processing of the previous packet. Asynchronous state operations buffer packets that require coordinator lookups using a packet buffer table. Packets in the table are indexed by the keys of global states. Then, when the state is returned, the associated packet is retrieved from the table.

[1]https://www.virtualbox.org/wiki/Downloads

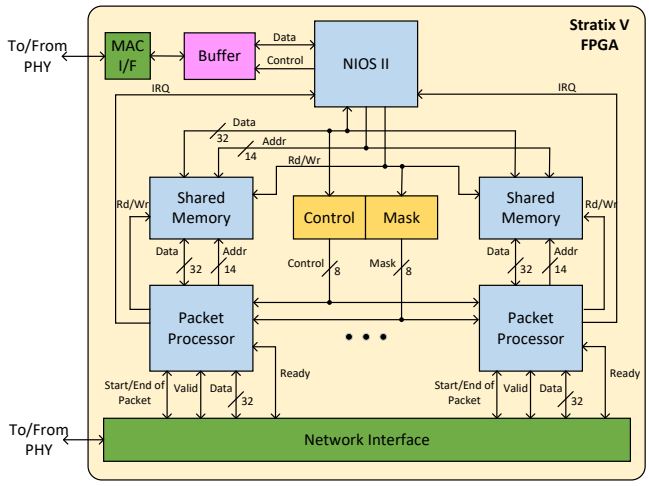

Fig. 4: Detailed FPGA implementation for multiple middlebox packet processors

During packet processing, state retrievals can be much more frequent than state updates. In this case, it is beneficial to cache global states at middleboxes to reduce remote retrieval delay. To cache states, the state proxy in each middlebox maintains a cache table that stores the key-value pairs of states. When the packet processor retrieves a state, the state proxy checks the cache table first. If it misses, the state proxy retrieves the state from the coordinator. When the state returns, it is added into the cache table.

C. DE5 Middlebox and FPGA Module Library

A detailed view of the FPGA platform that can accommodate multiple packet processing middlebox functions is shown in Figure 4. A NIOS II soft microprocessor is used as the interface, state proxy, and configuration proxy. This resource can communicate with the coordinator via a TCP connection implemented on a 1 Gbps link through a switch. The interface between the NIOS II and one or more middlebox packet processors takes place via shared memory and a control register accessed with the Avalon bus. The packet processors implement functions in conjunction with a network interface that includes data queues and port controllers. Incoming data from the PHY are placed in the input queues. Processed packets are sent to the output queues from which they are forwarded to the physical interface.

The implementation shown in Figure 4 illustrates the signal interfaces associated with the middlebox packet processors.

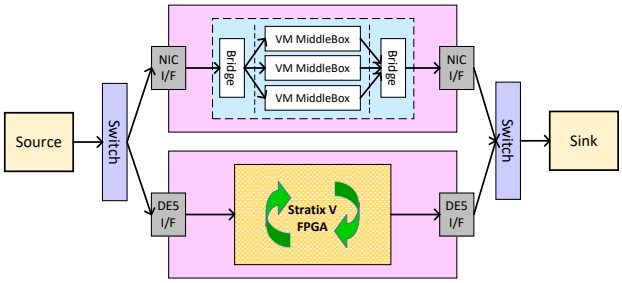

Fig. 5: Multi-receiver setup for scalable NFV including dynamic FPGA reconfiguration

These interfaces include data, address, and control connections to the shared memory and the network interface. These interfaces represent an effective *boundary* for partial FPGA reconfiguration of middlebox functionality. For this project, three middlebox functions for NAT, SQLi, and DDoS have been created with the interface, allowing for interoperability.

D. Dynamic Reconfiguration

The DE5 provides a high-performance platform to implement middleboxes. The choice of an FPGA platform for virtualization does create scalability concerns. Not all middleboxes may contain an FPGA or there may be insufficient resources to implement all needed middlebox functions in FPGAs. As a result, our system allows for the seamless use of both hardware and software middleboxes in the same system with the same coordinator interfaces.

Although minor updates to the hardware middlebox through configuration registers can enable parallelism and provide flexibility, it may not be sufficient for substantial changes in threats which require new hardware modules. As a result, techniques are needed to migrate computation from hardware to software and vice versa. This migration takes place following a sequence of events using FPGA dynamic reconfiguration:

1) **Configuration detection** - The *configuration manager* in the coordinator receives a trigger from the *resource evaluator* to consider middlebox resource allocation. The configuration manager contains state that indicates current resource deployment and required middlebox computation.
2) **Configuration update** - Functions included in the FPGA middlebox targeted for reconfiguration are either terminated or migrated to software on a processor-based middlebox. Traffic previously sent to the FPGA middlebox is retargeted to a processor-based middlebox via a network switch. The *configuration manager* sends messages to the middleboxes to replace their current functions with alternative configurations. In our system, an SDN switch is configured to reroute affected traffic (Figure 5).
3) **Middlebox configuration** - An FPGA-based middlebox loads the appropriate configuration for the new function into the FPGA.
4) **Middlebox response** - When the middlebox reconfiguration is complete, a response is sent to the coordinator. An SDN switch is configured to redirect traffic through the newly-configured FPGA middlebox.

A detailed example using middlebox functionality migration is described in Section VI.

To support FPGA-based middlebox configuration (step 3 above), the FPGA can be either partially or completely reconfigured. Both approaches are supported in our system. Whole-chip FPGA programming on the DE5 is initiated by a trigger signal sent from the FPGA to the MAX II CPLD used for configuration loading. Multiple configurations for the FPGA are available in on-board flash memory. The start address of the configuration image is specified in flash and used by the CPLD to initiate configuration image loading into the FPGA. Before reconfiguration starts, the NIOS II can overwrite this start address so that the next FPGA image can be changed. Once the new FPGA image has been loaded, the TCP connection between the coordinator and the *interface* implemented in the NIOS II is reinitialized.

A more effective approach for middlebox configuration is to swap one of the middlebox *packet processor* modules in Figure 4[2]. Our partial reconfiguration approach requires the definition of a partial reconfiguration boundary that consists of the 99 interface signals on the module. These signals interface to lookup tables in the module which are driven to a known value during reconfiguration. Partial reconfiguration is controlled by the *configuration proxy* software implemented on the NIOS II. During partial reconfiguration, the NIOS retrieves new configuration information from flash and programs it into the FPGA configuration memory via a partial reconfiguration control block instantiated in the device. Once the middlebox has been properly configured, the coordinator is notified and the switch is reprogrammed to forward associated network traffic for processing. Both partial and full reconfiguration have been successfully used in our system.

V. Middlebox Applications

For experimentation, three FPGA-based library modules which meet the requirements of the previous section were created and tested. The following discussion provides an overview of module operation and use.

1) NAT Implementation: As mentioned in Section III-B, the NAT function converts an inside, local subnet (IP, port) pair to a public network (IP, port) pair. In our implementation, all translations are determined at the coordinator and stored in the coordinator's global state memory. Translation information is returned to a requesting middlebox via a *reply state* message following a *state fetch* message. The middlebox packet processor was implemented in FPGA logic while the state proxy was implemented using a NIOS II processor.

The blocks used in the FPGA-based NAT application are shown in Figure 6. The interface signals on the left of the figure match the packet processor interface signals shown in Figure 4. The extraction module extracts the source address, source port, destination address, destination port, and protocol

[2]Our current testing only uses one packet processor per FPGA

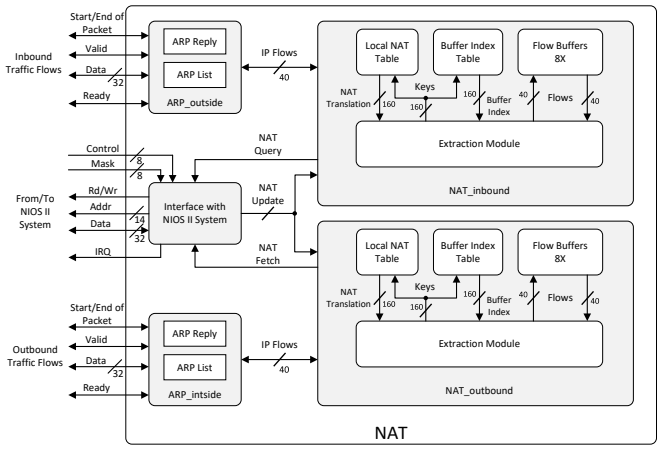

Fig. 6: FPGA middlebox implementation of NAT application

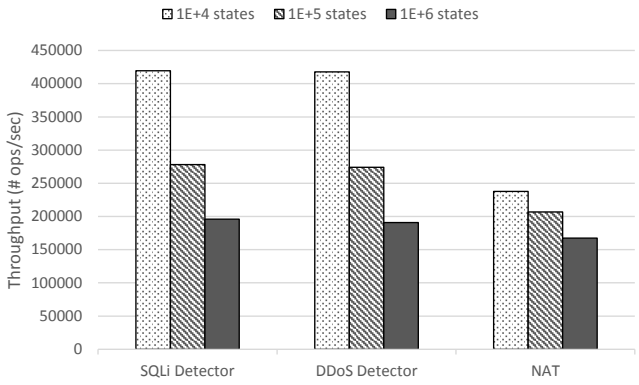

Fig. 7: Results of coordinator stress test. For each test, requests are made to the coordinator at the fastest rate supported by the network interface.

information from the packet header to form a key. The ARP module contains two ARP lists (caches) and reply modules. These blocks allow for the conversion of IP addresses to physical addresses. The NAT module allows other packets to be forwarded while the middlebox waits for the NAT translation to arrive from the coordinator. As a result, packet buffering is needed. In our implementation, eight 16K entry $\times$ 40 bit buffers are used for packet sizes ranging from 64 to 1,500 bytes. A buffer index table, implemented as a hash table, is used to store the index of the buffers for specific flows. For each flow, the key is used as the input to the buffer index table and the local NAT translation table (implemented as a hash table) of depth 4,096 entries. If the translation is not found in the table, a NAT state fetch from the coordinator is initiated by the state proxy. The round trip time to fetch the translation from the coordinator is about 0.2 ms.

Separate translation units are provided in the middlebox for inbound and outbound subnet traffic. The software version of the NAT middlebox implemented on a PC performs the same functions and uses the same message sizes. The state proxy is implemented as a separate VirtualBox module programmed with APIs.

2) SQL Injection Detection: The second function used to test our system was an SQLi detection block. Both FPGA and processor-based implementations of this application are supported. Processor implementations are based on Bro[3]. SQLi detection attempts to identify possible web-based attacks by examining packet payloads for known attack data. The SQLi implementation uses a regular expression matching engine (REME) to find keywords in the GET and POST request lines of an HTTP packet [14]. In the design, a REME can take at most 64 input characters. In our system, TCPreplay[4] is used to send packets ranging in size from 54 to 1514 bytes through SQLi detectors via 1 Gbps ports at varying speeds. A total of 32 regular expression matchers are used.

When a detection occurs, a 41-byte set of information is sent to the coordinator as a message. This information includes the packet source and destination. The coordinator then sends a 51-byte signature to a firewall on another middlebox which is either implemented in an FPGA or a VirtualBox container. The firewall is located between the client and the switch input to the subnets. After activation by the coordinator, the firewall identifies packet headers with offending source and destination addresses and ports and drops them.

3) DDoS Implementation: The final module used to test our system was a distributed denial of service (DDoS) block, based on an earlier design [15], [16]. During a DDoS attack, the attacker floods a victim's network with SYN packets without sending the corresponding ACK packets. Incoming packets which arrive at the middlebox are sampled and a counter (*SYN_ACK_CNT*) is used to keep track of unmatched SYN packets for up to 1,000 destination addresses. The values of the *SYN_ACK_CNT* counters are periodically evaluated to identify deviations from expected values as determined by the mean and standard deviation of the counters. If the values vary beyond a variable threshold for a destination address, a possible DDoS attack is identified. This result triggers a message for the coordinator. The coordinator can identify messages from a number of middleboxes to identify if a pattern exists for a specific destination address. After activation by the coordinator, the software rate limiter identifies packets with offending SYN messages and limits their transmission.

VI. Results

Three separate experiments were performed using our PC and FPGA-board virtualization system. Three Xeon processor-based workstations were sliced into four VirtualBox middleboxes each. An Intel Duo processor-based machine was used as the coordinator. Two Stratix V based DE5 boards were used as FPGA processors. Hardware details of each component were provided in Section IV-A. The results for the coordinator stress, scalability, and reconfiguration tests are described below.

Stress Test: For a distributed system, the state manager in the coordinator may manage millions of global states for a network function. In this first experiment, the state manager was flooded with state requests at the maximum rate of the coordinator network interface to test its processing capabilities. Figure 7 shows the throughput of the state manager portion

[3]http://www.bro.org

[4]http://tcpreplay.synfin.net/

	LUTs	FFs	Block Mem bits
NAT	56,345	59,637	18,202,624
SQLi attack detector	86,127	51,009	1,726,768
DDoS attack detector	16,273	10,467	1,191,936
Firewall	11,328	12,379	1,442,816
NIOS II system	24,634	34,340	3,412,704
Available in FPGA	469,440	938,880	52,428,800

TABLE I: Resource usage for NFV library cores targeted to a Stratix V 5SGXEA7N

	Throughput (Mbps)		Latency (ns)	
	VM	FPGA	VM	FPGA
NAT	522	915	1,009,000	2,000
SQLi	408	898	10,600	336
DDoS	442	908	5,040	135

TABLE II: Latency and throughput comparison of FPGA and VM module implementations.

of the coordinator for the three network functions with the number of global states growing from ten thousand to one million. As the figure shows, the coordinator keeps a high processing speed of more than 100,000 operations per second for the three functions. It was determined that the network interface is the limiting factor in this setup.

Scalability Test: In a second experiment, the ability of the FPGA circuits and virtual machine-based middleboxes to process packets for a scaled set of middleboxes was tested. For the FPGA functions, the resources of the packet processor modules and NIOS II are shown in Table I. The SQLi attack detector requires the most logic resources and defines the region size for partial reconfiguration. All circuits operate at 100 MHz. Table II shows the performance benefits of using the FPGA circuits versus VM implementations. For all three packet processor modules the data throughput of the FPGA implementations matched the input throughput[5]. The dramatically reduced latency numbers for FPGA versus VM (hundreds versus thousands of ns) indicate the benefit of FPGA usage. The FPGA throughput numbers for all three circuits are constrained by the speed of the 1 Gbps network interface. All three circuits support network speeds approaching 10 Gbps.

To evaluate scalability we measured the throughput of our system using an increasingly large set of hardware and software middleboxes and examining overall processing throughput using the SQLi application. Software versions of SQLi are implemented using Bro software. Two workstations sliced into four VirtualBox middleboxes each are used to implement software SQLi. Two DE5 boards implement FPGA versions. All middleboxes are connected to the coordinator via TCP connections. A separate PC is used to generate packets for the subnetwork using TCPreplay and to retrieve packets. ARP protocol is used to steer generated packets through switches to middleboxes. Packets used for testing range in size from 54 to 1514 bytes. Figure 8 shows the scalability of our heterogeneous network system for between 1 and 10 middleboxes for the SQLi application. The first two middleboxes used in the system are FPGA-based, hence the higher slope of throughput on the left side of the graph. As more VM middleboxes are added, system performance versus the ideal case remains close

[5]The TCPreplay tool sourced packets at a slightly reduced rate from 1 Gbps

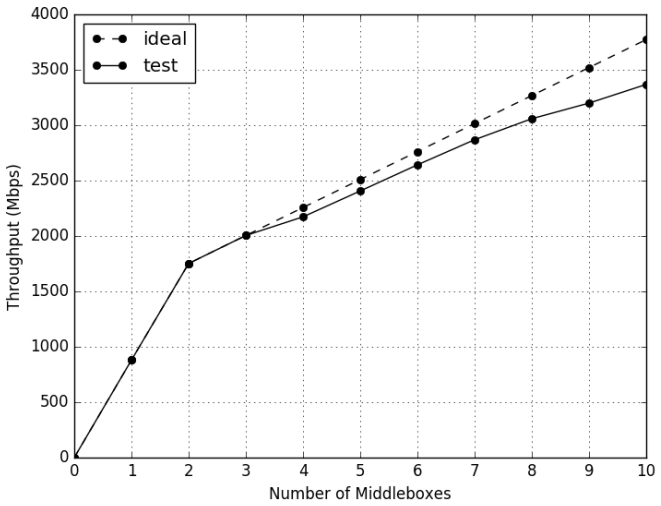

Fig. 8: Scalability of SQLi implemented with up to 2 FPGAs and 8 virtual machines

indicating the capability of the state manager in the coordinator to keep up with simultaneous state requests from both FPGA and VM middleboxes.

Reconfiguration Test: The use of NFV requires the ability to dynamically reconfigure middleboxes in response to changing networking needs. For example, it may be necessary to periodically change middlebox functionality between DDoS and SQLi operations. We performed an experiment with transient variations in the incoming workloads for DDoS and SQLi. Initially, FPGA hardware is used to detect DDoS attacks and software is used to detect SQLi attacks. Although a traffic increase targeted to the SQLi middlebox does not necessarily imply an attack, a microprocessor cannot perform SQLi detection effectively due to throughput limitations. In this case, the microprocessor sends a message to the coordinator indicating the desire for an FPGA middlebox update to support SQLi. The coordinator can decide to swap FPGA NFV functions from DDoS to SQLi attack detection during this period of high SQLi traffic if DDoS processing is limited at the moment.

In a final experiment we determined how quickly a packet processing function can be replaced within an FPGA by the configuration manager in a system with two VM and one FPGA middleboxes. The steps needed to perform the reconfiguration are described in Section IV-D. As seen in Figure 9, initially a DDoS detector is implemented in the FPGA and an SQLi detector is implemented in VM1. When input traffic rate into VM1 consistently exceeds 408 Mbps (the VM throughput limit in Table I), VM1 notifies the configuration manager in the coordinator. Since the DDoS detector throughput is less than 442 Mbps and can be handled in software, its function is migrated to VM2 and the FPGA middlebox is reconfigured to support SQLi detection.

Figures 9 and 10 show the delays associated with the redirection of the SQLi traffic from VM1 to the FPGA and FPGA reconfiguration using both full (FPGA_FR) and partial device (FPGA_PR) configuration. Results in the graphs were generated from experimentation with FPGA and VM

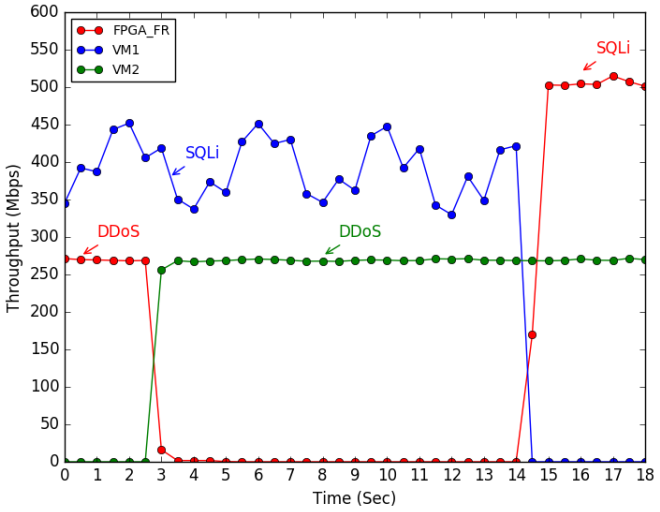

Fig. 9: Performance of system resources during full FPGA reconfiguration.

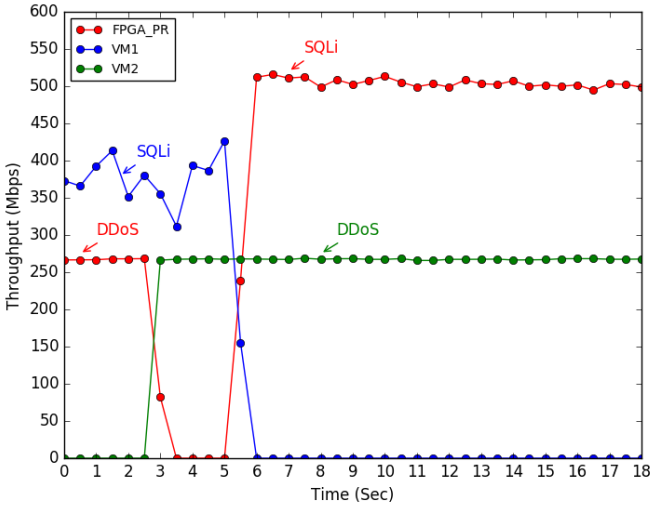

Fig. 10: Performance of system resources during partial FPGA reconfiguration.

middleboxes in the lab. The full FPGA reconfiguration process requires about 12 seconds. This delay includes the time needed to remap traffic using the SDN switch, reconfigure the FPGA, reboot the NIOS II, and reinitiate the connection between the NIOS II and the coordinator. The partial FPGA reconfiguration process requires about 2.5 seconds which primarily consists of partial bitstream loading from flash by the NIOS II. The size of the entire bitstream is 31.3 MB, while the partial bitstreams for both SQLi and DDoS are 15.7 MB. The FPGA reconfiguration time is dramatically reduced for partial versus full reconfiguration since the NIOS II does not need to be resynchronized with the coordinator in the latter case. Since partial reconfiguration is much faster, further advancement of this concept for NFV is desirable.

VII. Conclusions and Future Work

In this paper, a new heterogeneous hardware-software approach to NFV construction is demonstrated that provides scalability and programmability. The platform leverages both FPGAs and microprocessors to support a range of user defined network functions with a common interface. As the number of required functions and their characteristics change, FPGA logic is automatically reconfigured under system-wide control. To evaluate our approach, a series of software tools and NFV modules have been implemented. The scalability and hardware reconfigurability of the hybrid system is demonstrated for known network attacks. Partial FPGA reconfiguration is shown to accelerate the migration of FPGA NFV functions by a factor of 5. In the future we plan to migrate our system to 10 and 100 Gbps networks. Larger and more diverse functions will also be targeted.[6]

References

[1] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM*, 2013, pp. 27–38.

[2] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proc. ACM SIGCOMM Conf. on Appl., Tech., Arch., and Protocols for Comp. Comm.*, 2012, pp. 13–24.

[3] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proc. of the 11th ACM Workshop on Hot Topics in Networks*, 2012, pp. 7–12.

[4] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: enabling innovation in network function control," in *Proc. ACM SIGCOMM*, 2014, pp. 163–174.

[5] J. Martins et al., "ClickOS and the art of network function virtualization," in *Proc. USENIX Conf. on Networked Sys. Design and Impl.*, 2014, pp. 459–473.

[6] D. Unnikrishnan, R. Vadlamani, Y. Liao, J. Crenne, L. Gao, and R. Tessier, "Reconfigurable data planes for scalable network virtualization," *IEEE TComputer*, vol. 62, no. 12, pp. 2476–2488, Dec. 2013.

[7] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. FCCM*, May 2014.

[8] A. Putnam et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. ISCA*, June 2014, pp. 13–24.

[9] W. Jiang and V. Prasanna, "Scalable packet classification on FPGA," *IEEE TVLSI*, vol. 20, no. 9, pp. 1668–1680, Sep. 2012.

[10] M. Gokhale et al., "Granidt: Towards gigabit rate network intrusion detection technology," in *Proc. FPL*, Sep. 2002, pp. 404–413.

[11] B. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. FCCM*, Apr. 2002.

[12] S. Dharmapurikar and J. Lockwood, "Deep packet inspection using parallel Bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[13] B. Li et al., "ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware," in *Proc. ACM SIGCOMM*, July 2016.

[14] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *Proc. ANCS*, Nov. 2008, pp. 30–39.

[15] K. Lu, D. Wu, J. Fan, S. Todorovic, and A. Nucci, "Robust and efficient detection of DDoS attacks for large-scale Internet," *Computer Networks*, vol. 51, no. 18, pp. 5036–5056, Dec. 2007.

[16] H. G. Hosseini and K. Li, "Implementation of transient signal detection algorithms on FPGA," *International Journal of Computer Applications*, vol. 41, no. 12, 2012.

[6]This research was supported by NSF grant CNS-1525836. We thank Intel/Altera for the donation of the DE5 boards.