Application Layer Packet Classifier in Hardware

Racyus D. G. Pacífico and Matheus S. Castanho and
Luiz F. M. Vieira and Marcos A. M. Vieira
Universidade Federal de Minas Gerais
Email: {racyus,mscastanho,lfvieira,mmvieira}@dcc.ufmg.br

Lucas F. S. Duarte and José A. M. Nacif
Universidade Federal de Viçosa
Email: {lucas.f.duarte,jnacif}@ufv.br

***Abstract*—Traffic classification is fundamental to network operators to manage the network better. L7 classification and Deep Packet Inspection (DPI) using regular expressions are vital components to provide application-aware traffic classification. Nevertheless, there are open challenges yet, such as programmability and performance combined with security. In this paper, we introduce eBPFlow, a fast application layer packet classifier in hardware. eBPFlow allows packet classification with DPI on packet headers and payloads in runtime. It enables programming of regular expressions (RegEx) and security protocols using eBPF (extended Berkeley Packet Filter). We built eBPFlow on NetFPGA SUME 40 Gbps and created several application classifiers. The tests were performed in a physical testbed. Our results show that eBPFlow supports packet classification on the application layer with line rate. It only consumes 22 W.**

1. Introduction

Traffic classification (TC) allows network operators to characterize packet flows better, manage resources, and improve network security. It is fundamental to applications such as traffic engineering, network analytics, and Quality of Service (QoS) [1]. However, the traditional TC is a straightforward approach that fails in situations where fields are inconclusive or unavailable. For example, in peer-to-peer (P2P) traffic, the applications do not have default ports, and the HTTP server runs on different ports from port 80 with encrypted connections. L7 classification (Application layer packet classification) rises as a solution to overcome these limitations. In this approach, the classification occurs based on patterns often shared with other applications [2].

Deep Packet Inspection (DPI) is often applied for TC. It examines each byte of the packet's payload at runtime and returns when it finds one or more patterns (e.g., malicious traffic or attacks) previously defined. DPI is a crucial component of classification, and it uses the Regular Expressions (RegExs) power to increase performance to find patterns. However, some limitations and challenges remain open in the context of L7 classification and DPI, such as programmability, performance, efficiency, security, scalability, and usability [3].

To address these problems, we propose eBPFlow, an fast application layer packet classifier on NetFPGA SUME. eBPFlow allows packet classification with DPI in hardware using eBPF (extended Berkeley Packet Filter) [4], a soft processor that enables programming of regular expressions (RegEx) and security protocols using low-level instructions without relying on any pre-established protocol. The Linux kernel already supports executing the eBPF instruction set. Moreover, it combines flexibility and programmability in software with high hardware performance using an FPGA (Field Programmable Gate Array) [5], [6]. It is easy to use and allows DPI applications to operate on packet headers and payload at runtime. We built eBPFlow on the NetFPGA SUME 40 Gbps platform [7]. Our tests were performed in a physical testbed, demonstrating that the eBPFlow supports packet classification on the application layer with line rate. Efficient power usage [8] is also important. eBPFlow only consumes 22 W.

This work's main contributions are: (i) A system allowing users with little hardware expertise to perform high throughput and low latency packet classification with DPI; (ii) Easily programmable, application-layer classification using the well-established eBPF environment; (iii) Multi-core, 40-Gbps NetFPGA SUME [7] eBPF implementation; (iv) Flexible and efficient programming of regular expressions and protocols; and (v) Wildcard lookup mechanism with parallel search of L7 classification rules in all memory using only one operation of constant time $O(1)$. The eBPFlow has two parts: a data plane and userspace. The data plane contains four eBPF engines that share an instruction memory, a timer, and a coprocessor. The userspace has a controller for managing the network and tools to compile/load eBPF programs, handle maps, and compile regular expressions.

The remainder of this paper is organized as follows. In §2, we present an overview of the eBPFlow. In §3, we describe the eBPFlow design. In §4, we present the implementation details of eBPFlow built on top of the NetFPGA SUME platform. In §6, we show the evaluation and results in a realistic environment. In §8, we describe and compare the related work. Finally, in §9, we present the conclusion.

2. Overview

Figure 1 presents an overview of the configuration flow and data processing in eBPFlow. Users can write new protocols in eBPF. Re2c is used to convert regular expressions

978-3-903176-32-4

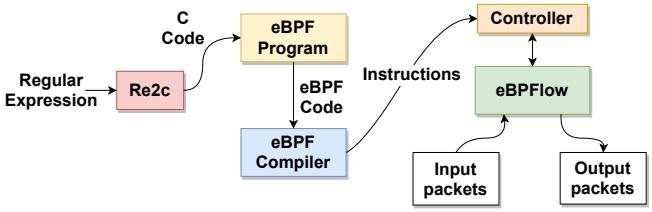

Figure 1. Overview of the configuration flow and data processing.

of any level of complexity to eBPF-compliant C code. However, some adaptations were necessary for the generated code to work in the eBPFlow. The process consists of encapsulating the FSM (Finite State Machine) generated by Re2c within a function with control pointers. It is also necessary for all states to verify that the pointer's current value is greater than the address at the end of the packet to prevent invalid memory access. After this, the code is packaged in a single file and compiled by the eBPF compiler. Then, a controller sends the generated instructions to eBPFlow, which saves them in its instruction memory. From that moment on, the system waits for incoming packets, processes them according to the instructions present in the memory, and performs the appropriate actions according to each packet's content.

2.1. eBPF

eBPF is a general-purpose soft 64-bit processor available on Linux kernel since version 3.15. It allows fast processing of packets at runtime inside the kernel and provides programmability and flexibility on packet computing. Users can compile eBPF programs to bytecode before loading it on the kernel. Languages such as C and P4 support this technology. More details about eBPF are available in [4], a complete course about the subject. High-level languages are used to write code to the data plane and compile it into the eBPF instruction set. Since version 3.7, the LLVM compiler collection has a backend for the eBPF platform, allowing programming in this subset of C and generating executable code in eBPF format.

3. eBPFlow Design

Figure 2 gives an overview of the eBPFlow design and implementation. The system has two components: data plane and userspace tools. The data plane contains four eBPF engines that share an instruction memory, a timer, and a coprocessor. The instruction memory includes a system that changes programs without stopping the processing. The timer allows measurement of network performance (e.g., EWMA, latency, and jitter) when storing the timestamp of packets on metadata. The coprocessor works as eBPF maps in hardware using TCAM/CAM memories to store pairs <key, value>. The userspace is composed of a loader to compile/load programs and handle maps, an eBPF disassembler to convert binary code to instructions, an emulator to debug, and a CLI application to interact with eBPF engines.

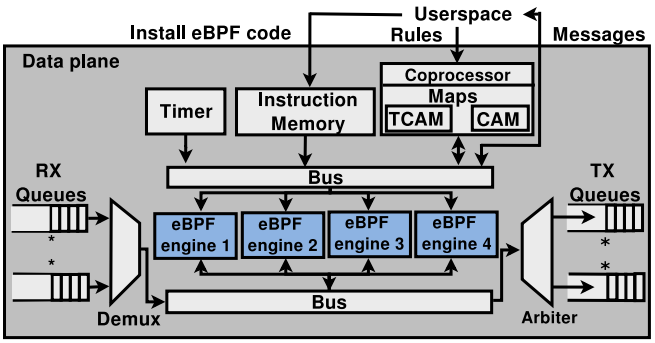

Figure 2. Overview of the eBPFlow design.

eBPF engine: Each eBPF engine comprises three hardware modules: the eBPF processor, a Finite State Machine (FSM), and an action module. The eBPF processor is responsible for performing the parse, matching, and actions using instructions stored in the instruction memory. Also, the processor communicates with the control plane through a socket. The FSM controls the whole operation of packet processing. The action module forwards or discards the packet according to the value stored in R0 after processing the eBPF instructions.

Metadata: The data plane receives the packet through the input interface and stores it in the input queue with additional information called metadata. Table 1 shows the metadata header. The first line indicates the byte order and size. The other lines show the stored structure. After the metadata is received, comes the Ethernet frame. The fields present in the metadata header can be used by the eBPF program as well as in any other protocol field.

TABLE 1. METADATA: INFORMATION RETRIEVED FROM THE PACKET.

0			bit			255
8 bits	8 bits	8 bits	32 bits	32 bits	16 bits	152 bits
Input Port	Source Queue	Destination Queue	Time-stamp (s)	Time-stamp (ns)	Length (bytes)	Free
Ethernet Frame						
Payload						

Actions: The return value of the eBPF processor is stored in register R0 and is used to determine which action the processor should perform on the packet. Table 2 describes the return values of eBPF and their respective actions. eBPFlow enables other dynamic actions such as modifying the packet header, the packet payload, adding or removing fields. Since the data memory stores the packet, a store instruction modifies its content. The packet content can also be used for arithmetic and logical operations, for example, decrementing TTL or recomputing checksum.

4. Implementation

Here, we describe the implementation details of eBPFlow. The data plane was built in Verilog HDL on top of the NetFPGA SUME platform [9]. The user space tools are composed by a controller that defines how the data plane

TABLE 2. ACTION PERFORMED ON THE PACKETS.

Action	Code	Description
Forwarding	0 - 0xFFEF	Forwards packet.
Controller	0xFFF3	Send packet to the controller.
Drop	0xFFF0	Drop packet.
Flood	0xFFFF	Send packet to all ports except for the input port.

forwards the packets, and a software to configure the data plane features according to the user-created C program.

Hardware Instance: FPGA enables the building of hardware logic systems. The NetFPGA SUME hardware has four SFP+ transceivers that support 10 Gbps Ethernet ports. It connects to a motherboard through a PCIe Gen 3 x8 adapter. It contains a Xilinx Virtex-7 690T FPGA [10], which has approximately 693,120 logic cells, a 27 MiB SRAM and 5 ns (200 MHz) clock cycle.

4.1. eBPF Processor with Pipeline

The eBPF processor is responsible for performing the parse, matching, and actions according to the eBPF instructions generated from the C program. When starting the operation of the device, the user must load the eBPF instructions into the instruction memory to define the behavior of the data plane. The Figure 3 presents the data and control path in register transfer level (RTL), containing five data functional units (program counter, instruction memory, register file, arithmetic logic unit (ALU), data memory), and three control units (hazard detection, forwarding, and control).

The system searches for the instruction in the memory and divides it into five parts: operation code, destination and source register address, offset, and immediate value. There is a specific unit that receives each part of the instruction. The control unit receives the operation code and forwards the control signals to the functional units, defining the behavior of each unit. For example, since instructions of ALU class do not require the data memory to be processed, read and write signals from the data memory are not activated.

We design the eBPF processor with a 5-stage pipeline: instruction fetch (IF), instruction decode (ID), execute (EXE), memory (MEM), and write back (WB). IF stage gets instruction from memory and increments program counter (PC). ID stage translates opcode into control signals and reads registers from the register file. EXE stage performs ALU operation and computes jump/branch targets. MEM stage accesses data memory if needed, and the WB updates the register file. This design follows the MIPS load-store pipeline architecture [11]. We add four pipelined registers (between the stages), the units (forwarding and hazard).

4.2. Data Memory (Optimized FIFO)

To avoid the overhead of copying the packet from the transfer FIFO to the processor data memory, we designed a new abstract data type called Data Memory FIFO (DM_FIFO) that works as a FIFO and as a data memory. It enables the eBPF processor to access packet's data with zero-copy and run load and store operations efficiently. It works with the NetFPGA datapath's modules (input arbiter and output queues) synchronously. It has 2MB (64 of depth x 256 of width) and can store up to 32 packets of 64 bytes.

The eBPF engine can start processing the packet even if the packet has not fully arrived yet. Inside DM_FIFO, for each word, we added a validation bit to indicate if it contains data from the new incoming packet. If the word is invalid (a piece of packet did not arrive yet), DM_FIFO informs that to the processor which switches to the idle stage and waits for the word to be stored (a piece of the packet to arrive). Thus, DM_FIFO brings two advantages: it does zero copy, and it allows the eBPF engine to start processing the packet before having to wait for the entire packet to arrive.

4.3. Instruction Memory

The eBPF instructions define the behavior of how the eBPF processor handles the packets. The instructions are inserted into the instruction memory via the NetFPGA register interface. Software registers were created to insert the eBPF instructions into the processor instruction memory. The instructions are received on the network element through messages sent by the controller. The instructions are then written to the software registers and forwarded to instruction memory via the hardware PCI bus.

As a design decision, we placed the instruction memory outside of the eBPF processor to enable the connection of multiple eBPF processors. By increasing the number of processors, more packets can be processed simultaneously, contributing to the increase the throughput.

4.4. Maps

eBPF Linux kernel implementation allows for maps. Maps are a generic data structure that stores different types of data in the form of key-value pairs. In our design, we currently provide three types of maps: most extended prefix matching (LPM), exact-match, and array map, which are provided in eBPFlow as hardware components. For LPM, we use the Ternary Content Addressable Memory (TCAM) module combined with the BRAM (Block RAM). TCAM is a specialized high-speed memory that does LPM searches very fast (1 cycle). TCAM uses three different inputs: 0, 1, and "don't care". Our TCAM module spends 16 cycles for a write operation and only one cycle for a read operation. For the exact-match, we use the Content-Addressable Memory (CAM) module combined with the BRAM. For the array map, we use the DRAM memory. CAM enables writing using two cycles instead of 16 cycles.

There are three functions to manipulate the maps: update, delete, and lookup. The update operation updates an item onto the map. If the item does not exist, it inserts the item. The delete operation removes the item with the given key. The lookup operation searches for the key and returns an item.

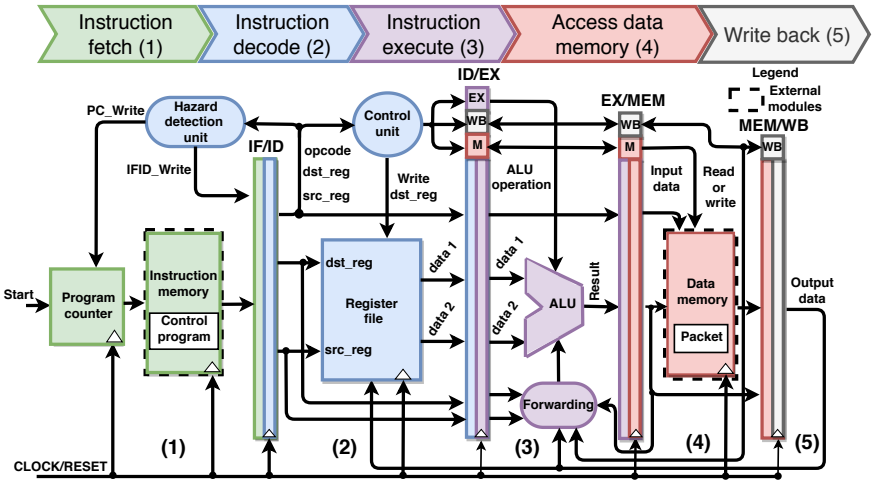

Figure 3. Control and Datapath of the eBPF processor.

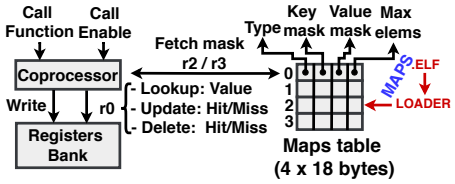

Figure 4. Coprocessor and Maps table.

eBPF programs	#Instructions	#LoC in C
Stateful Firewall	23	24
DDoS Mitigation	35	57
BitTorrent Filter	181	86
SQL Injection (Tautology)	143	110
SQL Injection (Sleep)	183	117

Figure 5. Applications implemented on eBPFlow.

The coprocessor needs to know the actual size of the data being read or written to the map's memory. That information is stored in a *maps table*, shown in Figure 4, which holds metadata about each map declared by the current loaded program. A lookup on maps table occurs on every map operation to retrieve key and value masks used in a bitwise-AND operation with the data to clean any unwanted bits.

4.5. User Space

It has a controller, a loader, and tools created at the user level. We implemented all in Python language.

Controller: It opens a socket connection TCP/IP to the device to exchange the Southbound messages. After after establishing the connection, the operator can transmit the eBPF program already compiled as bytecode. The controller install the bytecode in the hardware at runtime.

Loader: It is responsible for the following operations: loading code to the processors, appending two instructions, handling maps, and interaction with the processor register interface. *Loader* specially designed for the eBPF processor. At the beginning of every eBPF program, registers R1 and R10 need to be initialized with two pointers: one to the packet and one to the stack's top. Since these are specific to the runtime environment (here, the processor), such initialization is not part of the code generated by *clang* compiler. To handle maps, the compiler adds map information to the eBPF ELF file as a relocation section, which needs to be processed before code execution. *Loader* adjusts all map *call* instructions with their corresponding map values according to the relocation table in the ELF file. Finally, the *loader* interacts with the system through the register interface. Also, it can query status information about the processor.

Tools: A set of tools were implemented as part of the eBPFlow infrastructure: an eBPF disassembler, a software emulator, and a CLI application to interact with the eBPF engines. The emulator leveraged the uBPF [12]. GitHub project and aims to replicate the processor's behavior in software. It enables code testing and debugging with well-known tools such as *gdb*, enabling faster and easier bug detection and correction even before deploying the code.

5. Applications

We have implemented some network functions (NF) (Figure 5) to demonstrate the support of L7 classification in hardware. We present the number of eBPF instructions (#Instructions) and the number of C code lines (#LoC in C) for each application. Here is the description of them:

DDoS Mitigation (DDoS): mitigates DDoS attacks by analyzing and dropping potentially dangerous packets [13].

Stateful Firewall (SFW): is a network firewall that tracks the status and features of network connections, distinguishing packets for different types of communications and propagating only packets that match the active connections.

SQL Injection with Tautology (SQL_TAU): this attack is characterized by the insertion of tautologies in an SQL query, making them manipulable. For example, if the system has the query *SELECT * FROM Users WHERE Id = "username"* where *username* is a user-supplied parameter and no input filter exists, the attacker can exploit the vulnerability by sending the string *"OR 1 = 1*. The resulting query will be *SELECT * FROM Users WHERE Id = "" OR 1 = 1*, which is valid and returns all rows in the *Users* table, since *1 = 1* is always true. Figure 6 shows the FSM corresponding to the regular expression proposed to detect the threat. It consists of three states that lead to two possible final actions: PASS if there is no attack or DROP if the malicious string is detected before the end of the payload. The first state detects the beginning of the attack, rest by single or double-quotes. The second detects the mandatory presence of spaces between the beginning and the end of the attack, the keyword or, detected in the third state's entry and exit transitions. The current state of the FSM is reset every time the standard sequence is disobeyed.

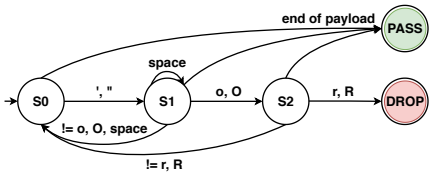

Figure 6. SQL Tautology RegEx.

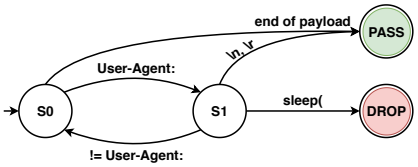

Figure 7. SQL Sleep RegEx.

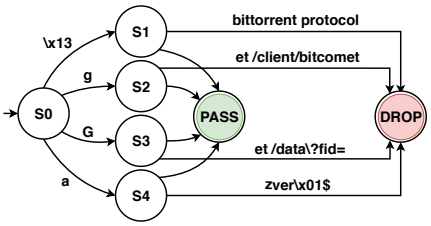

Figure 8. Bittorrent Filter RegEx.

SQL Injection with Sleep function (SQL_SLEEP): this attack allows hackers to look for possible SQL vulnerabilities on a server. It uses the User-Agent field of HTTP requests to send an SQL query that calls the function *sleep*, applying a delay in seconds to the current operation. During the delay period, any further requests received run only after the end of the first query, which indicates to the attacker that there are vulnerabilities that allow the insertion of other SQL attacks. Figure 7 shows the FSM corresponding to the regular expression of this example. The first state detects the presence of the User-Agent keyword or ends processing if the end of the payload is found. The second state looks for the *sleep (* string, which indicates the presence of the attack within the specified field. If a line break is found before this string, the processing is terminated.

The SQL injections presented above are examples of stateful functions that use regular expressions to analyze packet payload efficiently. This type of analysis is critical today, in which servers store a large amount of valuable data and demand defense against this and other types of attacks.

BitTorrent Packets (BITP): BitTorrent can cause many simultaneous connections, which can overload the network. This NF is able to detect four types of BitTorrent packets based on Strait and Sommer [14]. Figure 8 shows the FSM with five states to match the patterns composed of 64 characters. The first state is responsible for detecting the patterns' initial character at the beginning of the packet payload. The subsequent four states detect the rest of the strings. The packet is sent if the patterns are not found by the end of the payload.

6. Evaluation

The topology used during tests consisted of one NetFPGA SUME, one server running *pktgen-DPDK* [15] acting as a traffic generator and as a custom controller used to interact with the eBPFlow eBPF engines. Our server includes a SmartNIC with two 10 Gbps interfaces. Each 10 GbE port is connected directly to each of two active NetFPGA ports, sending network traffic at line rate, following the same configuration as [16]. i7-7700 3.60 GHz processors with eight cores and 8 GB of RAM hosted both the traffic generation server and the server holding the NetFPGA. Table 3 lists the logic and memory resource used by eBPFlow implementation on NetFPGA SUME. We compared the resource used by eBPFlow with a reference switch.

TABLE 3. NEFPGA'S RESOURCE EBPFLOW AND REFERENCE SWITCH.

Resource Type	Reference switch	eBPFlow
# Slice LUTs	49436 (11%)	81762 (21.16%)
# Block RAMs	194 (13%)	220.50 (15%)

Throughput: Figure 9 summarizes the measured packet processing rates between minimum-sized (64 bytes) and maximum-sized (1,500 bytes) packets, respectively, for each application for one interface (as in FlowBlaze [16]). DDoS mitigation, SQL Injection attacks, and Bittorrent packets achieve line rate (10 Gbps) for all packets size. These applications do not realize operations with maps using CAM/TCAM memories. Packets of 128 to 1,500 bytes for all applications achieve line rate without the overhead. Stateful Firewall had throughput affected for 64 bytes packets due to operations on the CAM map. The execution of lookup operations in CAM takes a clock cycle. Update operation in the respective memory takes three cycles which causes throughput reduction. It is not possible to reduce them because they are factory default values. There are extra clock cycles in the FSM internal control that manage the requisition of operations on the eBPF engine.

Latency: In addition to the throughput, we also measured the average and tail latencies for each application (Figures 10 and 11) using *pktgen-DPDK*, with 1 μs precision. We repeated each experiment 33 times for each application for packet sizes between 64 and 1,500 bytes. As expected, latency increased according to the packet size increase because the number of words on the data plane increased with eBPF engines and took more time to run the entire program. All experiments had a latency of less than 20 μs. The number of instructions of the applications loaded on the eBPF engines did not impact much on latency. This metric demonstrates little change in the processing time between same-sized packets for a single application, consequently leading to reduced jitter. Similarly, the tail latency is close to the average value in almost all cases. The bars in Figures 10 and 11 represent the standard deviation, which was close to zero in all cases.

Power: When idle, the NetFPGA device consumes 16 W. When we synthesize eBPFlow, the power consumption is 22 W regardless of the packet rate or running program. Meanwhile, a 1U x86 server [17] consumes on average 75 W/interface. Thus, eBPFlow saves power in comparison to other packet processing devices.

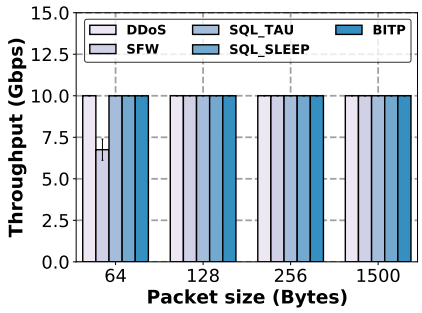

Figure 9. Throughput.

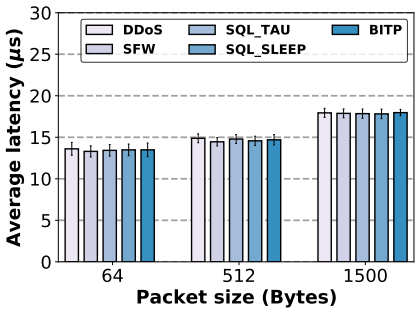

Figure 10. Average latency.

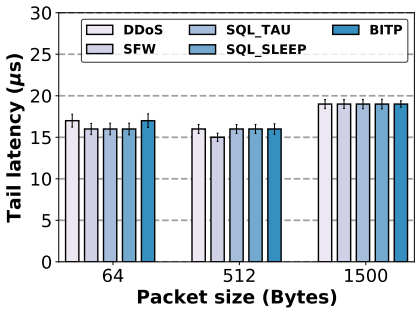

Figure 11. 99th-percentile latency.

7. Discussion

RegEx complexity vs. Performance: RegExs are transformed into eBPF instructions. Complex RegExs can generate more instructions increasing the program size. A large program (e.g., 256 instructions) can spend more processing time, harming the throughput.

Sessions: It is related to the number of lines in the memories, 32 lines. However, the use of wildcard rules allows for increasing the number of sessions using the wildcard operator. The rule just checks if the incoming packet belongs to an existing session. Moreover, the number of memories can be increased.

8. Related Work

NIDS and DPI. Zeek [18], and L7 Filter [14] are NIDS that use RegEx to identify content inside packets on the application layer. They run on the Linux kernel or in userspace, sharing CPU resources between other processes. When the number of packets that arrive on the host increases, processing overhead occurs due to the CPU being busy. It harms the performance of applications L7. NetFilterOffloader [19] proposed a NetFilter Firewall on NetFPGA 1G using Iptables. It does not support RegExs and L7 classification. Moreover, it uses hardware deprecated with a throughput of 4 Gbps. Some works exploit specialized hardware such as TCAMs, GPUs, and FPGAs [20]–[28]. None of these works presents the set of eBPFlow contributions together in one system.

Multi-stage match-action. RMT [29], dRMT [30], and FlexPipe [31] are reconfigurable chips that follow a match-action processing model. RMT [29] and its latest version dRMT [30] provides an architecture that allows reconfigurable matching tables for multi-stage packet processors implemented in ASIC. The architecture offers certain programmability and can process the packets in the input rate of the network interfaces. RMT can not execute regular expressions when parsing nor manipulate packet payload.

BPF related. BPFabric [32] is a platform in software that allows protocol-independent packet processing. It uses eBPF instructions to define packet processing methods and forwarding in the data plane. BPFabric was initially implemented over a Linux raw socket interface and later adapted over the DPDK. Katran [33] is an open-source eBPF load-balancer application provided by Facebook. It shows that the adoption of eBPF in the industry is a trend.

FPGA related. P4FPGA [34] is a platform developed in hardware that performs conversion of P4 programs to Verilog. ClickNP [35] also focuses on increasing programmability and flexibility. It provides ClickNP, a declarative language that can be compiled into an intermediate HDL and synthesized on the FPGA. To reprogram it, it requires 1∼2 hours due to HDL synthesis tools. FlowBaze [16] is an FPGA-based SmartNIC that allows stateful packet processing in hardware by programming using Extended Finite State Machines (EFSM). It cannot operate on packet payload and only supports storing 64 states. eBPFlow does not have these limitations since all FSM states and transitions are converted into instructions.

Smart NICs. SoftNIC [36] provides a programmable NIC interface with Click [37]. It is a different approach to provide network programmability. It provides new functionalities for host NIC through software. Netronome [38] provides a SmartNIC that is programmable through eBPF instructions. Netronome NIC can not compute DPI protocols because it does not process the packet payload. Furthermore, it does not allow wildcard lookups which is fundamental to provide fast access to rules or data on memory.

9. Conclusion

eBPFlow, composed of hardware and software components, is an application layer packet classifier in hardware. eBPFlow allows packet classification with DPI on NetFPGA SUME using eBPF, a software processor. We presented our designed and implemented packet classifier with multiple eBPF software processors at its core. The system supports regular expressions and security protocols without relying on any specific protocol. Also, it allows L7 classification with DPI on packet headers and payload in runtime at line rate. As future work, we plan to add a coprocessor per engine providing parallel access between eBPF engines and processing off-line on user space with new routines for the processing and analysis of already classified packets.

References

[1] D. Sanvito, D. Moro, and A. Capone, "Towards traffic classification offloading to stateful sdn data planes," in *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–4.

[2] A. Ribeiro and H. Pereira, "L7 classification and policing in the pfsense platform," in *21st International Teletraffic Congress (ITC 21), Paris, France*, 2009.

[3] G. A. Pimenta Rodrigues, R. de Oliveira Albuquerque, F. E. Gomes de Deus, G. A. De Oliveira Júnior, L. J. García Villalba, T.-H. Kim *et al.*, "Cybersecurity and network forensics: Analysis of malicious traffic towards a honeynet with deep packet inspection," *Applied Sciences*, vol. 7, no. 10, p. 1082, 2017.

[4] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3371038

[5] J. C. Penha, L. B. Silva, J. M. Silva, K. K. Coelho, H. P. Baranda, J. A. M. Nacif, and R. S. Ferreira, "Add: Accelerator design and deploy - a tool for fpga high-performance dataflow computing," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 18, p. e5096, 2019, e5096 cpe.5096. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5096

[6] L. B. D. Silva, R. Ferreira, M. Canesche, M. M. Menezes, M. D. Vieira, J. Penha, P. Jamieson, and J. A. M. Nacif, "Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[7] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[8] A. L. A. Zuquim, L. F. M. Vieira, M. A. Vieira, A. B. Vieira, H. S. Carvalho, J. A. Nacif, C. Coelho, D. da Silva, A. O. Fernandes, and A. A. F. Loureiro, "Efficient power management in real-time embedded systems," in *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 03TH8696)*, vol. 1. IEEE, 2003, pp. 496–505.

[9] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "Netfpga–an open platform for gigabit-rate network switching and routing," in *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, ser. MSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 160–161. [Online]. Available: http://dx.doi.org/10.1109/MSE.2007.69

[10] Xilinx, "Virtex-7 family overview," 2010. [Online]. Available: https://www.xilinx.com

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[12] Big Switch Networks, Inc., "Userspace ebpf vm," https://github.com/iovisor/ubpf, 2020, accessed on 06/22/2020.

[13] G. Bertin, "XDP in practice: integrating XDP into our DDoS mitigation," in *Proceedings of the Technical Conference on Linux Networking*, 2017, p. 5.

[14] M. Strait and E. Sommer, "L7 filter - bittorrent," http://l7-filter.sourceforge.net/layer7-protocols/protocols/bittorrent.pat, 2003, accessed on 06/22/2020.

[15] D. Turull, P. Sjödin, and R. Olsson, "Pktgen: Measuring performance on high speed networks," *Computer Communications*, vol. 82, pp. 39–48, 2016.

[16] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019.

[17] I. System, "Calculating tco for energy," 2011.

[18] The Zeek Project, "Zeek," https://www.zeek.org/, 2020, accessed in: 01/10/2020.

[19] M.-S. Chen, M.-Y. Liao, P.-W. Tsai, M.-Y. Luo, C.-S. Yang, and C. E. Yeh, "Using netfpga to offload linux netfilter firewall," in *2nd North American NetFPGA Developers Workshop*, 2010.

[20] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Sax-pac (scalable and expressive packet classification)," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 15–26.

[21] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 193–204, 2005.

[22] A. X. Liu and M. G. Gouda, "Complete redundancy removal for packet classifiers in tcams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 424–437, 2008.

[23] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 335–346, 2012.

[24] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, "Optimizing many-field packet classification on fpga, multi-core general purpose processor, and gpu," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015, pp. 87–98.

[25] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 213–224.

[26] R. D. Pacífico, L. B. Silva, G. R. Coelho, P. G. Silva, A. B. Vieira, M. A. Vieira, Í. F. Cunha, L. F. Vieira, and J. A. Nacif, "Bloomtime: space-efficient stateful tracking of time-dependent network performance metrics," *Telecommunication Systems*, pp. 1–23, 2020.

[27] W. Sun and R. Ricci, "Fast and flexible: parallel packet processing with gpus and click," in *Architectures for Networking and Communications Systems*. IEEE, 2013, pp. 25–35.

[28] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multilayer packet classification with graphics processing units," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2728–2741, 2015.

[29] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486011

[30] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098823

[31] R. Ozdag, "Ethernet switch fm6000 series- software defined networking.intel corporation," 2012.

[32] S. Jouet and D. P. Pezaros, "Bpfabric: Data plane programmability for software defined networks," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 38–48. [Online]. Available: https://doi.org/10.1109/ANCS.2017.14

[33] Facebook, "Katran source code repository," https://github.com/facebookincubator/katran, 2018.

[34] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 122–135. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050234

[35] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934897

[36] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[37] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[38] D. Beckett, J. Joubert, and S. Horman, "Host dataplane acceleration (hda)," New York, NY, USA, 2018.