DeepMatch: Practical Deep Packet Inspection in the Data Plane using Network Processors

Joel Hypolite
The University of Pennsylvania
jhypolit@cis.upenn.edu

John Sonchack
Princeton University
jsonch@princeton.edu

Shlomo Hershkop
The University of Pennsylvania
hershkop@cis.upenn.edu

Nathan Dautenhahn
Rice University
ndd@rice.edu

André DeHon
The University of Pennsylvania
andre@acm.org

Jonathan M. Smith
The University of Pennsylvania
jms@cis.upenn.edu

ABSTRACT

Restricting data plane processing to packet headers precludes analysis of payloads to improve routing and security decisions. DeepMatch delivers line-rate regular expression matching on payloads using Network Processors (NPs). It further supports packet reordering to match patterns in flows that cross packet boundaries. Our evaluation shows that an implementation of DeepMatch, on a 40 Gbps Netronome NFP-6000 SmartNIC, achieves up to line rate for streams of unrelated packets and up to 20 Gbps when searches span multiple packets within a flow. In contrast with prior work, this throughput is data-independent and adds no burstiness. DeepMatch opens new opportunities for programmable data planes.

CCS CONCEPTS

• **Networks** → **Deep packet inspection**; **Programming interfaces**; **Programmable networks**;

KEYWORDS

Network processors, Programmable data planes, P4, SmartNIC

ACM Reference Format:
Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2020. DeepMatch: Practical Deep Packet Inspection in the Data Plane using Network Processors. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3386367.3431290

1 INTRODUCTION

Data plane programmability is a powerful tool for dynamic, network-based optimizations. Concurrently, "big data", decentralized micro services, and the Internet of Things (IoT) are adding traffic to networks. Today's data plane processing, while useful for network telemetry and optimization, misses opportunities to better classify and route traffic using data that lie beyond the layer 3 and 4 headers. For example, malicious flows otherwise indistinguishable from other traffic can often be identified based on patterns that occur in packet payloads [28, 46, 55]. Flow classification, discussed in Sec. 2, shows similar benefits. In deep packet inspection (DPI), payloads are scanned for patterns written as regular expressions (regex), a more expressive and natural extension of existing data plane programming abstractions. Today's systems, comprised of software-defined networking (SDN) switches (which lack datapaths needed for DPI's payload processing) and the P4 language [19] (which falls short of the expressiveness needed to process payloads) force DPI solutions into middleboxes or hosts. Hardware and software needed to upgrade from header-only processing adds complexity and may be costly.

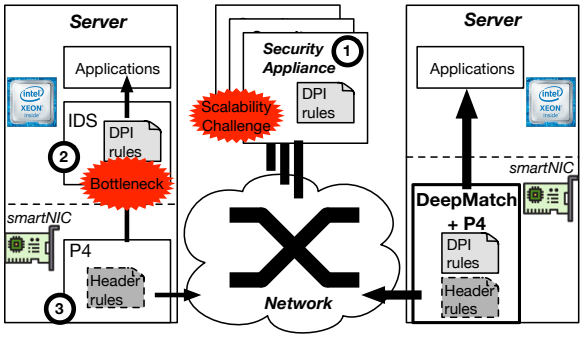


Figure 1: DPI today (left and network) is limited by performance, scalability, and programming/management complexities (shaded red) inherent to the underlying deployment models. DeepMatch (right) pushes DPI into the commodity SmartNICs currently used to accelerate header filtering and integrates DPI with P4, which improves performance and scalability while enabling a simpler deployment model.

Network processors (NPs) can meld payload matching into data plane processing. The DeepMatch design exploits the manycore parallelism available in NPs to achieve a high performance DPI regex matching capability integrated with P4 data plane programs. DeepMatch is tuned to exploit architectural characteristics of NPs, including complex distributed memory hierarchies, direct memory access (DMA), and multithreading.

DeepMatch's parallel processing scheme (Sec. 5) distributes packets to a tight Aho-Corasick deterministic-finite-automata (DFA) matching loop [12] running on the NP cores (Sec. 5.2). Latency-aware placement of DFA state within the NP 's memory hierarchy

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12.
https://doi.org/10.1145/3386367.3431290

lets multithreading mask access times and enables the manycore NP to achieve 40 Gbps line-rate pattern matching within packets. Regex matching across the packets comprising a flow requires support for reordering, as out-of-order arrivals must be correctly sequenced. As this is missing from P4, DeepMatch must provide it, increasing the complexity of packet handling due to the NP's limited number of active threads, limited memory for instructions and local state, and lack of dynamic memory allocation. DeepMatch 's novel resource placement and scheduling overcomes these limitations and serializes many interleaved flows concurrently using the NP, significantly expanding the power of P4. Notably, even as DFA state expands to occupy larger (and slower) memories, making memory speed a bottleneck, DeepMatch 's throughput is guaranteed, regardless of packet contents.

We benchmark our DeepMatch prototype to evaluate how it responds to specific types of traffic including increasing number of flows, out-of-order packets (OoO), and packet bursts (Sec. 6). DeepMatch illustrates design challenges for payload-processing at line-rate on NPs, proposes solutions engineered to overcome performance implications, and provides results from a thorough performance evaluation that guides more general payload-processing tasks.

Overall, this paper makes the following contributions:

- Demonstrate the first line rate DPI primitive, providing guaranteed data-independent throughput, embedded in a P4 dataplane using the Netronome NFP-6000 programmable NP (Sec. 5.4, Sec. 6.2)
- Demonstrate packet accounting and reordering to allow flow-based DPI processing on programmable NPs (Sec. 5.5, Sec. 6.3)
- Characterize performance achievable across a wide range of task and network characteristics, providing insight into the capabilities and limitations of programmable NPs (Sec. 6)
- Provide valuable lessons learned for future advanced uses of NPs involving payload processing tasks

We provide an open-source release of DeepMatch at:
https://github.com/jhypolite/DeepMatch

2 PACKET FILTERING

Packet filtering (or classification) is the process of categorizing packets in a network data plane [29]. It is a fundamental capability allowing a shared data plane to provide differentiated services, such as policy-based routing [22], monitoring [28], and security enforcement [52, 60, 66].

There are two types of packet filters. First, header inspection classifies based on layer 2—4 headers. It is typically used to distinguish between flows defined by endpoint or connection identifiers (e.g., `IP` address or `TCP/IP` 5-tuple). Second, DPI classifies flows based on patterns found in payloads. This enables finer-grained filtering and richer policies that would be difficult or impossible to implement using only information found in headers.

While both header filtering and DPI are critical in today's networks, DPI is significantly more expensive and challenging to scale, primarily due to the greater computation needed to perform per byte payload processing. Furthermore, as Fig. 1 (left and middle) illustrates, this gap is exacerbated due to the use of different platforms and deployment models.

DPI has been deployed in essentially the same way for decades: either atop dedicated appliances (i.e., middleboxes, Fig. 1 ①) or software at the endhost (Fig. 1 ②). DPI appliances use tightly optimized hardware and software to keep per-unit power cost low [42, 68, 72]. However, they are expensive in terms of unit cost, can become the choke point in a network, and are notoriously difficult to manage at scale [59]. The need to program them separately and differently from the data plane is one contributor to the management complexity.

Though running DPI engines on endhosts simplifies management [40, 48], it also places a computational burden on general purpose processors. Performance depends highly on both the input patterns and packet workload (Sec. 7). Further, extra background load on servers can have drastic impacts on application performance [24, 37], e.g., response time.

Unlike DPI, header filtering has become significantly less expensive and easier to scale. The key has been leveraging commodity programmable networking equipment and designing high-level abstractions that make header filtering policies easier to implement (Fig. 1 ③). For example, P4 programmable SmartNICs reached 10% of total controller and adapter market revenue in 2018, with estimates to be over 27% by 2021 [36]. These devices have the raw compute power to support custom filtering at or near line rate.

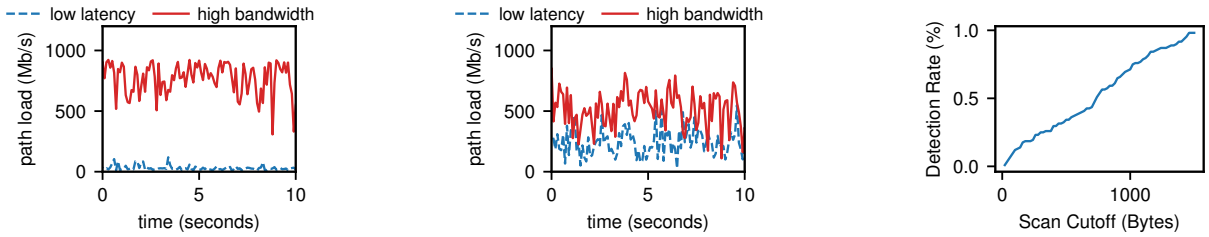
Equally important, SmartNICs are only marginally more expensive than their non-programmable counterparts,[1] which allows network operators to solve scalability issues by simply provisioning the commodity programmable elements in every server [2, 24] or switch.

The primary goal of DeepMatch, illustrated in Fig. 1 (right), is to leverage existing commodity network processing hardware to support not only header inspection, but also DPI at high line rates across the entire network with simple programming abstractions. This is challenging for two reasons. First, as we discuss in Sec. 4, real-time, network-processing hardware has inherent limitations on memory sizes in order to provide guaranteed high-throughput operation. Second, DPI is a much more computationally intensive and complicated task than header filtering. The computational intensity is inherent: finding a pattern that can start anywhere in a packet and has arbitrary length requires scanning every byte of a packet. Though simple string matching can have minimal computational complexity, regex matching can be more efficient when it is necessary to match many potential strings and variations.

3 P4 DPI INTEGRATION

Integrating DPI into P4 simplifies building advanced flow classification and security applications and integrating them with programmable forwarding policies. In this section, we highlight several specific motivating examples.

[1] As an example, the Netronome NFP-4000 2x40Gbps SmartNIC lists for around the same amount as the latest generation Mellanox ConnectX fixed-function NIC ($650) [6, 9] and uses similar amounts of energy under full load (25W) [5, 44]



(a) DeepMatch payload-content routing (b) routing with first 1024B of packet (c) Scan depth vs. detection rate

Figure 2: Sorting between low latency and high bandwidth paths for small (>1 KB) Redis object transfers

```
ingress {
       if ((tcp.dport == REDIS_PORT)
       ||  (tcp.sport == REDIS_PORT)) {
              logObjSize = scanPayload(packet); //
                   DeepMatch
              if (logObjSize != 0)
                     lastObjSize[pkt.key] = logObjSize;
              else if (lastObjSize[pkt.key] >= 5)
                     ip.tos = 1; // prefer high bandwidth.
              else
                     ip.tos = 0; // prefer low latency.
       }
       apply(forwardingTable);
}
```

Figure 3: DPI for application layer routing.

3.1 Redis Application Layer Routing

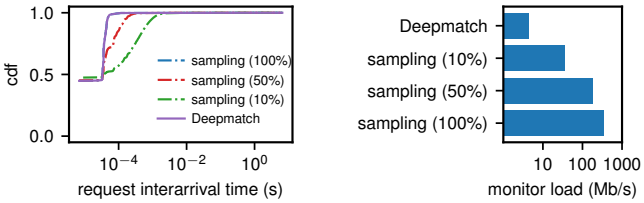
DPI enables fine-grained quality of service (QoS) policies based on application layer information [21, 38]. As an example, consider a network optimized to balance latency and throughput of a scaled out Redis [54] key-value database.

A typical deployment services two request classes: frequent requests for small latency-sensitive objects [33] and infrequent requests for large throughput-sensitive objects that increase overall latency by saturating network queues.

The goal of a Redis-aware QoS policy is to isolate the two types of requests, routing small objects on prioritized low latency paths and large objects on low priority paths with high bandwidth. Determining the size of a Redis object is conceptually simple: it is declared in the object's header. However, extracting this header correctly requires inspection of full packet payloads. A new object can start at any point in a TCP stream because Redis connections carry many requests. This means object requests may cross packet boundaries, and packets must be re-ordered to observe the full request.

We quantify the benefit of DPI for application-aware QoS by analyzing a 2.7GB Redis trace with 4 clients streaming requests to a single server, with a 90% / 10% split between small (1KB) and large (1MB) objects. The policy uses regex to extract object sizes and select network paths.

Fig. 2(a) shows the volume of traffic routed across each path. The DPI signatures parse every object's size prefix and ensure that only small objects are routed on the low latency path. As Fig.2(b) and (c) show, DPI is critical for correctness. Object headers are missed when less than the entire payload is scanned, causing transfers to be routed on the wrong paths.



(a) Accuracy (b) Load

Figure 4: DPI Signature Sampling.

This policy is straightforward to implement with DPI capabilities integrated into a P4 program. Fig. 3 shows pseudocode for the program. It scans payloads for regex that discriminate between object size prefixes with different orders of decimal magnitude, e.g., regular expressions of the form: `"\r\n\$.{`D`}\r\n"` , where D is the number of decimal digits in the object size value. The P4 program tracks the size of the most recent object transfer in each Redis flow and tags packets with the path ID for downstream switches.

3.2 Efficient Network Monitoring

We now look at prefiltering based on application layer metadata not visible to today's header-based prefilters [28, 43]. For example, consider a telemetry system measuring Redis request interarrival times and sizes. To measure these distributions, a telemetry backend needs to process packets that contain application layer request headers. However, the data plane needs DPI to identify these packets because a request header can begin anywhere in a TCP stream, including crossing packet boundaries.

Fig. 15 in App. A shows a P4 program using DPI for prefiltering. DeepMatch scans payloads for regular expressions that identify request types. For example, this expression identifies GET requests: `"\r\nGET\r\n"`. Whenever DeepMatch identifies a new request, the P4 program simply clones the packet to a backend telemetry server for measurement.

Fig. 4 quantifies the benefit of application layer prefiltering. It shows the accuracy (a) and workload (b) of a telemetry server measuring Redis GET request interarrival times. With DeepMatch DPI

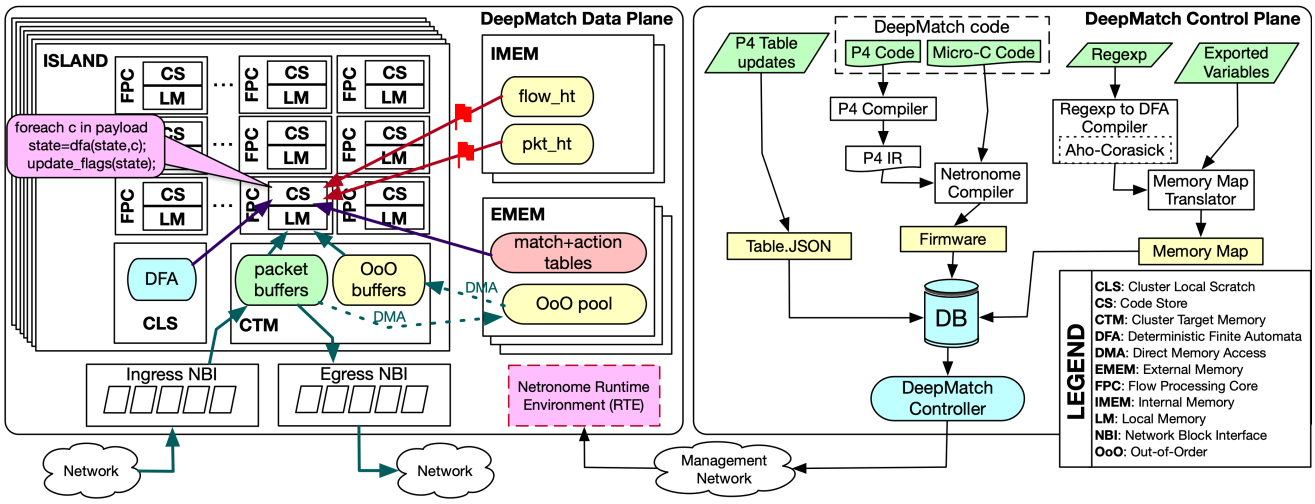


Figure 5: The DeepMatch Architecture

prefiltering, the server measured the exact interarrival distribution, while only processing the small fraction of packets containing request headers. In comparison, packet sampling had significantly lower accuracy while cloning orders of magnitude more packets to the server.

3.3 Packet Filtering for Security

IDS-style packet inspection (e.g. Snort [55] or Bro [46]) can be similarly integrated. We provide an example in App. B. It is particularly important to perform matches that cross packet boundaries for security [53]; otherwise, an adversary could deliberately arrange their malicious content to cross the boundary to evade detection. This, too, demands packet reordering.

4 DPI ON NETWORK PROCESSORS

The restrictions on today's P4-programmable data planes that only allow them to process headers makes realizing DPI challenging. NPs have considerable potential to perform DPI, but their design encourages the algorithm and decomposition to be architecture-aware.

4.1 Network Processor Architectures

NPs are manycore processors with cores specialized for guaranteed-throughput packet processing. They are more akin to real-time embedded processors than general-purpose cores. Small per-core footprints mean that many such cores can be packed onto a chip to maximize processing throughput. Hundreds of threads accessing shared mutable state would create a bottleneck, so a decentralized set of small memories local to each core is used to minimize contention; these scratchpad memories are managed by the programmer rather than the system to guarantee predictable timing. Programmers must manage hard constraints on code and data size.

These characteristics are common to most NPs (Tab. 1) and are driven by fundamentals such as performance, silicon structure cost, and demands of network processing; as such, these basic characteristics are likely to persist in future NPs.

Table 1: Network Processor Characteristics

Vendor	NP	cores avail.	clock freq	threads total
Cisco	FP 2017 [39]	672	1.00 GHz	2,688
Cisco	FP 2015 [1]	40	1.20 GHz	160
Microsemi	WinPath4 [8]	48	0.50 GHz	320
Microsemi	WinPath3 [7]	12	0.45 GHz	64
Netronome	NFP-6000 [73]	80	1.20 GHz	640 (320)
Netronome	NFP-4000 [10]	50	0.80 GHz	400 (200)

total threads in reduced thread mode in parentheses

The simplicity of the NP cores allows NPs to scale to support large volumes of traffic in a cost-effective way, but it also means they do not natively support applications written in multi-threaded C code targeted at general-purpose processors with larger memories, implicitly managed caches, and complex, best-effort (not real-time) processing cores, such as Instrusion Detection Systems (IDS). As Sec. 7 shows, best-effort processing on general-purpose cores can have high, data-stream dependent variation in throughput, making them less suitable for providing consistent QoS guarantees.

4.2 Netronome Target Architecture Details

DeepMatch targets the Netronome NFP-6000 SmartNIC, a 40 Gbps P4 programmable NIC. Characteristics and relevant components of the NFP-6000 that impacted the design and implementation of DeepMatch are shown in Fig. 5. While the parameters are NFP-6000 specific, the features and the fact that programs must be tuned to the specific numbers are common across NPs.

Processing Cores and Context. User code runs on up to 81 Flow Processing Cores (FPC) distributed on seven islands. FPCs are 32-bit RISC-based cores that run at 1.2 GHz and have 8 thread contexts. 648 threads are therefore available to a program. At most one thread is executing at a time on an FPC. The threads in a FPC are non-preemptive; threads must explicitly yield execution, and switching contexts takes 2 cycles. Each FPC has a private code store that can hold 8K instructions shared by its threads; these are not a cache

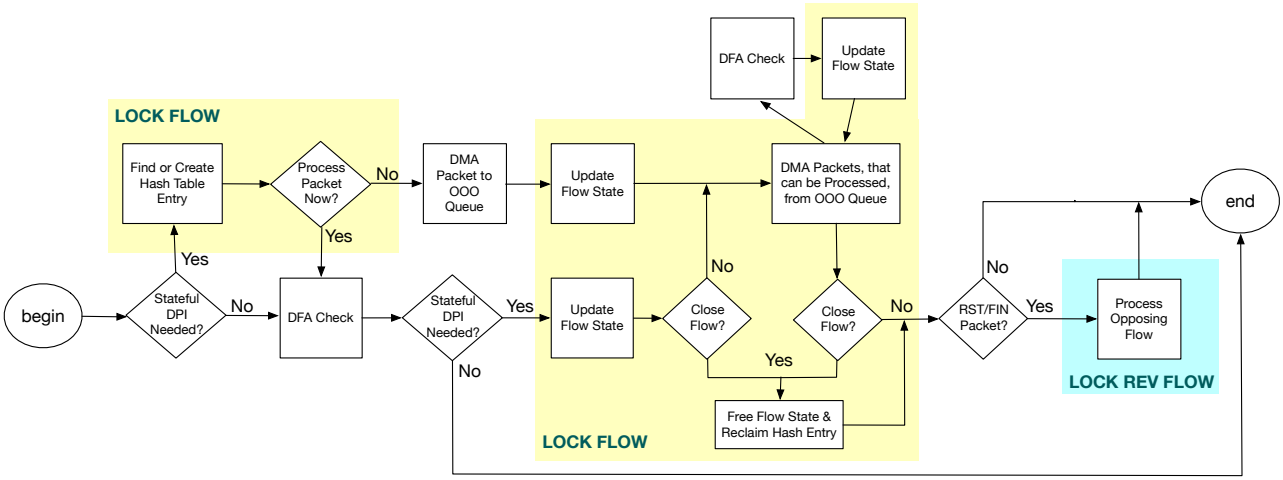


Figure 6: DeepMatch Flow of Execution for Intra- and Inter-Packet Regular Expression Matching

Table 2: Netronome memory hierarchy [73]

Memory	Size	Latency (cycles)
Code Store (CS)	8 K Instrs.	1
Local Memory (LM)	4 KB	1-3
Cluster Local Scratch (CLS)	64 KB	20-50
Cluster Target Memory (CTM)	256 KB	50-100
Internal Memory (IMEM)	4 MB	150-250
External Memory (EMEM)	2 GB	150-500

of active portions of a program; the entire application must be described in 8K instructions (or 16K in shared-code mode). Each FPC has 256 32-bit general-purpose registers shared amongst its 8 threads, or 32 registers per thread. If additional registers are needed per thread, the NFP may run in a reduced thread mode with four active threads per FPC, each with 64 registers.

Memory Hierarchy and Latency. FPCs have access to large, shared global memories and small, local memories that are fast but require programmer management. This hierarchy is shown in Tab. 2. Upon issuing a read/write memory request, a FPC thread context switches and waits until its request is handled. This overlaps memory latency with computation in a different thread.

Packet Handling. When a packet arrives, the ingress Network Block Interface (NBI) copies it to the CTM packet buffer of an available FPC. By default, the first 1024 bytes of a packet are copied to CTM, with the remainder copied to IMEM. DeepMatch doubles the ctm_split_length to 2048 bytes to accommodate a complete 1500 byte MTU in the CTM buffer. This wastes ca. 500 bytes but maximizes payload processing performance. Each FPC processes its packet to completion, at which point it is sent to the egress NBI for transmission.

DMA Engines. The NFP-6000 provides DMA engines for fast and safe, semaphore-protected data transfers. There is a limit of 16 outstanding DMA commands per CTM.

Programming Languages. The NFP-6000 can be programmed in both Micro-C and P4. Micro-C is an extended subset of C-89 [73]. It is limited by FPC capabilities, e.g., no recursion, no variable argument lists to functions, no dynamic memory allocation. P4 is supported with a compiler extension that translates P4 code into Micro-C. Native Micro-C functions can be called by P4 programs.

5 SYSTEM DESIGN

DeepMatch is designed to provide fast and comprehensive DPI, supporting two modes that offer different design points between these goals. First, DeepMatch provides a stateless intra-packet regex matching capability that is carefully designed to achieve sustained peak processing rate when processing full payloads (Sec. 5.4). Second, DeepMatch provides a stateful inter-packet regex matching capability (Sec. 5.5). Content specified by a regex may appear anywhere in a flow (Sec. 3), even crossing packet boundaries. Thus, DeepMatch must support reordering of packets and scanning across the ordered payload of an entire TCP stream. Packet reordering is not natively supported by P4, so we describe a capability that will also be important to other advanced uses of NPs.

Fig. 5 illustrates DeepMatch's architecture. Newly arrived packets are dispatched to worker threads running on FPCs. When inter-packet matching is desired, FPCs consult and update flow state to continue matches across packets (Fig. 6). Out-of-Order (OoO) packets are buffered in large, shared memory until they can be processed in order.

5.1 Key Challenges

The key challenge to guaranteed real-time, line-rate payload handling is to simultaneously satisfy high computation, memory, and data transfer requirements. This constrains the computation that can be performed per payload byte and the tolerable memory latency. Automatically-managed caches would make memory latency variable, so NP architectures avoid caches, forcing the programmer to explicitly move data to control memory latency. Similarly, programmers must limit data-dependent branching and looping to guarantee worst-case computation and memory access time. In some cases, tradeoffs among compute, memory, and data transfer are needed to maximize achievable performance. We show how we address these issues for DPI exploiting the architectural features in NPs with a general methodology that forms the starting point for

analysis and implementation of other line-rate payload handling tasks on NPs.

5.2 Regular Expression Matching Strategy

Each FPC performs matching within a packet. To acheive high-throughput matching, we must minimize the instructions and memory access required to process each byte of the payload. DeepMatch uses the Aho-Corasick algorithm [12] to compile regex into a deterministic finite automaton (DFA) [17]. The DFA allows a process to check all patterns simultaneously by making a single state transition per byte. This avoids backtracking and results in guaranteed, constant work per byte of input, important for real-time, line-rate processing. The DFA is compactly implemented using a state transition table that maps a current state and a byte to a next state. *End states* store flags indicating that one or more patterns matched, and an *end state table* maps end states to the set of patterns matched. DeepMatch's DFA match loop examines every byte in the payload. As such, the throughput achieved is not payload data-dependent.

5.3 Integrating P4 and DPI on Netronome

DeepMatch is implemented as an external P4 action, written in Micro-C [73]. P4 code parses and filters packet headers and metadata. P4-defined match-action tables, populated in the control plane at run-time, determine actions to apply to the packet based on the parsed headers. DeepMatch is invoked as a table action setting metadata for subsequent tables to match on (e.g., Fig. 16). The control plane compiles regex into a DFA and loads DeepMatch at runtime.

5.4 Stateless Intra-Packet Regex Matching

Packet payloads are processed by the core DFA matching loop (Sec. 5.2). Nominally, this requires that we read one character from the payload, perform one lookup into the DFA state transition table, and perform a number of instructions to manipulate the data, setup the memory references, and handle loop control. Our implementation required 15 instructions per input character to execute a DFA state update.

We use a simple throughput analysis to compute an upper bound on the network throughput the NP can sustain:

$$N_{cores} \times F_{core} \times CPB \geq \text{Network Throughput} \quad (1)$$

where N_{cores} is the number of cores, F_{core} is core frequency, CPB is the cycles the core must execute per input bit.

Assuming throughput limits computation, rather than memory latency which we address next, the N_{cores}=81 FPCs running at F_{core}=1.2 GHz, can support about 50 Gbps:

$$81 \times 1.2 \times 10^9 \text{cycles/s} \times \frac{8\text{b/byte}}{15\text{cycles/byte}} = 52\,\text{Gbps} \quad (2)$$

FPCs can exploit threading to hide memory latency. However, the memory latencies for performing the payload byte read and DFA state lookup can be high (Tab. 2). Accessing a payload in CTM can take 100 cycles. Similarly, reading a DFA table from CTM costs 100 cycles. To fully hide the read latency, we would need 200/15+1 = 14 threads running on each FPC, which is greater than the FPCs can support.

In general, we hide memory latency by switching to other threads (Fig. 7). When all threads are running the same computation, each

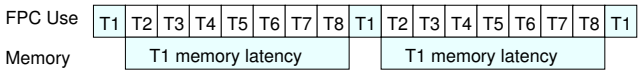


Figure 7: Using Threads to Hide Memory Latency

thread can run its compute cycles, $Compute_{cycles}$, 15 here, while another thread is waiting on memory access latency. When the total memory latency is L_{mem}, this gives a constraint:

$$L_{mem} < Compute_{cycles} \times (N_{threads} - 1) \quad (3)$$

Alternately, since $N_{threads} = 8$ for the NFP-6000, we can use Eq. 3 to see that we can afford a memory latency of about 15×7=105 cycles to approach the full computational throughput of the FPCs. This implies the DFA transition table cannot be placed in the large EMEM, with a memory latency of 150–500 cycles, if we aim to operate at 40 Gbps (line rate processing). Further, we may need to avoid performing per byte reads from the payload in CTM.

The payload byte read penalty is reduced by transferring the payload in 32-byte batches from the CTM packet buffer to local memory via transfer registers. Each batch is then processed byte-by-byte. The performance gains are significant, as CTM access times are 50-100 cycles, compared to 1-3 cycles for local memory. For example, a 1024-byte payload requires approximately 76,800 cycles to access all bytes when read one at a time from the CTM buffer, whereas, the batched approach requires only 4,448 cycles, or about 5 cycles per byte read. Performing the batched transfers does require more FPC instruction cycles (20), bringing our peak performance from FPCs down to about 40 Gbps.

$$81 \times 1.2 \times 10^9 \text{cycles/s} \times \frac{8\text{b/byte}}{20\text{cycles/byte}} = 39\,\text{Gbps} \quad (4)$$

This is an example where we trade more computation (larger $Compute_{cycles}$) in order to reduce memory latency that would otherwise place a larger limit on computational throughput. An equivalent way of formulating the memory latency effect in Eq. 3 is to represent the effect of memory latency on throughput:

$$N_{cores} \times F_{core} \times \frac{N_{threads} - 1}{L_{mem}} \times bits \geq \text{Network Throughput} \quad (5)$$

Here, $bits$ is the number of bits processed on each read (or reads) requiring the L_{mem} memory latency cycles. The achievable throughput is the minimum of Eq. 1 and Eq. 5.

Eq. 4 shows us 20 cycles per byte is the upper bound on available computation for *any* payload processing application to run at the 40 Gbps line rate on the NFP-6000.

Placing the DFA transition table in CLS keeps memory latency low enough (50 cycles) that we can, potentially, achieve the full 40 Gbps DPI throughput. Since CLS memories are local to a cluster, DFA transition tables must be replicated in each cluster (8 copies). This limits performance loss and variability from memory contention, which can occur on the larger, shared memories, which is not modeled in our simple equations above.

5.5 Stateful Inter-packet Regex Matching

An extended DeepMatch can support regex matching across packet boundaries within a flow. This forces us to evaluate packets in flow order (serialization), requiring per-flow state for Out-of-Order(OoO) packet storage and retrieval. Thus, throughput available to a single

flow is limited and aggregate performance depends on the number of flows.

5.5.1 Finding patterns across packet boundaries. DeepMatch maintains per-flow DFA state and sequences packets in a flow using TCP sequence numbers. Immediately after a packet is processed (see Fig. 6), the OoO buffer is checked for queued packets; this maximizes the per flow processing rate.

The flow state resides in IMEM while the OoO packet data is kept in EMEM. There are two 4MB IMEM memory engines. As each flow needs a 28B record, there is an upper limit of 290K flows. Since IMEM is split and shared (about 10% is used for system variables), the actual limit is lower. 50K flows occupies about one third of one of the IMEMs. There are three EMEM memory engines, each with 8 GB of DRAM. Each EMEM can hold 5.4 million maximum length (1500B) packets that can be split between flows and OoO packets per flow.

$$EMEM_{OoO\_capacity} \geq N_{flow} \times N_{OoO\_per\_flow} \tag{6}$$

About 30% of total EMEM capacity is used for system variables. Using 4 GB, half of one of the EMEMs, for the OoO buffers, and assuming 100 OoO packets per flow ($N_{OoO\_per\_flow}$), over 25K flows can be supported. As references to this state occur only once per packet, longer access times are tolerable; as access by any FPC can occur, these memories must be accessible by all FPCs. DMA engines offload data transfers of packet data to and from EMEM.

For our prototype, we sized the packet buffers to track 320 flows (N_{flow}) with 100 OoO packets per flow ($N_{OoO\_per\_flow}$). This is in line with requirements for operation in data center NICs, which typically observe from 10s to 1000s of concurrent flows [56]. With 320 flows, and 28B records, the flow state table in IMEM needs less than 10KB of memory. DeepMatch manages locks on shared state to avoid stalls (Fig. 6). Locks are flow-specific and are held only during brief per-packet operations; they are specifically not held during packet DMA operations and payload scanning.

5.5.2 Performance. Supporting inter-packet regex matching requires additional per-packet handling to consult and maintain per flow state. For large packets, this cost is mostly amortized across the payload bytes. The header processing time goes from 3,309 cycles per packet in the intra-packet case to 8,221 cycles per packet in the inter-packet flow matching case; this means header processing time can dominate for payloads smaller than 411 Bytes.

$$Header_{cycles} \geq CPB \times 8 \times N_{bytes} \tag{7}$$

$$\frac{Header_{cycles}}{CPB} = \frac{8221}{20} = 411 \geq N_{bytes} \tag{8}$$

The core DFA matching loop remains unchanged, so the same basic performance phenomena are in play, but the combination of the limits in the NFP architecture, including the size of the local store and the number of registers per thread, combined with the additional code and state needed for inter-packet regex work to reduce the performance we can extract. Finally, OoO packets must be sent to and retrieved from larger memories, introducing additional time and throughput limits on packet processing.

Increased Code and Data Requirements. With the additional inter-packet matching support described above, the DeepMatch code compiles to 12,729 instructions. Each FPC code store has an 8K instruction limit, but the shared code store option on the NFP allows a larger image to be split between two paired FPCs, effectively doubling the instruction limit, but potentially adding contention on instruction fetch. We use the default code-splitting mode, where even instructions are placed in one code store and odd instructions in the other code store. Shared code mode demands an even number of FPCs, so we can only use 80 (instead of 81) FPCs in this mode. As shown in Fig. 9, we do see performance impacts from contention on the instruction memories.

Due to the added code complexity, we hit the limit for the compiler register allocator and local memory spillage, and thus, DeepMatch must run in reduced-thread mode when supporting inter-packet regex matching. In this mode, only four threads are available per FPC reducing the ability to tolerate memory latency by half (Eq. 3). The benefit is that each of the four threads has access to twice as many context relative registers. Combined with shared code mode's requirement to have an even number of MEs, the result is a maximum of 80×4=320 concurrent threads.

Impact on DFA matching loop. Dropping to 4 threads, reduces our ability to tolerate memory latency (Eq. 3) to about 20×3=60 cycles. By itself, that might not impact performance when the DFA transition table is in CLS, but it will make the performance loss higher for the larger memories. The shared-code operation may double the number of cycles for instruction execution from 20 to 40. At the extreme, this now brings our potential performance down to about 20 Gbps.

$$80 \times 1.2 \times 10^9 \text{cycles/s} \times \frac{8\text{b/byte}}{40\text{cycles/byte}} = 19\,\text{Gbps} \tag{9}$$

OoO Data Movement. When packets arrive in order, the basic dataflow does not change. The packet is stored in the CTM which is located within the cluster of FPCs and does not change the time for data access. OoO packets must be sent to EMEM and then retrieved from EMEM to the CTM. DMA engines make this more efficient than simply reading individual words from the high latency EMEM, but there is added latency to recover a packet from EMEM. Furthermore, the limit of 16 concurrent DMA transfers means contention effects can further limit performance.

Per Flow Performance. A single flow is now essentially serialized to operate on a single FPC. Multithreaded latency hiding does not matter to single flow throughput. With 20 instructions per byte of processing and 50 cycles of CLS latency, a single flow will be limited to about 140 Mbps.

$$1.2 \times 10^9 \text{cycles/s} \times \frac{8\text{b/byte}}{(20+50)\text{ cycles/byte}} = 0.137\,\text{Gbps} \tag{10}$$

Additional flows add linearly at first, with each getting its own FPC. Even at 80 flows, while the FPCs share instruction memories, most of the cycles are in memory wait rather than instruction fetch. Eventually, instruction contention becomes a significant effect as noted above.

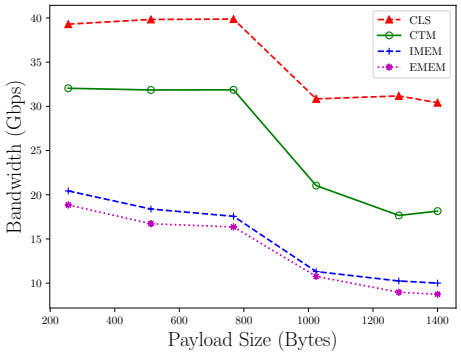


Figure 8: Intra-packet Regex Matching Performance: DFA location and payload size affect throughput when shared code mode and reduced threads mode are off.

6 EVALUATION

We evaluate DeepMatch with a series of benchmarks that characterize the performance achieved under a variety of task and network scenarios.

6.1 Experimental Setup

6.1.1 Testbed. We benchmark DeepMatch in a simple two node topology. The NFP card that runs DeepMatch is installed in a Dell PowerEdge R720 with dual Intel Xeon E5-2650 v2 8-core 2.60 GHz processors and 64 GB DDR3 1600MHz RAM. It is connected, via 40 GbE cables, to a traffic generation server. The traffic generation server is a Dell PowerEdge R720 with dual Intel Xeon E5-2680 v2 10-core 2.80 GHz processors, 256 GB DDR3 1866 MHz RAM, and a Mellanox ConnectX-4 40/100 GbE NIC. It uses dpdk/pktgen [3] for replay and capture.

6.1.2 Measurements. We measure the *lossless throughput* of DeepMatch: the maximum sustained rate at which DeepMatch can operate without dropping a single packet. We use packet traces of synthetic TCP flows that we generated to control the following factors: packet size, flow count, flow bandwidth, packet ordering, and burst size. To determine throughput, an automated script repeats trials that send packets to DeepMatch at a target transmit rate then checks for packet drops. Trials continue with decreasing transmit rates until no drops are detected. For stateful flow scanning, the script also ensures that all flows close properly.

6.2 Stateless DeepMatch

Fig. 8 shows we achieve line rate (40 Gbps) with the DFA transition table in CLS memory up to 800 byte packets. Throughput drops for larger memories since the 8 threads are not sufficient to hide the memory latency performing the DFA lookups. The performance drop between 800 byte and 1024 byte packets is very distinct and repeatable, but we have not been able to isolate a specific mechanism or effect that clearly explains the performance drop.

We highlight typical packet sizes reported by CAIDA and Facebook. The range of mean and median packet sizes reported by the CAIDA nyc (dirA) monitor for an 11 month period (March 2018 - January 2019) is 785-924 bytes and 865-1400 bytes respectively [26]. Facebook provides the distribution of packet sizes in a datacenter for four host types [56]. They report Hadoop traffic as bimodal, either MTU length (1500 bytes) or TCP ACK. The median packet size for other services is less than 200 bytes with less than 10% of packets fully utilizing the MTU.

The DeepMatch results in Fig. 8–14 do not depend on the specific regex being matched beyond the size of the DFA. This is an advantage of the real-time, guaranteed-throughput design. Fig. 8 and others show separate performance curves for the different memories implied by the DFA sizes. Tab. 4 shows how various pattern rulesets map to DFA sizes and memory requirements. Simple filtering tasks like PII scanning easily fit in the fast, local CLS memory. Larger Snort rule sets for more complex IDS tasks must be placed in slower memories.

6.2.1 Discussion. Since the small local memories are such a large benefit, it may be worthwhile to compress the transition tables so that they fit into smaller, faster memories. In particular, the transition tables are often sparse. This could potentially benefit from DFA compression techniques in the literature [23, 35]. Sparse table handling often requires more computational instructions to unpack or interpret the transition table representation. This is a case where the computational complexity must be balanced with the compactness gain to maximize net performance.

6.3 Stateful DeepMatch

To benchmark stateful DeepMatch, which scans across the ordered payload of an entire TCP stream, we vary four additional aspects of our workloads. First, the number of flows in a dataset are varied from 1 to 320—the maximum number of concurrently executing threads. This allows us to determine how the flow rate varies from the single flow rate up to peak utilization. Second, we vary the datasets OoO-ness using an algorithm that allows us to turn the knob on the number of OoO packets. The algorithm swaps the last two packets in every length k sequence, resulting in a dataset with 1/k OoO packets. This works up to a maximum 50% OoO when k=2. This has the effect that there is no more than one packet OoO at a time. Third, we vary the burstiness of the packets in the dataset by sending k consecutive packets of a single flow, followed by sending k rounds of single packets from the remaining flows. This has the effect of creating OoO-ness while keeping a constant aggregate flow rate. Lastly, we round-robin iterate through each flow sending k consecutive packets from a flow at each iteration. This forces DeepMatch to handle bursts from some flow and process OoO packets in other flows simultaneously.

Since the code for stateful DPI is complex, shared code store and reduced threads mode (4 threads per FPC) are set for all these trials.

6.3.1 Results. Fig. 9 shows DeepMatch throughput for 320 flows with the DFA transition table in CLS memory. The inter-packet flow matching achieves over 20 Gbps on large packets, consistent with expectations (Eq. 9). Smaller packets are penalized more by the additional per-packet processing needed to maintain flow state, with throughput dropping below half (10 Gbps) when header processing begins to dominate below 400 Byte packets, consistent with our estimates of header processing time (Sec. 5.5.2). The four additional, intra-packet matching curves help explain the impact of reduced

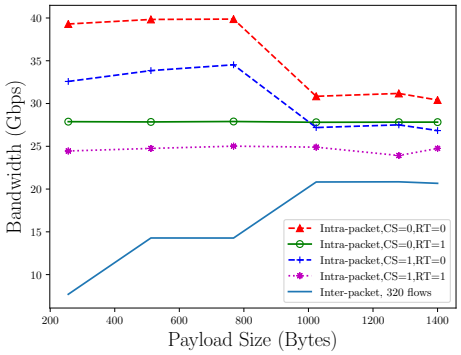


Figure 9: Intra- and Inter-Packet Performance on 320 Flows. CS=shared code mode, RT=reduced thread mode (CS=1 and RT=1 for the Inter-packet case).

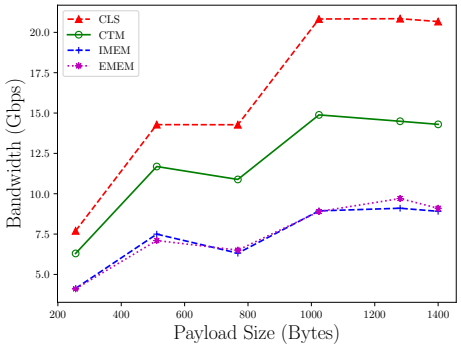


Figure 10: Inter-Packet Performance on 320 Flows

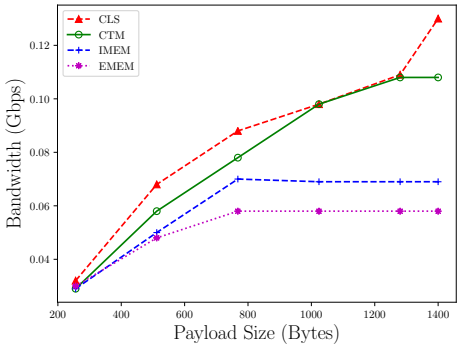


Figure 11: Single Flow Inter-Packet Performance

thread and shared code operating modes. The top curve with all 8 threads and no shared code operation is the same CLS performance we see in Fig. 8. The other three curves show the effects of reducing threads and sharing code. We see those effects alone are responsible for reducing peak performance to 24 Gbps. Maintaining flow state only adds an additionl 4 Gbps loss in throughput.

Fig. 10 shows that as we move to slower memories than the CLS, we see performance drops as expected.

As noted, the single flow performance is limited. Fig. 11 shows how this varies with memory and packet size. As expected, large packets in CLS achieve 0.140 Gbps. Fig. 12 shows how peak throughput increases with flows. Particularly for large packets, it scales roughly linearly as expected.

Fig. 13 shows the impact of OoO packets on performance. We vary the fraction of packets that are OoO. Packets that arrive in-order are processed directly from the CTM, while all packets that arrive before the next expected sequence number are sent to EMEM and retrieved in order. The percentage denotes the fraction of packets that must be sent to EMEM. This shows a drop to 15 Gbps for 5% OoO packets and graceful degradation as OoO-ness increases.

A large scale study of a Tier-1 IP Backbone measured approximately 5% of packets out of order [32]. In data centers, reordering can be significantly lower because data plane algorithms are typically engineered to minimize reordering [13] because of its effect on TCP throughput.

Fig. 14 shows the impact of traffic bursts on performance. If a burst of packets arrive for a single flow, they must be buffered and sequentialized at the single packet flow rate (Fig. 11). This effectively means the packets in that flow are treated the same as OoO packets. Except for the first packet in the flow, packets in the burst must be copied to EMEM until they can be sequentially processed by the FPC processing the first packet. Fig. 14 shows little performance degradation effects from bursts despite the fact these incur additional data transfers. The eager handling of OoO traffic allows us to sustain full rate even on flows that are serialized to a single FPC. The care to not hold locks during long operations means that little FPC processing capacity is lost while coordinating the storage and retrieval of these OoO packets. DCTCP [14] measures the workloads in three production clusters. They report packet bursts are limited by the TCP window and at 10 Gbps line rates, hosts tend to send bursts of 30-40 packets.

6.3.2 Discussion. While 40 Gbps line rate is not possible in the flow-based case, under a larger set of scenarios it is possible to support 10 Gbps traffic. This is still a healthy network rate for many clients and servers.

There is a large performance drop for the flow-based case that comes from the larger code requirements and the larger amount of state. It is possible that a tighter implementation could reduce the code and state. As such, the intra-packet case serves as a limit estimate on the additional performance achievable with a tighter implementation.

With more sophisticated code splitting or replication, it is likely possible to reduce or eliminate the impact of shared code, bringing the flow-based case closer to the intra-packet matching case. For example, the critical DeepMatch loop code is a small fraction of the total code. If this code were replicated in the instruction memories and accessed locally, the main performance impact of shared code would go away.

Our implementation shows that 40 Gbps line-rate payload processing is possible on the NFP-6000. For *any* payload inspection task (e.g., feature extraction, anomaly detection [70]), *it will be necessary to keep cycles per byte below 20 and memory latency per byte below 105 cycles to achieve the full 40 Gbps line rate.* More generally, Eq. 1 and 5 provide the first order budgeting constraints for this class of NPs.

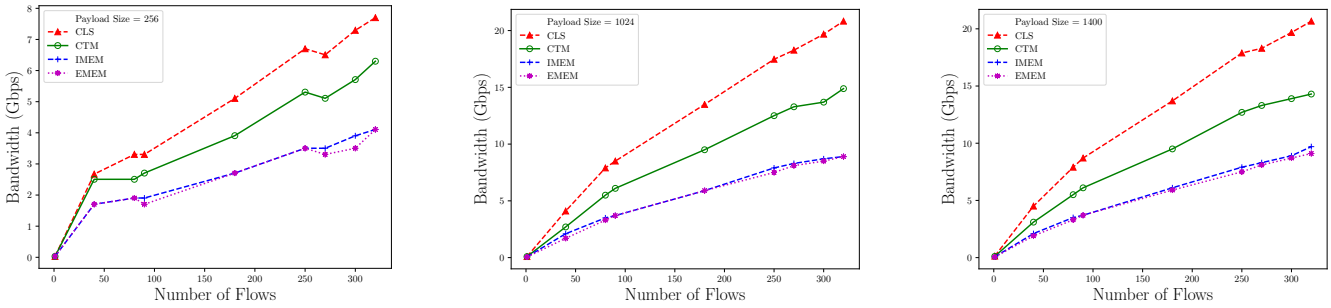


Figure 12: Inter-Packet Performance versus Flows (In-Order Packets)

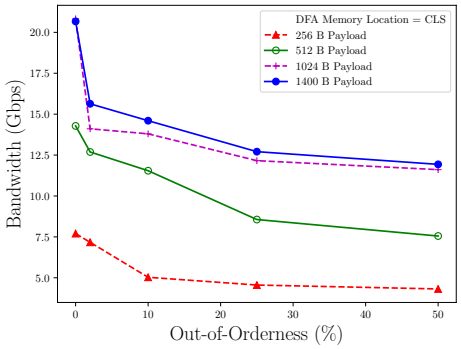


Figure 13: Inter-Packet Bandwidth vs. OoO (320 Flows)

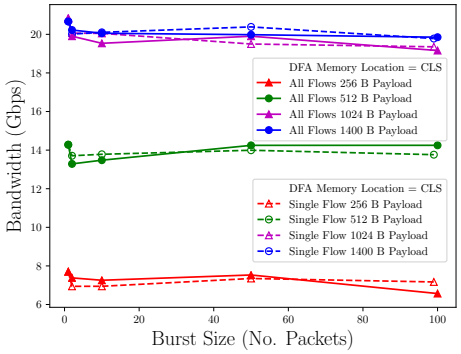


Figure 14: Inter-Packet BW vs. Burst (320 Flows)

Table 3: Server-Class Machines used for Evaluation

Name	CPU	Cores	Clock (GHz)	Power (W) Idle	HS
High Clock	E3-1270 v6	4	3.8	30	89
Many Core	E5-2683 v4	32	2.6	135	389

"HS" = Hyperscan

7 COMPARISON WITH HYPERSCAN

We compare DeepMatch with Hyperscan, a heavily optimized state-of-the-art DPI library that runs on general-purpose processors [71, 72] and is a best-in-breed representative of current multi-core pattern matchers. Hyperscan aggressively exploits direct string matching to reduce DFA sizes and to maximally exploit SIMD matching support on modern Intel IA64 architectures. Hyperscan exploits common-case optimizations making its performance highly data dependent; its performance varies to the extent the data streams it processes match these common cases that it exploits.

We use two hosts to evaluate Hyperscan (Tab. 3). Each with Intel Xeon CPUs (with SSE_3, $SSE_{4.2}$, POPCNT, BMI, BMI_2, and AVX_2 support). and dual-port 40 Gbps Mellanox ConnectX-4 NICs.

We compare DeepMatch and Hyperscan performance on diverse rulesets (Tab. 4), including Emerging Threats [31], Snort [11], Redis [54] application layer routing (Sec. 3.1), personally identifiable information (PII), and network monitoring (Sec. 3.2). We run Intra- and Inter-packet pattern matching trials using various 768 byte/320 flow packet datasets, including random and ruleset-specific near-match payloads.

The "% patterns" columns in Tab. 4 show the percentage of patterns that remain after filtering for matcher compatibility. In some cases (e.g. community all, emerging threats all) Hyperscan only supports a small fraction of the rules. In these cases, DeepMatch supports a larger fraction while maintaining the same performance. This also makes the comparison optimistic for Hyperscan, since properly supporting those patterns would likely degrade throughput.

Raw Hyperscan does not perform packet reordering. The processor cores will need to spend cycles processing and reordering packets. This typically requires 1 cycle/bps [25], suggesting the two systems will saturate at 15.2 Gbps and 83 Gbps even without Hyperscan running. We measure TCP receive and reorder on our test machines and see they both achieve 5 Gbps on a single core at 3.8 GHz and 2.6 GHz. Generously assuming perfect scaling, this means 20 Gbps and 160 Gbps using all processors on the machines to perform TCP reception and reordering. For a simple calculation, we generously assume the servers can reorder traffic at the rate identified above while running Hyperscan. We report the feasible throughput as the minimum of the throughput supported by Hyperscan pattern matching and TCP reordering.

The results in Tab. 5 show that whereas DeepMatch performance is data independent, Hyperscan performance is highly data dependent. To illustrate this, we show two columns of results for Hyperscan on each server. One column includes random traffic, while the second includes traffic designed to be near-matches to patterns in the pattern set. This near-match pattern set is designed to be a worst-case for Hyperscan, defeating many of its common-case optimizations that allow it to perform simpler processing when it can

Table 4: Regular Expression patterns and their effect on DFA size and performance

Pattern Set	Pattern Type	Total Patterns	HyperScan % Patterns	DeepMatch % Patterns	DeepMatch # DFA States	DeepMatch DFA Size (bytes)	DeepMatch Memory
scada	string	1	100%	100%	11	5632	CLS
emerging-icmp	string	16	100%	100%	37	18944	CLS
PII	regex	3	100%	100%	96	49152	CLS
emerging-telnet	regex	3	100%	100%	10	5120	CLS
redis	regex	8	100%	100%	31	15872	CLS
netmon	regex	1	100%	100%	8	4096	CLS
protocol-finger	string	14	100%	100%	142	72704	CTM
protocol-imap	string	25	100%	100%	355	181760	CTM
emerging-shellcode	regex	15	93%	100%	349	178688	CTM
os-mobile	regex	8	88%	100%	189	96768	CTM
emerging-p2p	string	143	100%	100%	4340	2222080	IMEM
protocol-ftp	string	18	100%	100%	554	283648	IMEM
emerge mobile_mal	regex	43	26%	98%	1537	786944	IMEM
emerging-scada	regex	8	75%	100%	857	438784	IMEM
server-other	string	2118	100%	100%	86837	44460544	EMEM
emerging-trojans	string	9608	100%	100%	412676	211290112	EMEM
emerging-trojans	regex	1496	30%	96%	742765	380295680	EMEM
server-mail	regex	93	91%	95%	3642492	1864955904	EMEM
emerging-pop3	regex	16	100%	100%	34524	17676288	EMEM
community all	string	134	100%	100%	3464	1773568	IMEM
emerging threats all	string	5546	100%	100%	243857	124854784	EMEM
community all	regex	546	54%	84%	3631070	1859107840	EMEM
emerging threats all	regex	5159	32%	90%	2121305	1086108160	EMEM

"PII" = personally identifiable information, "emerge mobile_mal" = emerging_mobile malware

quickly classify non-matches. To generate a near-match payload for a specific pattern, we use a library [4] to generate an exact-match payload, then truncate it by one byte at a time until it no longer matches. Hyperscan often slows down by over an order of magnitude when processing near-match traffic. While Hyperscan can often outperform DeepMatch on random traffic, the near match traffic almost always runs slower on the "high clock" machine. This highlights DeepMatch's ability to provide guaranteed, data-independent throughput.

Hyperscan achieves some of its most impressive speeds running pure string matching tasks where it can directly exploit the SIMD datapath to match multiple characters per cycle. In contrast, when Hyperscan must handle more complex regex, its performance can drop significantly (e.g., protocol-finger, emerging-trojans). DeepMatch performance is more predictable, depending only on the memory placement based on the size of the DFA for the rule set.

The "high clock" and "many core" servers consume 89 W and 389 W, respectively (Tab. 3). The DeepMatch NFP-6000 consumes 40 W. For 40 Gbps intra-packet matching, DeepMatch requires only 45% and 10% the energy of the servers. Since the "high clock" server is limited to 20 Gbps when reordering packets, DeepMatch also provides this energy benefit for inter-packet matching. Even if we generously assume the "many core" server can support two 40 Gbps network ports with reordering, DeepMatch is still only 20% the energy of the server. For near-match traffic, the servers running Hyperscan drop below DeepMatch performance, providing an even greater energy advantage to DeepMatch.

8 RELATED WORK

DeepMatch builds on prior work investigating payload-based intrusion detection on NPs [18, 47, 50], offering orders of magnitude more performance and features such as flow scanning and integration with a high-level data plane language.

Recent studies leveraging SDN to improve security [57, 61] have made control planes more secure [74], and security applications easier to implement [60] and more capable [69]. Data plane efforts have focused on security extensions to defend against various attacks [45, 62, 65, 66]. DeepMatch compliments these efforts by adding support for DPI and demonstrating a path to SmartNIC and P4 integration.

Prior work on data plane programming abstractions have focused on header processing [16, 19, 41, 49, 58]. DeepMatch extends this to harness rich payload data.

Efforts to improve server-based DPI performance have focused on reducing overheads [34] or offloading work to GPUs [27, 63, 64, 67] and accelerators [42]. In contrast, DeepMatch guarantees data-independent throughput and runs on more energy efficient NPs [51].

There has also been work on optimizing pattern matching algorithms. Chen et al. [20] and Hua et al. [30] increase throughput by redesigning Aho-Corasick to process multiple bytes of input per

Table 5: DeepMatch and Hyperscan Performance (*all results in Gbps)

Pattern Set	Pattern Type	DeepMatch Mem.	DeepMatch Intra-	DeepMatch Inter-	Hyperscan High Clock Intra- Rnd	Hyperscan High Clock Intra- Near	Hyperscan High Clock Inter- Rnd	Hyperscan High Clock Inter- Near	Hyperscan Many Core Intra- Rnd	Hyperscan Many Core Intra- Near	Hyperscan Many Core Inter- Rnd	Hyperscan Many Core Inter- Near
scada	string	CLS	40	14	210	7.4	20	7.4	839	29	160	29
emerging-icmp	string	CLS	40	14	160	12	20	12	830	160	160	160
PII	regex	CLS	40	14	30	13	20	13	130	51	130	51
emerging-telnet	regex	CLS	40	14	160	5.1	20	5.1	740	23	160	23
redis	regex	CLS	40	14	188	12	20	12	830	39	160	39
netmon	regex	CLS	40	14	198	10	20	10	840	18	160	18
protocol-finger	string	CTM	32	11	48	2.0	20	2.0	210	10	160	10
protocol-imap	string	CTM	32	11	190	6.8	20	6.8	780	31	160	31
emerging shellcode	regex	CTM	32	11	100	3.3	20	3.3	440	18	160	18
os-mobile	regex	CTM	32	11	480	47	20	20	1200	220	160	160
emerging-p2p	string	IMEM	18	8.0	16	4.8	16	4.8	70	20	70	20
protocol-ftp	string	IMEM	18	8.0	190	2.0	20	2.0	170	12	160	12
emerge mobile_mal	regex	IMEM	18	8.0	190	2.4	20	2.4	760	13	160	13
emerging-scada	regex	IMEM	18	8.0	160	35	20	20	710	190	160	160
server-other	string	EMEM	16	6.4	3.8	1.5	3.8	1.5	17	6.9	17	6.9
emerging-trojans	string	EMEM	16	6.4	1.8	0.97	1.8	0.97	7.7	4.2	7.7	4.2
emerging-trojans	regex	EMEM	16	6.4	17	1.2	17	1.2	62	6.0	62	6.0
server-mail	regex	EMEM	16	6.4	5.9	1.2	5.9	1.2	26	5.4	26	5.4
emerging-pop3	regex	EMEM	16	6.4	150	31	20	20	640	160	160	160
community all	string	IMEM	18	8.0	99	9.5	20	9.5	460	50	160	50
emerging threats all	string	EMEM	16	6.4	45	4.2	20	4.2	200	19	160	19
community all	regex	EMEM	16	6.4	28	4.1	20	4.1	110	20	110	20
emerging threats all	regex	EMEM	16	6.4	0.35	0.37	0.35	0.37	1.8	1.8	1.8	1.8

"PII" = personally identifiable information, "emerge mobile_mal" = emerging-mobile_malware

operation. Such optimizations compliment DeepMatch and can be integrated into future versions.

9 CONCLUSION

Programmable data planes enable further progress in in-network processing. While initially limited to header processing, network processors make payload processing viable, but getting high throughput on multigigabit links without compromising QoS remains tricky. DeepMatch can examine all data in a packet (e.g., for DPI) at 40 Gbps. More complexity (e.g., larger regular expression matching, flow matching, reordering) reduces throughput, but DeepMatch sustains throughput greater than 10 Gbps while preserving QoS. DeepMatch is a natural extension to P4 header processing, and the approach of offloading sophisticated DPI to SmartNICs is inexpensive and energy-efficient.

ACKNOWLEDGMENTS

This work is funded in part by the Office of Naval Research under grant N000141512006 and N000141812557. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

REFERENCES

[1] 2015. The Cisco Flow Processor: Cisco's Next Generation Network Processor. (2015), 13. https://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.pdf

[2] 2017. Ericsson Cloud SDN with Netronome Agilio® Server Networking Platform Achieves Massive TCO Savings in Cloud Data Centers. https://www.netronome.com/m/redactor_files/SB_Netronome_Ericsson_Cloud.pdf. (2017).

[3] 2019. Data Plane Development Kit (DPDK). https://www.dpdk.org/. (2019).

[4] 2019. EXREX. https://github.com/asciimoo/exrex. (2019).

[5] 2019. HP EDR InfiniBand Adapters (Mellanox ConnectX-5). https://h20195.www2.hpe.com/v2/GetPDF.aspx/c04950955.pdf. (2019).

[6] 2019. Mellanox ConnectX-5 EN Dual Port 40 Gigabit Ethernet Adapter Card. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3394. (2019).

[7] 2019. Microsemi WinPath3. https://www.microsemi.com/product-directory/winpath/4120-winpath3. (2019). Accessed: 2019-09-11.

[8] 2019. Microsemi WinPath4. https://www.microsemi.com/product-directory/winpath/4122-winpath4#overview. (2019). Accessed: 2019-09-11.

[9] 2019. Netronome Agilio CX Dual-Port 40 Gigabit Ethernet SmartNIC. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3018. (2019).

[10] 2019. Netronome Agilio SmartNICs. https://www.netronome.com/products/agilio-cx/. (2019). Accessed: 2019-09-11.

[11] 2019. Snort Community Ruleset. https://www.snort.org/downloads#rules. (2019). Accessed: 2019-07-11.

[12] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (June 1975), 333–340. https://doi.org/10.1145/360825.360855

[13] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 503–514.

[14] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010.

Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[15] Kostas G Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos D Keromytis. 2005. Detecting targeted attacks using shadow honeypots. (2005).

[16] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 29–43.

[17] G. Berry and R. Sethi. 1986. From Regular Expressions to Deterministic Automata. *Theor. Comput. Sci.* 48, 1 (Dec. 1986), 117–126. http://dl.acm.org/citation.cfm?id=39528.39537

[18] Herbert Bos and Kaiming Huang. 2004. A network intrusion detection system on IXP1200 network processors with support for large rule sets. (2004).

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.

[20] Chien-Chi Chen and Sheng-De Wang. 2012. A multi-character transition string matching architecture based on Aho-Corasick algorithm. *Int. J. Innovative Comput. Inf. Control* 8, 12 (2012), 8367–8386.

[21] Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano. 2014. ndpi: Open-source high-speed deep packet inspection. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*. IEEE, 617–622.

[22] Hilmi E Egilmez, Seyhan Civanlar, and A Murat Tekalp. 2013. An optimization framework for QoS-enabled adaptive video streaming over OpenFlow networks. *IEEE Transactions on Multimedia* 15, 3 (2013), 710–715.

[23] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. 2008. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 29–40.

[24] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[25] Annie P Foong, Thomas R Huff, Herbert H Hum, Jaidev R Patwardhan, and Greg J Regnier. 2003. TCP performance re-visited. In *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.* IEEE, 70–79.

[26] Center for Applied Internet Data Analysis. 2019. The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces. https://www.caida.org/data/passive/passive_trace_statistics.xml. (November 2019).

[27] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator.. In *NSDI*. 83–96.

[28] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 357–371.

[29] Pankaj Gupta and Nick McKeown. 2001. Algorithms for packet classification. *IEEE Network* 15, 2 (2001), 24–32.

[30] Nan Hua, Haoyu Song, and TV Lakshman. 2009. Variable-stride multi-pattern matching for scalable deep packet inspection. In *INFOCOM 2009, IEEE*. IEEE, 415–423.

[31] Proofpoint Inc. 2019. Emerging Threats Rule Server. https://rules.emergingthreats.net/. (2019). Accessed: 2019-07-11.

[32] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. 2007. Measurement and classification of out-of-sequence packets in a tier-1 IP backbone. *IEEE/ACM Transactions on Networking (ToN)* 15, 1 (2007), 54–66.

[33] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 9.

[34] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas

[35] Sailesh Kumar, Jonathan Turner, and John Williams. 2006. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '06)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/1185347.1185359

[36] David Levi and Mark Reichenberg. 2018. Ethernity Networks Company Overview: Making Truly Programmable Networks a Reality. (November 2018).

[37] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[38] Y. Li and J. Li. 2014. MultiClassifier: A combination of DPI and ML for application-layer classification in SDN. In *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*. 682–686. https://doi.org/10.1109/ICSAI.2014.7009372

[39] James Markevitch and Srinivasa Malladi. 2017. A 400Gbps Multi-Core Network Processor. (August 2017).

[40] McAfee. 2014. Next-Generation IPS Integrated with VMware NSX™ for Software-Defined Data Centers. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/products/nsx/vmware-nsx-mcafee-solution-brief.pdf. (2014).

[41] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev. (CCR)* 38, 2 (2008).

[42] Jaehyun Nam, Muhammad Jamshed, Byungkwon Choi, Dongsu Han, and Kyoung-Soo Park. 2015. Haetae: Scaling the Performance of Network Intrusion Detection with Many-Core Processors. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 89–110.

[43] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 85–98.

[44] Linley Newsletter. 2016. Netronome Optimizes iNIC for Cloud. https://www.linleygroup.com/newsletters/newsletter_detail.php?num=5475&year=2016&tag=3. (February 2016).

[45] Taejune Park, Yeonkeun Kim, and Seungwon Shin. 2016. Unisafe: A union of security actions for software switches. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 13–18.

[46] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-Time. (1999).

[47] V. Paxson, R. Sommer, and N. Weaver. 2007. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In *2007 IEEE Sarnoff Symposium*. 1–7. https://doi.org/10.1109/SARNOF.2007.4567341

[48] Justin Pettit, Ben Pfaff, Joe Stringer, Cheng-Chun Tu, Brenden Blanco, and Alex Tessmer. 2018. Bringing platform harmony to VMware NSX. *ACM SIGOPS Operating Systems Review* 51, 1 (2018), 123–128.

[49] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: a programming system for NIC-accelerated network applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 663–679.

[50] Piti Piyachon and Yan Luo. 2006. Efficient memory utilization on network processors for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 71–80.

[51] Gergely Pongrácz, László Molnár, Zoltán Lajos Kis, and Zoltán Turányi. 2013. Cheap silicon: a myth or reality? picking the right data plane hardware for software defined networking. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 103–108.

[52] S. Pontarelli, M. Bonola, and G. Bianchi. 2017. Smashing SDN "built-in" actions: Programmable data plane packet manipulation in hardware. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. 1–9. https://doi.org/10.1109/NETSOFT.2017.8004106

[53] Thomas H. Ptacek and Timothy N. Newsham. 1998. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. https://apps.dtic.mil/sti/pdfs/ADA391565.pdf. (January 1998).

[54] Redis. 2018. Redis. https://redis.io/. (February 2018).

[55] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.

[56] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 123–137.

[57] Sandra Scott-Hayward, Gemma O'Callaghan, and Sakir Sezer. 2013. SDN security: A survey. In *2013 IEEE SDN For Future Networks and Services (SDN4FNS)*. IEEE, 1–7.

[58] Niraj Shah, William Plishker, and Kurt Keutzer. 2004. NP-Click: A programming model for the Intel IXP1200. In *Network Processor Design*. Elsevier, 181–201.

[59] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 13–24.

[60] Seungwon Shin, Phil Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proc. Network and Distributed System Security Symposium (NDSS)*.

[61] Seungwon Shin, Lei Xu, Sungmin Hong, and Guofei Gu. 2016. Enhancing network security through software defined networking (SDN). In *2016 25th international conference on computer communication and networks (ICCCN)*. IEEE, 1–9.

[62] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 413–424.

[63] Pragati Shrivastava and Kotaro Kataoka. 2016. FastSplit: Fast and Dynamic IP Mobility Management in SDN. In *26th International Telecommunication Networks and Applications Conference, ITNAC 2016, Dunedin, New Zealand, December 7-9, 2016*. 166–172. https://doi.org/10.1109/ATNAC.2016.7878803

[64] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking Abstractions for GPU Programs. *ACM Trans. Comput. Syst.* 34, 3, Article 9 (Sept. 2016), 31 pages. https://doi.org/10.1145/2963098

[65] John Sonchack, Anurag Dubey, Adam J Aviv, Jonathan M Smith, and Eric Keller. 2016. Timing-based reconnaissance and defense in software-defined networks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 89–100.

[66] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan Smith. 2016. Enabling Practical Software-defined Networking Security Applications with OFX. https://doi.org/10.14722/ndss.2016.23309

[67] Weibin Sun and Robert Ricci. 2013. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13)*. IEEE Press, Piscataway, NJ, USA, 25–36. http://dl.acm.org/citation.cfm?id=2537857.2537861

[68] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. 2008. Gnort: High performance network intrusion detection using graphics processors. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 116–134.

[69] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. 2018. Towards fine-grained network security forensics and diagnosis in the sdn era. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 3–16.

[70] Ke Wang and Salvatore J. Stolfo. 2004. Anomalous Payload-Based Network Intrusion Detection. In *Recent Advances in Intrusion Detection*, Erland Jonsson, Alfonso Valdes, and Magnus Almgren (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–222.

[71] Xiang Wang et al. 2019. HyperScan. https://www.hyperscan.io. (2019).

[72] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, 631–648. http://dl.acm.org/citation.cfm?id=3323234.3323286

[73] Stuart Wray. 2014. The Joy of Micro-C. https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf. (December 2014).

[74] Menghao Zhang, Guanyu Li, Lei Xu, Jun Bi, Guofei Gu, and Jiasong Bai. 2018. Control plane reflection attacks in SDNs: new attacks and countermeasures. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 161–183.

```
ingress {
   if ((tcp.dport == REDIS_PORT)
   ||  (tcp.sport == REDIS_PORT)) {
      requestType = scanPayload(packet); // DeepMatch
      if (requestType != 0){
         apply(cloneTable); // clone to monitor.
      }
   }
   apply(forwardingTable);
}
```

Figure 15: DPI for fine-grained telemetry filtering.

A P4 DPI INTEGRATION FOR NETWORK MONITORING

Fig. 15 shows how DeepMatch DPI flow classification is integrated for the network monitoring task.

B PACKET FILTERING FOR SECURITY

Fig. 16 shows a longer concrete example: a $P4_{14}$ program[2] that uses DeepMatch to embed an advanced DPI-based security policy into a header-based P4 forwarding program. In a SmartNIC-based data center, this program could be pushed to the SmartNIC in every server to directly enforce global security policies at high line rates.

The program in Fig. 16 breaks down into two general stages: header/DPI filtering and policy enforcement. We describe each stage and an example of how it could be used to secure a deployment of Apache servers below.

B.1 Header and DPI filtering

The filtering stage classifies packets based on their headers and the presence of patterns in their payloads. In Fig. 16's program, `secFilterTable` is a match-action table that selectively applies DeepMatch based on source IP address. Each entry in the table maps a source IP address to an invocation of either the `noOp` action, which does nothing, or the `deepMatch` payload scanning action. An entry also stores action parameters for `deepMatch` that determine which rule set to use and whether to match across packet boundaries. In a server running Apache, a P4 SmartNIC could apply an application-specific ruleset to detect threats (e.g., the server-apache ruleset benchmarked in Tab. 4) in flows from untrusted external hosts.

B.2 Policy enforcement

Based on the output of the filtering stage, the policy enforcement stage determines how a packet should be handled. In Fig. 16's program, `secPolicyTable` uses `secMeta.dpiRuleMatchId`, the output of DeepMatch, along with the packet's destination IP address and TCP / UDP port to determine whether to allow, drop/alert, redirect, or rate limit a packet. For the drop/alert and redirect scenarios, the policy is enforced by altering the contents of the packet header and setting a metadata flag (`secMeta.policy`) that prevents the standard forwarding table from being invoked. Rate limiting is enforced by a downstream table that uses the *P4* metering primitive to determine if a flow is exceeding a threshold rate. In the Apache webserver example, this stage could drop packets that match malware rules and are destined for a port running an Apache service, redirect matching packets destined for non-Apache ports to a honeypot server [15], and rate limit flows that match Denial-of-Service (DoS) rules.

[2]The core DeepMatch Micro-C function could also be integrated into $P4_{16}$ as an external function or object.

```
// Per-packet security related metadata.
header_type secMeta_t {
   fields {
      dpiRuleMatchId : 16; // Match pattern ID
      policy       : 8;  // Security policy
      meterColor   : 8; } // Flow rate meter
}
metadata secMeta_t secMeta;
// Entry point for parsed packets.
control ingress {
   apply(secFilterTable); // Apply DPI
   apply(secPolicyTable); // Enforce policy
   if (secMeta.policy == RATELIMIT) {
      apply(rateLimitTable); }
   // Forward if no policy violations
   if (secMeta.policy == PERMIT) {
      fwdControl(); }
}
table secFilterTable {
   reads { ipv4.srcAddr : ternary; }
   actions { deepMatch; noOp; }   }
table secPolicyTable {
   reads {
      ipv4.dstAddr : exact;
      tcpUdp.dstPort : exact;
      secMeta.dpiRuleMatchId : exact; }
   actions { permit; deny;
      rateLimit; honeypot; }
}
table rateLimitTable {
   reads { secMeta.meterColor : exact; }
   actions { permit; noOp; } }
// Allow packet.
action permit() {
   modify_field(secMeta.secPolicy, PERMIT); }
// Drop packet & clone to monitor w/ match info
action deny(collectorCloneSpec) {
   modify_field(secMeta.secPolicy, DENY);
   clone_i2i(collectorCloneSpec, secMetaFields); }
// Rate limit this flow.
action rateLimit(flowId) {
   modify_field(secMeta.secPolicy, RATELIMIT);
   meter(ddosMeter, meta.flowKeyHash, secMeta.meterColor); }
// Redirect packet to honeypot.
action honeypot(honeypotAddr, egrPortId) {
   modify_field(ipv4.dstAddr, honeypotAddr);
   modify_field(meta.egrPortId, egrPortId); }
// Invoke deepMatch & set secMeta.dpiRuleMatchId.
extern action deepMatch(flowOrdering, rulesetId);
```

Figure 16: P4 Integration of DPI Packet Classification