Achieving 100Gbps Intrusion Prevention on a Single Server

Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, Justine Sherry
Carnegie Mellon University

Abstract

Intrusion Detection and Prevention Systems (IDS/IPS) are among the most demanding stateful network functions. Today's network operators are faced with securing 100Gbps networks with 100K+ concurrent connections by deploying IDS/IPSes to search for 10K+ rules concurrently. In this paper we set an ambitious goal: Can we do all of the above in a single server? Through the Pigasus IDS/IPS, we show that this goal is achievable, perhaps for the first time, by building on recent advances in FPGA-capable SmartNICs. Pigasus' design takes an FPGA-first approach, where the majority of processing, and all state and control flow are managed on the FPGA. However, doing so requires careful design of algorithms and data structures to ensure fast common-case performance while densely utilizing system memory resources. Our experiments with a variety of traces show that Pigasus can support 100Gbps using an average of 5 cores and 1 FPGA, using 38× less power than a CPU-only approach.

1 Introduction

Intrusion Detection and Prevention Systems (IDS/IPS) are a critical part of any operational security deployment [39, 40]. Such systems scan packet headers and payloads to check if they match a given set of *signatures* containing a series of strings and regular expressions. Signature rulesets are obtained through offline techniques (*e.g.*, crafted by experts or obtained from proprietary vendor algorithms) [6].

A recurring theme in IDS/IPS literature is the gap between the workloads they need to handle and the capabilities of existing hardware/software implementations. Today, we are faced with the need to build IDS/IPSes that can support line rates on the order of *100Gbps* [14] with *hundreds of thousands* [11] of concurrent flows and capable of matching packets against *tens of thousands of rules* [6]. This paper answers this challenge with the Pigasus FPGA-based IDS/IPS which meets the above goal *within the footprint of a single server*.

An important technology push that enables our effort is the emergence of server SmartNICs [29, 31]. Here, FPGA capabilities have become embedded in commodity server NICs. Of the various classes of high-performance accelerators (*e.g.*, GPUs), SmartNIC FPGAs are an especially promising alternative in terms of cost-performance-power tradeoffs, if they can be harnessed appropriately. Indeed, recent efforts have demonstrated the promise of FPGAs for low power, low costs and high performance for some simpler network functions such as software switching in Microsoft's AccelNet [21].

While many before us have integrated FPGAs with IDS/IPS processing [10, 16, 18, 19, 22, 34, 41–43, 46, 48, 49], for the most part these have focused on offloading only a specific functionality (*e.g.*, regular expression matching) to the FPGA. Unfortunately, traditional offloading cannot close the order-of-magnitude performance gap to offer 100Gbps IDS/IPS processing within the footprint of a single server. Even if regular expression search were *infinitely fast*, Snort 3.0 would still on average operate at 400 Mbps/core, requiring 250 cores to keep up with line rates! For orders of magnitude improvements, an accelerator has to improve performance for a *majority* of processing tasks, not just a small subset.

Hence in designing *Pigasus*, we argue for an *FPGA-first architecture in IDS/IPS processing*. Here, the majority of packet processing for IDS/IPS is performed via a highly-parallel datapath on-board the 100Gbps SmartNIC FPGA. Pigasus FPGA performs TCP reassembly directly on the FPGA so that it can immediately apply exact string matching algorithms over payload data to determine which "suspicious" packets need to enter a "full match" mode requiring additional string matches and regular expression matching. Inverted from the classic FPGA offload paradigm, Pigasus FPGA leaves to the CPU only the final match stage to check a small number of signatures on excerpts of the bytestream (on average, 1.1 signatures/packet and 5% of packets are sent to the CPU). By processing most benign traffic on the FPGA, the Pigasus FPGA-first architecture can reach 100Gbps and 3μs latency in the common case.

A natural consequence of Pigasus' FPGA-first approach is that we are now faced with supporting stateful packet processing on FPGA. The challenge includes not only multi-string pattern matching for *payload matching* but also *TCP bytestream reassembly* to be both performed at 100Gbps line rate. (Out-of-order TCP packets must be reassembled so detection is made even when a rule's pattern match across TCP packet boundaries.) Existing NF-specific programming frameworks for FPGAs [30, 36] do not provide the necessary abstractions for searching bytestreams, nor do they scale to the necessary scale and efficiency to meet our goals. A practical system needs to be able to track at least 100K flows and check at least 10K patterns at 100Gbps line rates, all the while staying within the available processing and memory resources of modern SmartNIC FPGAs. To meet these objectives, our design makes two key contributions:

Hierarchical Pattern Matching: Traditional streaming string search algorithms use NFA-based (state machine) algorithms. While these algorithms are very fast with small rulesets, we measured that supporting the Snort Registered Ruleset [6]

would require 23MB of Block RAM (BRAM), more than the entire capacity of our FPGA (16 MB). We instead take inspiration from Hyperscan [47], designed for x86 processors, which uses hash table lookups instead of a traditional state machine approach for exact-match string search. The (software) pattern matcher in Snort 3.0, which uses Hyperscan, offers a better starting point for our hardware design: it can support all 10K rules using 785KB of memory at a rate of 3.2Gbps on our FPGA. Scaling this to 100Gbps, however, requires replicating the state 32 times over (once again overflowing memory); Pigasus uses a set of *hierarchical filters* to reduce the overall amount of memory required per pipeline replica, enabling search over 32 indices in parallel while consuming only about 2MB of BRAM. Because Pigasus' pattern matcher is so memory-efficient, Pigasus additionally checks for extra strings in the pattern matcher that Snort would push to the 'full match' stage (implemented in Pigasus on the CPU). At the additional cost of 1.3MB of memory, scanning for extra strings results in 2× fewer packets (and 4× fewer rules/packet) reaching the full matcher than Snort.

Fast Common-Case Reassembly: Conventional approaches to TCP reassembly use fixed-length, statically allocated buffers. This prevents highly out-of-order flows from hindering performance (since insertion is constant time) and from consuming too much memory (since buffer sizes are fixed). However, fixed buffers are very memory inefficient; the strategy proposed in [49] would require 6.4GB of memory to support 100K flows. A linked list would be more memory dense, but more vulnerable to out-of-order flows stalling pipeline parallelism or exhausting buffer space. Pigasus adopts the memory-dense linked list approach, but only on an out-of-order *slow path* which runs in parallel to the primary fast path; if the memory buffer approaches capacity, large flows are preferentially reset to prevent overload. On the fast path, packets access a simple table storing the next byte expected and increment it accordingly, thus, in-order flows are performance-wise isolated from out-of-order flows. This allows Pigasus to be memory-efficient (requiring on average 5KB per out-of-order flow) while isolating well-behaved traffic from performance degradation when the IDS is inundated with misbehaving, out-of-order packet data.

Together, the above design choices allow us to do *the majority of processing on a highly-parallelized FPGA fast-path* while fitting *memory within the available resources*. As a result, for the empirical traces, Pigasus can process 100Gbps using an average of 5 cores and 1 FPGA, requiring an average of 85 Watts. In contrast, Snort 3.0 [5] – which uses Hyperscan [47] for string matching – would require 364 cores and 3,278 Watts. Pigasus is publicly available at https://github.com/cmu-snap/pigasus.

2 Background & Motivation

We now introduce software IDS/IPS systems (§2.1) and FPGAs (§2.2). We then analyze IDS/IPS performance and bound the throughput gains achievable via offloading using measurements of Snort 3.0 (§2.3).

2.1 IDS/IPS Functionality

The key goal of a signature-based[1] IDS/IPS system is to identify when a network flow triggers any of up to tens of thousands of signatures, also known as *rules*.

A given signature may specify one or several *patterns* and the entire signature is typically triggered when all patterns are found. Patterns come in the following three categories:

- *Header match*: a filter over the flow 5-tuple (*e.g.*, 'all traffic on port 80', 'traffic from 145.24.78.0/24');
- *String match*: an exact match string to detect within the TCP bytestream or within a single UDP packet;
- *Regular expression*: a regular expression to detect within the TCP bytestream or within a single UDP packet.

Signatures are detected at the granularity of a 'Protocol Data Unit' (PDU) – that is, a signature is only triggered if all matches are found within the same PDU (not over the course of the entire flow). By default, a PDU consists of one packet, but it is possible to define other protocol-specific PDUs spanning multiple packets (*e.g.*, one HTTP GET request).

When an IDS/IPS operates in *detection* mode, a triggered signature results in an alert or an event recorded to a log. When an IDS/IPS operates in *prevention* mode, a triggered signature may raise alerts, record events, or *block* traffic from the offending flow or source. IPSes hence must operate inline over traffic and are latency sensitive – a packet may not be released to the network until after the IPS has completed scanning it. IDSes, on the other hand, may operate asynchronously and are often deployed over a secondary traffic 'tap' which provides copies of the active traffic.

Software IDS/IPS Performance: One of the most widely-known IDS/IPSes is Snort [38] and our work aims to be compatible with Snort rulesets. In our experiments, we primarily work with the Snort Registered Ruleset, which contains roughly 10,000 signatures [6]. This ruleset, combined with conversations with system administrators, sets our goal of supporting 10K rules. In addition, we target 100Gbps as the state-of-the-art line rate [14] and we aim to support 100K flows. To the best of our knowledge there exists no measurement study detailing how many flows to expect at 100Gbps so we derive our 100K flow goal by extrapolating a two-orders-of-magnitude growth factor from a 2010 study [11].

In 2019, Intel published Hyperscan [47], an x86-optimized library for performing both exact-match and regular-expression string matching. Hyperscan is the key new element

[1] There exist other models of IDS/IPS which are 'script based' – executing arbitrary user code over scanned traffic – such as Zeek [7, 35]. These IDS/IPSes are out of scope for this work.

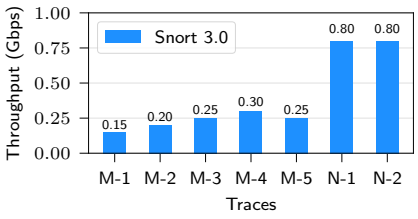


Figure 1: Single-core, zero loss throughput for Snort 3.0 over empirical traces.

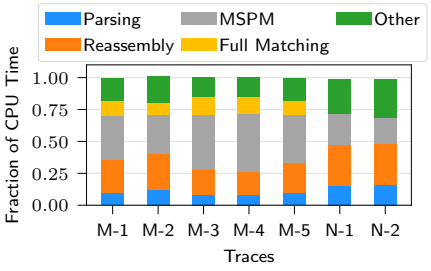


Figure 2: Fraction of CPU time spent performing each task in Snort 3.0.

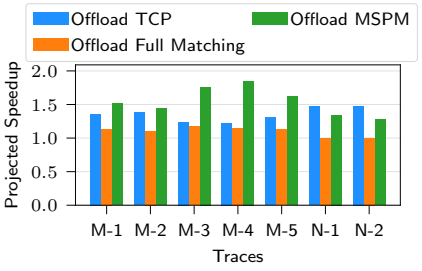


Figure 3: Projected speedup in software given various single-task offloads.

in *Snort 3.0*, which is 8× faster than its predecessor. Nonetheless, we find that Snort 3.0 cannot meet our goal of supporting 100Gbps, 100K flows, and 10K rules on a single server.

We ran Snort 3.0 on a 3.6GHz server and measured the *single-core throughput* over 7 publicly available network traces (described more thoroughly in §6.1). We plot the results in Figure 1. Generously assuming that Snort 3.0 is capable of *perfect* multicore scalability, this would require 125-667 cores to support 100Gbps of throughput, or 4-21 servers.

2.2 FPGA Basics

Why look to FPGAs to improve IDS/IPSes? While there are many platforms ('accelerators') that offer highly parallel processing, we choose FPGAs because they are (a) energy-efficient (using 4-5× fewer Watts than GPUs [12]) and (b) because they are conveniently deployed on SmartNICs where they are poised to operate on traffic without PCIe latency or bandwidth overheads.

FPGA Compute: FPGAs allow programmers to specify custom circuits using code. However, implemented naively, FPGA-based designs can be much slower than their CPU counterparts because FPGA clock rates operate 5-20× slower than traditional processor clock rates. To achieve performance speedups relative to CPUs, circuits must be designed with a high degree of parallelism. FPGAs achieve parallelism either through *pipeline parallelism*, in which *different* modules operate simultaneously over different data, or through *data parallelism* in which copies of the *same* module are cloned to operate simultaneously over different data.

FPGA Memory: Today's FPGAs offer programmers a collection of memory options. Block RAM (BRAM) is the 'king' of FPGA memory because read requests receive a response within one cycle. Furthermore, BRAM is very friendly to parallelism. Divided into 20Kb *blocks* with two ports each, it is possible to read from all BRAM blocks in parallel (and each BRAM block twice) per cycle. When a developer wishes to issue more than two parallel reads to a BRAM block per cycle, they may choose to *replicate* the block to allow more simultaneous read-only accesses to stored data. Our FPGA offers 16MB of BRAM.

Our FPGA also offers 8GB of on-board DRAM (which takes about 20 cycles between read request and response) and 10MB of eSRAM (which takes fixed 12 cycles between read request and response). Because of the multi-cycle latency for these two classes of memory, they are not suitable for storing data that must be read/written every cycle. Furthermore, both are more bandwidth-limited than BRAM and offer fewer lookups in parallel. However, as we will discuss in §4.2, pushing what data is feasible into these classes of memory is necessary to free up as much BRAM as possible to support fast-path memory-intensive processing.

2.3 FPGAs and IDS/IPS Performance

We are not the first to integrate FPGAs into IDS/IPS processing. However, prior work follows an 'offload' approach to utilizing the FPGA, where the CPU is 'primary' and performs the majority of processing, and the FPGA accelerates a single task [10, 16, 18, 19, 22, 34, 41–43, 46, 48, 49]. Most research in this space targets offloading regular expression matching alone [22, 41, 48], although some target TCP reassembly [42, 49] or header matching [27] instead. Unfortunately, a basic analysis based on Amdahl's law reveals why this approach fundamentally cannot bring IDS/IPS performance onto a single server at 100Gbps.

In Figure 2 we illustrate the *fraction of CPU time* spent on each task in Snort 3.0: *MSPM* (which, in Snort 3.0, implements header and partial string matching), *Full Matching* (which, in Snort, implements regular expressions and additional string matching), TCP *Reassembly*, and *other tasks*. As we can see, no single task dominates CPU time – at most, the MSPM consumes 46% of CPU time for one trace.

Using Amdahl's Law, we can see that even if MSPM were offloaded to an imaginary, *infinitely-fast* accelerator, throughput would increase by only 85% to 600Mbps/core, still requiring 166 cores to reach 100Gbps. In Figure 3, we show the idealized 'speedup factor' from offloading any individual module (assuming an infinite accelerator) for each of our traces; no module even reaches as much of a speedup as 2×.

The key lesson is simple: a much more drastic approach is needed to achieve line-rate throughput on a single server.

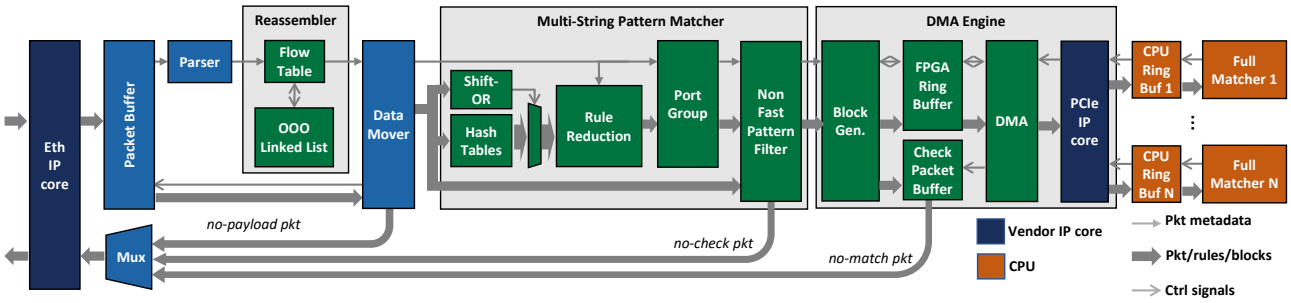


Figure 4: Pigasus architecture.

3 System Overview

We now present an overview of Pigasus. Recall that our target is to achieve 100Gbps, for 100K concurrent flows, and 10K rules within the footprint of a single server. We first describe our rationale behind Pigasus' *FPGA-first* approach (§3.1) before presenting Pigasus' packet processing datapath module by module (§3.2) and discussing how Pigasus makes best use of memory resources (§3.3). Finally, we lay out the threat model that we consider (§3.4).

3.1 An FPGA-First Design

Following our analysis in §2.3, we argue for an *FPGA-first* design for IDS/IPS processing. By FPGA-first, we mean that the FPGA is the *primary* compute platform – performing the majority of work – and that the CPU is *secondary*, operating only as needed. Following our analysis in §2.3, any approach to speed up IDS/IPSes by orders of magnitude must take on parallelizing *as much of the system as possible.*

We can even consider an extreme design, running the *entire system* on the FPGA, disposing the need for CPUs. However, we avoid this approach and choose instead to leave regular expressions and the 'full match' stage on traditional processors. The reason is simple – compared to the other packet processing modules, implementing the Full Match stage entirely on the FPGA provides lower performance benefits at a higher cost in terms of memory. As we will see in §5, the Full Match stage only interacts with ≈5% of packets in the Pigasus design. Hence, it is *not* a performance bottleneck for the majority of packets. Furthermore, regular expression parsing is a very mature research and yet state-of-the-art hardware algorithms do not reach our performance and memory demands for Pigasus. We estimate that GRAPEFRUIT [37], a state-of-the-art regular expression engine for FPGAs, would require 8MB of BRAM to statically map all the regular expressions from our ruleset on the FPGA, and yet would still only keep up with a few Gbps of traffic. Hence, we would likely need multiple replicas of the GRAPEFRUIT design – at least 24MB of BRAM – to keep up with the average of 5Gbps of traffic that reach the full matcher. Therefore, offloading regular expressions would exhaust our memory budget for little gain, in that *the majority of packets will never execute the full matcher anyway*.

3.2 Pigasus Datapath

Figure 4 depicts the major components of Pigasus' architecture. Notice that the Parser, Reassembler, and Multi-String Pattern Matcher (MSPM) are implemented in the FPGA while the Full Matcher is offloaded to the CPU.

Initial packet processing: Each packet first goes through a 100Gbps *Ethernet Core* that translates electric signals into raw Ethernet frames. These frames are temporarily stored in the *Packet Buffer*; each frame's header is separately sent to the *Parser* – which extracts TCP/IP header fields as metadata (*e.g.*, sequence numbers, ports) for use by the Reassembler and MSPM – and then forwards the header to the Reassembler.

Reassembler: The Reassembler sorts TCP packets in order so that they can be searched contiguously (*i.e.*, to identify matches that span across multiple packets). The Reassembler is able to record the last few bytes of the packet's predecessor in that flow in order to enable cross-packet search in the MSPM. UDP packets are forwarded through the Reassembler without processing. The key challenge in designing the Reassembler is doing so *at line rate* with state for *100K flows*; we discuss the design of the Reassembler in §4.

Data Mover: While the Parser and Reassembler operate on headers and metadata alone, the MSPM operates on full packet payloads. The Data Mover receives the (sorted) packet metadata from Reassembler and issues requests to fetch raw packets from the *Packet Buffer* so that they can be forwarded to the MSPM.

Multi-String Pattern Matcher: The MSPM is responsible for (a) checking every packet against the header match for all 10,000 rules, and (b) *every index of every packet* against all of the string-match filters for all 10,000 rules. Pigasus' MSPM does more work than Snort 3.0's equivalent module: Snort 3.0 searches for only *one* exact match string (called the fast pattern) in the MSPM, and pushes detection of the remainder of the strings to the 'full match' module. By searching for all of the exact match strings in the MSPM, Pigasus reduces the number of packets sent to the Full Match stage by more than 2× relative to Snort 3.0, and also reduces the number of suspected rules for the full matcher to check per packet by 4×. We describe the Pigasus MSPM design in §5.

DMA Engine: For each packet, the MSPM outputs the set of *rule IDs* that the packet partially matched. If the MSPM outputs the empty set, the packet is released to the network; otherwise it is forwarded to the DMA Engine which transfers the packet to the CPU for Full Matching. To save CPU cycles, the DMA Engine keeps a copy of the packets sent to the Full Matcher; this allows the Full Matcher to reply with a (packet ID, decision) tuple as a response rather than copying the entire packet back over PCIe after processing. The DMA Engine distributes tasks across cores in a round-robin fashion.

Full Matcher: On the software side, the Full Matcher polls a ring buffer which is populated by the DMA Engine. Each packet carries metadata including the *rule IDs* that the MSPM determined to be a partial match. For each rule ID, the Full Matcher retrieves the complete rule (including regular expressions) and checks for a *full match*. It then writes its decision (forward or drop) to a transmission ring buffer which is polled by the DMA Engine on the FPGA side. If the decision is to forward, the DMA Engine forwards the packet to the network; otherwise the packet is simply erased from the DMA Engine's Check Packet Buffer.

3.3 Memory Resource Management

The core obstacle to realizing an FPGA-first design is fitting all of the above functionality (except the full matcher) within the limited memory on the FPGA. As discussed in §2.2, BRAM is the 'best' of the available memory: it is the only class of memory that can perform read operations in one cycle, and it is also the most parallel. However, it is limited to only 16MB even on our high-end Intel Stratix 10 MX FPGA.

Therefore, we reserve BRAM only for modules which read or write to memory the most frequently, with multiple accesses per packet; namely the Reassembler and the Multi-String Pattern Matcher. Specifically, the Reassembler performs multiple accesses per packet as it needs to check for out-of-orderness, manage the out-of-order packet buffer, and check and release packet headers when an out-of-order 'hole' is filled. The MSPM too entails multiple memory accesses per packet, as every index of every packet must be checked against 10K exact-match string rules, and every packet header must be checked against the header matches for 10K rules.

To save BRAM, we allocate the other stateful modules such as the Packet Buffer and DMA Engine to less powerful eSRAM and DRAM respectively.[2] eSRAM and DRAM turn out to be sufficient for these tasks because the Packet Buffer and DMA Engine have much less stringent demands in terms of bandwidth and latency. In the case of the packet buffer, packet data is written and read only once and hence bandwidth demand is low but still exceeds DRAM's peak throughput; the data mover prefetches each packets 12 cycles before pushing it to the MSPM, keeping throughput high with a negligible latency overhead. The DMA Engine uses DRAM – which has the highest and variable latency and the lowest bandwidth – for the Check Packet Buffer. Since on average only 5% of packets require Full Matching functionality, this places little stress on DRAM bandwidth; the latency overhead of DRAM, while high when compared to BRAM, is still 10× faster than the PCIe latency suffered by packets sent to the CPU for full match.

Even though this leaves us with almost[3] the full capacity of BRAM for the Reassembler and Multi-String Pattern Matcher, realizing these modules is challenging. For instance, using traditional NFA-based search algorithms for the MSPM, given our public ruleset, would require 23MB – more than our 16MB BRAM capacity. Similarly, statically allocating 10KB of out-of-order buffer (*i.e.*, 10 packets) per flow for even 10K flows easily exceeds 100MB. Thus, in §4 and §5, we describe our design optimizations to ensure that the Reassembler and MSPM both 'fit' on-board without compromising performance.

3.4 Threat Model

Pigasus sits between an attacker and its intended target; an attacker may attempt to target Pigasus itself in order to indirectly damage the services Pigasus protects. We assume the attacker does not have physical access to the server running Pigasus, that the operating system and configuration tools for Pigasus are secure, and that the attacker cannot modify Pigasus' configuration. The attacker *may* inject arbitrary traffic into the input stream which Pigasus processes. In this context, we consider two classes of attacks. First, an attacker may attempt to *bypass* Pigasus – that is, send traffic which matches an IDS/IPS rule, but somehow 'trick' Pigasus into allowing it through – *e.g.*, by reording packets, sending duplicate packets, etc. Snort 3.0 employs numerous approaches to address these types of attacks (*e.g.*, timeouts and memory limits) which are straightforward for Pigasus to replicate. Second, an attacker may attempt to *slow down* Pigasus – sending out-of-order or very small packets to reduce Pigasus' effective throughput and hopefully stall or drop innocent traffic. These classes of attacks can always be overcome by 'scaling on demand' – *i.e.*, running more instances – but we set an additional goal of Pigasus being *at least as robust* as Snort 3.0.

4 Reassembly

Reassembly refers to the process of reconstructing a TCP bytestream in the presence of packet fragmentation, loss, and out-of-order delivery. Reassembly is necessary in Pigasus because the MSPM and Full Matcher must detect patterns (strings or regular expressions) that may span across more than one packet (*e.g.*, searching for the word 'attack' should

[2] FPGA manufacturers have been experimenting with varied classes of memory on-board the FPGA over the past few years. From the manufacturers' perspective, Pigasus can be seen as a success story for how varied memory enables more diverse applications which tailor their memory usage to per-task and data structure demands.

[3] We do use BRAM in some other places for internal buffers/queues.

not fail just because 'att' appears at the end of packet n and 'ack' appears at the beginning of packet $n+1$). Note that our goal is not a full TCP endpoint and hence we are not responsible, *e.g.*, for producing ACKs; the IDS/IPS is a passive observer of traffic between two existing endpoints, merely re-ordering the packets it observes for analysis. The key objective of our Reassembler is to perform this re-ordering for 100K's of flows, while operating at 100Gbps, *within the memory limitations* of our FPGA.

4.1 Design Space for TCP Reassembly

Hardware design often favors data structures that are *fixed-length*, *constant-time* and generally *deterministic*, and most TCP reassembly designs follow suit. For instance, [49] allocates a fixed 64KB packet buffer in DRAM and uses 7 pairs of pointers to track OOO state for each flow; similarly, [42] maps a fixed-sized 'segment array' in DRAM to track per-flow state. By using static buffers, these designs are guaranteed constant-time insertion of out-of-order packets into memory; furthermore, the memory consumed by any individual flow is fixed so freeing space is also deterministic. In addition, each flow is bounded in its resource consumption and so a highly out-of-order flow cannot take over the available address space, starving other flows.

The problem with these designs is that, by allocating a fixed buffer, they both *waste memory* and *limit out-of-order flows*. For example, allocating 64KB for *each and every flow* [42] would require 6.4GB to support 100K flows – orders of magnitude bigger than our BRAM capacity. Even worse, the vast majority of flows *don't need the space* most of the time because *most packets arrive in order*. On the other hand, flows which do suffer a burst of out-of-order packets (perhaps due to network loss) that exceeds the 64KB capacity cannot be served, even if there is memory available.

For software developers, the obvious response to these challenges is to use a more memory-dense data-structure such as a linked-list, where each arriving segment is allocated on-demand and inserted into the list in order. Because memory is allocated on demand, no memory is wasted, and those flows which need more capacity are able to consume more as available. In our empirical traces, 0.3% of packets arrive out of order, with 'holes' in the TCP bytestream typically filled in after 3 packet arrivals from the same flow. In a linked-list based design, this means that on average an out-of-order flow consumes 5K bytes at most.

From a hardware perspective, however, a linked list is an unorthodox choice: pipeline parallelism depends on each stage of the pipeline taking a fixed amount of time. Since linked lists have variable insertion times, depending upon how far into the list a segment must be inserted, linked lists can lead to pipeline stalls which result in non-work-conserving behavior upstream from the slow pipeline stage, and hence overall poor throughput. We find that by carefully designing the reassembly pipeline as a combination of a *fast path* (only handling in-order flows) and a *slow path* (that handles the remaining out-of-order flows), one can achieve the best of both worlds.

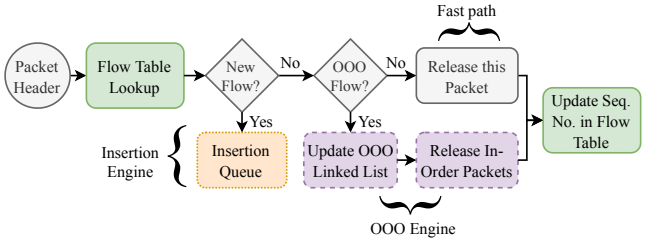


Figure 5: Reassembly Pipeline.

4.2 Pigasus TCP Reassembler

Pigasus takes the linked list approach, targeting a more memory efficient design. However, to avoid pipeline stalls due to variable-time packet insertions, Pigasus uses *three execution engines* to manage reassembly state, each of which handles a different class of incoming packet headers. The *Fast Path* processes in-order packets for established flows; the *Insertion Engine* handles SYN packets for new flows; and the *OOO Engine* handles out-of-order packets for existing flows. Because Pigasus is implemented in hardware, these engines can all run simultaneously (on different packet headers) without stalling each other, but must be careful not to conflict in accessing shared state in the Reassembly *Flow Table*. The flowchart in Figure 5 describes the sequence of steps that occur when a packet header arrives at the Reassembler.

The Flow Table, in representation, is a hash table mapping the classic flow 5-tuple identifier to a table containing (a) the *next expected sequence number* for an in-order packet, and (b) the *head* node for a linked list containing the headers of *out-of-order* packets waiting for a 'hole' in the TCP sequence number space to be filled. We discuss how the Flow Table is implemented in §4.3.

Fast Path: Upon arrival from the parser, each packet header is picked up by the Fast Path which looks up the flow's entry in the Flow Table. If no entry exists for that flow, the Fast Path pushes the packet header on to a queue for the Insertion Engine and moves on to the next packet. If there exists an entry for that flow, but (a) the packet header does not match the next expected sequence number in the Flow Table, or (b) the *head* node in the flow table is not null, the Fast Path pushes the packet header on to a queue for the OOO Engine. Finally, if the packet header *does* match the next expected sequence number in the flow, the Fast Path updates the expected sequence number in the Flow Table to the sequence number for the subsequent packet in the flow and pushes the current packet out towards the MSPM. Every task on the Fast Path runs in constant time, and so throughput is guaranteed through this engine to be 25 Million packets-per-second, which amounts to at least 100Gbps so long the average packet size is greater than 500B (Internet traces typically have an average packet

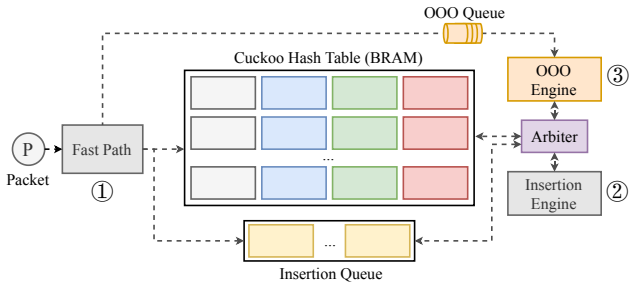


Figure 6: Flow Table and OOO Engine.

size of more than 800B [15]).

OOO Engine: The OOO Engine does not run in constant time, instead dequeueing packets provided for it from the Fast Path as it finishes operating over the previous packet.[4] For each dequeued packet, the OOO engine allocates a new node representing the packet's starting and ending sequence numbers, traverses the linked list for that flow, and inserts the newly-allocated node at the appropriate location. If the packet fills the first sequence number 'hole' in the linked list, then the OOO Engine removes the now-in-order packet headers from the list, releases them to the MSPM, and also updates the Flow Table entry with the new linked list head and next expected sequence number. If the OOO Engine detects that BRAM capacity for OOO flows exceeds 90% of its maximum capacity, it drops the flow with the longest linked list.

Insertion Engine: The Insertion Engine inserts new flow entries into the Flow Table; like the OOO Engine this path too can take variable time. We discuss the insertion engine in more detail in the next subsection.

Overall, allocating memory on-demand avoids memory wastage, and enables Pigasus to better serve OOO flows that do have a higher memory requirement. Additionally, bifurcating the reassembly pipeline into *fast* and *slow* paths prevents out-of-order flows – which require non-deterministic amounts of time to be served in our design – from impacting the performance of in-order flows, which represent the common case.

4.3 Implementing the Flow Table

While the Fast Path, Insertion Engine, and OOO Engine all operate simultaneously, they must synchronize over shared flow state (for instance, to keep the next expected sequence number for each flow consistent). We briefly discuss the implementation of our Flow Table that provides fast and safe concurrent access to these three engines.

The flow table design borrows a key data-structure from FlowBlaze [36]: an FPGA-based hash table that employs deamortized cuckoo hashing [8, 28]. We illustrate this data structure in Figure 6. The design provides high memory density (up to 97% occupancy using 4 or more sub-tables [8, 28, 36]), and worst-case constant-time reads, writes, and deletions for existing entries. It also guarantees that, for an Insertion Queue whose size is logarithmic in the number of flow table entries (in practice, a small value), the queue will not overflow [8]. We implement the hash table using dual-port BRAM, and the Insertion Queue using a parallel shift-register (capable of storing 8 elements).

The key to maintaining the hash table's deamortization property is the Insertion Engine, which is responsible for inserting: (a) new flows, and (b) flows that were previously evicted from the hash table during a 'cuckoo' step. Effectively, the Insertion Engine dequeues an element from the front of the Insertion Queue, and attempts to insert it into the hash table. If at least one of the 4 corresponding hash table entries is unoccupied, it simply updates the flow table and proceeds to the next queued element; otherwise, it *evicts* one of the 4 flow table entries at random, pushes the evicted entry onto the queue, and inserts the deqeued element in its place.

To guarantee conflict-free flow table access, we have the following prioritization of operations to the table. First, note that the Fast Path and OOO Engine never conflict over the same entry – the flow is either in order or not. The Insertion Engine *can* conflict with both the Fast Path and OOO Engine, as it may try to 'cuckoo' entries. Hence, we enforce the following priorities: (1) Fast Path > Insertion Engine (to ensure deterministic performance on the Fast Path), and (2) Insertion Engine > OOO Engine (to ensure that the queue drains and since, empirically, the OOO path is underutilized). Since our BRAM is dual-ported, we allow the Fast Path direct access to the Flow Table, while accesses originating from the OOO Engine or Insertion Engine are managed by an arbiter that enforces the aforementioned priority scheme.

4.4 Worst-Case Performance

Since Pigasus serves on the front-lines of network defenses, it is a prime target for Denial-of-Service (DoS) attacks. Most of Pigasus' underlying components are, by design, fully pipelined, enabling packet data to 'stream through' without ever stalling the system. However, this is not the case for the OOO path in the TCP reassembler. While all *in-order* packets are guaranteed full throughput, an attacker could potentially slow down the OOO path by injecting out-of-order flows into the system.

The key question is how this out of order traffic will impact 'normal' or 'innocent' TCP connections. We observe in our traces that 0.3% of packet arrivals from innocent connections arrive out of order, hence 99.7% of innocent traffic will 'stream through' the fast path, unimpacted by slowdowns on the OOO path. But, worst-case slowdowns on the OOO path *can* stall innocent traffic behind lengthy linked-list traversals due to a malicious sender.

Using mathematical models (elided for space), we quantify the performance of our system in terms of the *goodput*, or the

[4] Experimentally, we actually observe that this slow path is mostly idle when running over our traces, as most packets arrive in order or mostly in-order. We artificially stress this path to overload in our evaluation, but doing so requires an extreme rate of packet loss.

packet rate (in Gbps) corresponding to 'innocent' traffic that the system can sustain in steady state. Then, the attacker's objective is to inject adversarial traffic on the ingress link so as to minimize the achieved goodput.

Starting with a 100Gbps ingress link, we characterize the adversarial scenario using two parameters: the fraction of input traffic that is adversarial, a, and the fraction of non-adversarial input traffic that is in-order, t. Table 1 depicts the expected goodput for different values of a and t (using an average packet size of 500B for innocent traffic) according to the model. Ideally, all innocent traffic would traverse the system unhindered, but we see that slowdowns on the OOO path *can* make Pigasus fall short of this goal – especially at high rates of attack traffic injection – but that the OOO path is not entirely stalled and out of order packets do, eventually, make it through the system.

		In-order Traffic% (t)			
		99.7%	99.0%	90%	**Ideal**
Attack Traffic Rate (a)	10Mbps	99.7	99.1	91.5	99.99
	100Mbps	99.6	98.9	90.1	99.9
	1Gbps	98.7	98.0	89.1	99
	10Gbps	89.7	89.1	81.0	90

Table 1: Total Goodput (in Gbps) for various combinations of the attack traffic rate and fraction of in-order traffic. In-order traffic is isolated from slowdown, even when an adversary introduces substantial out-of-order flows.

5 Multi-String Pattern Matching

Checking tens of thousands of string patterns against a 100 Gbps bytestream makes the multi-string pattern matcher (MSPM) module by far the most operation-intensive and performance critical component in Pigasus.

Role of MSPM in IDS/IPS: As explained in Section 2, a Snort signature/rule comprises three classes of patterns: a *header match*, a set of *exact match strings*, and a set of *regular expressions*. A packet triggers the rule iff *all* patterns are identified.

To avoid checking every single pattern for every index and every packet, rulesets are designed for a two-step matching process. In Snort, the MSPM is responsible for checking *header matches* and *one, highly-selective exact match string*, called the *fast pattern*. Only packets which both match the header match and the fast pattern are forwarded to the *full matcher* which checks regular expressions and any secondary exact match strings (referred to as non-fast pattern strings). Pigasus' MSPM checks for fast patterns, headers, *and* non-fast patterns, reducing the load on the CPU-side full matcher.

MSPM Design Landscape: To the best of our knowledge, there are other no hardware or software projects reporting multi-string matching of tens thousands of strings at 100 Gbps. Classically implemented with parallel NFAs, the best hardware-based string matcher that we know of [16] would require 23MB of BRAM to represent the exact match search strings alone (ignoring the additional header matches).

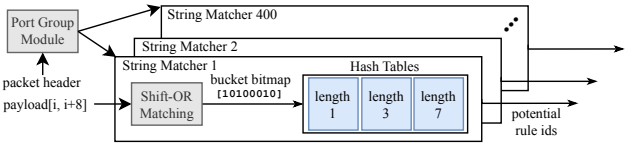


Figure 7: MSPM in Snort 3.0. Every String Matcher selected by the Port Group Module is evaluated sequentially.

To attain a more efficient design, we instead look to *software* and Intel's Hyperscan algorithm for string matching, which is AVX parallelizable and provided an 8× speedup compared to state-machine based string matchers in software [47]. Although naïvely re-implementing Hyperscan on the FPGA is in fact more memory intensive than the NFA approach (requiring 25MB to sustain 100Gbps), we find that by re-architecting Hyperscan's hash table-based design, we can reduce this memory footprint to only 2MB, leaving memory to spare and *expand* on the Hyperscan approach to search for *all* strings (rather than a fast-pattern only) for a total memory budget of 3.3MB. The key idea is to arrange hash table filters hierarchically, with *low memory* filters placed early in the pipeline with a *high replication factor*; this filters out a majority of traffic early. Subsequent stages of the MSPM may be more memory-intensive, per-module, but each stage handles less and less traffic and hence requires less replication.

In what follows, we first describe Hyperscan's two-stage MSPM, which checks for header matches and fast patterns. We then describe Pigasus' three-stage MSPM, which checks for fast patterns, header matches, and non-fast pattern strings using highly parallel, hierarchical filters to improve memory density.

5.1 MSPM in Software

Snort 3.0 + Hyperscan: In Snort 3.0, the MSPM is implemented using Intel's Hyperscan, illustrated in Figure 7.

Packets are first checked for their *header match*. Across all 10K rules, there are only ≈400 *unique* header match values. Rules which share the same header match fields are said to belong to the same *port group*. The port group module outputs a set of *port group IDs* which the packet matches; this output set is never empty because some rules wildcard their header match and hence match all packets. An average packet matches 2 port groups.

Packets are then checked for their *fast pattern string match*. For each port group, there exists a *string matcher* which checks fast patterns for all rules within that port group. Snort must check every string matcher for each port group the packet matches.

Within the string matcher, Snort must iterate over every index of the payload checking whether it matches any of the fast patterns in the port group. Rather than using a state machine

to do this, Hyperscan uses a collection of hash tables. For each possible fast pattern length[5] a hash table is instantiated containing the fast patterns of that length. Hyperscan then performs an exact-match lookup for all substrings at each index, looking up whether or not the substring is in the hash table – potentially $(8 \times l)$ lookups for a packet of length l.

To reduce the number of expensive sequential lookups, each string matcher contains a SIMD-optimized *shift-or filter* [9] prior to the hash table; this filter outputs either a '0' or '1' for every byte index of the packet, indicating whether or not that index matches *any* fast pattern in the hash tables; indices which result in a '0' output from the shift-or stage need not be checked.

The string matcher – combining shift-or and hash tables – then outputs a set of *rules* which the packet matched both in terms of *header* and *fast pattern*; together, the packet and the potential rule matches are passed to the full matcher. However, for 89% of packets, this stage outputs the empty set and the packet bypasses the full match stage entirely.

5.2 MSPM in Pigasus

A straightforward port of the Snort 3.0 MSPM engines and data structures onto the FPGA consumes 785KB of memory and forwards at a rate of 3.2Gbps. Taking advantage of the high degree of parallelism offered by the FPGA, one could, in theory, scale to 100Gbps via *data parallelism*, *i.e.*, replicating this 32 times. Unfortunately, doing so would require 25MB of BRAM. We now describe how Pigasus re-architects the Hyperscan algorithm to achieve this high degree of parallelism within available resources. Since this results in leftover memory, we can then *extend* Pigasus' MSPM to scan for non-fast pattern strings as well.

As shown in Figure 8, Pigasus flips the order of Hyperscan's MSPM, starting with string matching before moving on to header matching and port grouping.

Fast Pattern String Matching (FPSM): To perform string matching, Pigasus (like Hyperscan) also has a *filtering* stage in which packets traverse two parallel filters: a shift-or (borrowed from Hyperscan) and a set of per-fast-pattern-length hash tables. We check the shift-or and (32×8) hash tables in parallel. Hash tables only store 1-bit values indicating whether a given (index, length) tuple results in a match – but it does not store the 16-bit rule ID. The output from the filters is ANDed together, reducing *false positives* from either filter alone by 5×.

The shift-or and 1-bit hash table[6] consume only 65KB and 25KB respectively, thus they are relatively *cheap* to replicate 32× over in order to scale to 100Gbps. In theory, these filters can generate (32×8) matches per cycle (*i.e.*, 8 matches per filter); however, in the common case, most packets and most indices *do not match any rules*, and therefore require *no further processing*.[7] This gives us the opportunity to make subsequent pipeline stages narrower. We design a 'Rule Reduction' module that selects non-zero rule matches from the filter's 256-bit wide vector output and narrows it down to 8 values.

Applying this filter first allows us to use fewer replicas of subsequent data structures (which are larger and more expensive), since most bytestream indices have already been filtered out by the string matcher. This enables high (effective) parallelism with a lower memory overhead.

Header Matching: In this stage, we use the packet header data to determine whether the matches produced by the previous (FPSM) stage are consistent with the corresponding rule's Port Group. At this point, we only need to create 8 replicas of the 17KB Rule Table and 68KB Port Grouping modules to check 8 rules simultaneously.

Using the (index, length) tuple that resulted in a match in the FPSM stage, we look up the corresponding rule ID in the Rule Table. Next, using this rule ID, we look up the Port Group that this rule maps to; this could be a *single* port, a *list* or *range* of ports, or a *wildcard* (indicating a match on any port). If this packet's port number is a subset of this rule's Port Group, the rule is considered a match; otherwise, the rule is ignored.

Our initial design of the MSPM stopped here (at the Traffic Manager 1 stage in Figure 8), aiming merely to reproduce Snort's functionality, which only scans for fast patterns and headers. Packets which matched the fast pattern and header on at least one rule were sent to the CPU for processing. While packets which did not produce any matches at either the FPSM or Header Matching stage were simply streamed to the output interface.

This resulted in a design that sent 11% of packets to the CPU for processing, with an average of 4.4 rules searched per packet – and required only *2MB* of memory! Given that this amounted to a fraction of our resource budget for the MSPM, we asked ourselves: can we do more?

Non-Fast Pattern String Matching (NFPSM): Pigasus further filters down the packets and rules destined for the CPU to only 5% of packets, with just 1.1 rules/packet (on average), by additionally searching for *all* string matches within a rule on the board. Note that, on average, only 11% of packets reach the Non-Fast Pattern Matcher, and, by this point, we know which rules (on average 4.4 of them) the packet might match on. Naïvely, one might iteratively search for each string in the ≈ 4.4 rules, but because each packet has a variable number of rules and each rule has a variable number of strings (between 1 and 32), this approach would likely lead to low throughput and/or pipeline stalls.

Instead, Pigasus once again uses a set of hash tables (like in the FPSM) to search for all strings simultaneously. It then

[5] Up to 8 bytes – longer fast patterns are truncated.

[6] Subtly, this is not a true Bloom filter [13] because we only perform one hash per input; implementing multiple hashes increases resource utilization and complexity, we find, with little gain.

[7] Note that our filters never produce false negatives.

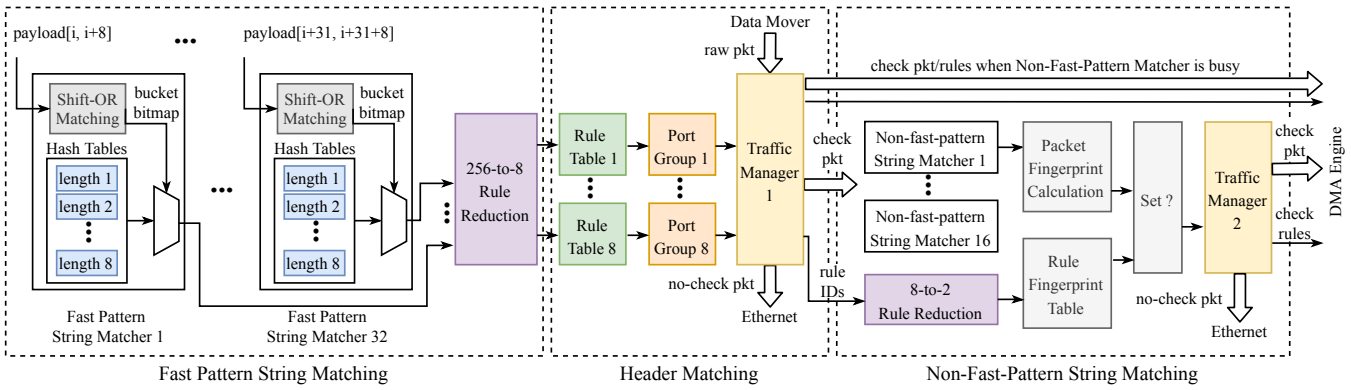


Figure 8: Pigasus' MSPM, which requires a total of 3.3MB of BRAM.

creates a compact, bloom-filter-like representation ('fingerprint') of the matched strings. To compute the fingerprint, we first represent the set of (index, length) tuples generated by the 8 NFPSM hash tables as a 16-bit vector by setting $bit[index \pmod{16}]$ to '1' for each length bucket. Next, for each bucket, all of the 16-bit vectors generated for a given packet are ORed together to create a 16-bit 'sub-fingerprint' for that bucket. Finally, these sub-fingerprints are concatenated into a 128-bit fingerprint representing the entire packet. The fingerprinting process is illustrated below:

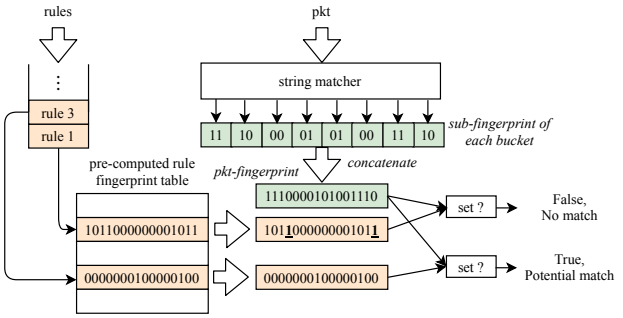


Figure 9: Rule matching fingerprints in the NFPSM.

The NFPSM can now look up a corresponding fingerprint – generated in the same way – for each of the ≈ 4.4 rules, and now can do a parallel set comparison between the two fingerprints. If, for every bit in the *rule's fingerprint*, the corresponding bit in the *packet's fingerprint* is also set, there is high probability that all of the exact match strings for the rule were matched. But, if any of the corresponding bits are *not* set, we can be certain that at least one of the non-fast pattern strings were not matched, thus eliminating the rule as a potential match. The 5% of packets which match at least one rule fingerprint are forwarded to the CPU; the remainder are released as non-matching and therefore innocent packets.

It is worth noting that, as the last stage of our hierarchical filtering, the non-fast pattern matcher has the lowest throughput capacity. This saves on resources, but can make the NFPSM vulnerable to overload. Where the fast pattern matcher is designed to process up to 100Gbps of incoming data, the non-fast pattern matcher tops out at a peak throughput of 50Gbps. 50Gbps is more than enough to handle the average rate of 11Gbps, but a spike of rule-matching malicious traffic can at times overload the non-fast pattern matcher. In this case, when the first traffic manager (between the header matcher and the non-fast pattern matcher) detects *backpressure* from the NFSM, it steers some packets directly to the CPU for checking. Temporarily increasing the load on software, where it is easier to 'scale out' and provision additional resources.

End-to-End: By hierarchically filtering out packets, the MSPM reduces the amount of traffic traversing each subsequent stage of the MSPM. This means that the earliest stages require high levels of replication, but the latter stages can, on average expect lower throughput and hence require less replication. Consequently, latter stages require lower memory consumption. End-to-end, the MSPM requires 3.3MB of memory, fitting well within our BRAM bounds while doing *more* filtering than what a naïve port of the Hyperscan algorithm would be capable of. Nonetheless, the reduced capacity of the MSPM in the latter phases of the MSPM does make these components vulnerable to overload; in these cases Pigasus temporarily shunts additional traffic to CPUs, where is easier to provision on demand and as needed.

6 Evaluation

In this section, we evaluate Pigasus and show that:

- Pigasus is at least an order of magnitude more efficient than state-of-art Snort running in software: using $23-200\times$ fewer cores, and $18-62\times$ less power;
- Pigasus' performance gains are resilient to a variety of factors such as small packets, out-of-order arrivals, and the rule-match profile of the traffic;
- The Pigasus architecture actually has resource headroom, suggesting a roadmap for handling even more complicated workloads.

We start by describing the evaluation setup we use for the rest of the section before the detailed results.

Module	ALM	BRAM (MB)	DSP	eSRAM (MB)
Packet Buffer	507 (0.1%)	0 (0%)	0 (0%)	5.91 (50.0%)
String Matcher	119,562 (17.0%)	3.30 (19.7%)	1,600 (40.4%)	0 (0%)
Flow Reassembler	20,728 (2.9%)	2.61 (15.6%)	0 (0%)	0 (0%)
DMA Engine	2,000 (0.3%)	0.32 (1.9%)	0 (0%)	0 (0%)
Instrumentation	1,189 (0.2%)	0 (0%)	0 (0%)	0 (0%)
Vendor IPs	42,028 (6.0%)	1.22 (7.3%)	0 (0%)	0 (0%)
Miscellaneous	21,946 (3.1%)	0.60 (3.6%)	0 (0%)	0 (0%)
Full Design	207,960 (29.6%)	8.05 (48.1%)	1,600 (40.4%)	5.91 (50.0%)

Table 2: Resource breakdown. Percentages are relative to the total amount of resources in a Stratix 10 MX FPGA.

6.1 Setup

Implementation and Resource Breakdown: We implement Pigasus using an Intel Stratix 10 MX FPGA Development card [2] as the SmartNIC in a 16-core (Intel i9-9960X @ 3.1 GHz) host machine. The Stratix 10 MX FPGA has 16MB of on-chip BRAM, 10MB of eSRAM, and 8GB of off-chip DRAM. Table 2 shows the FPGA resources used by each component of Pigasus when configuring it to support 100K flows and 10K rules. To implement Pigasus' CPU/software components, we adapt Snort 3 to allow it to receive reconstructed PDUs and rule IDs, coming from the FPGA directly into its Full Matcher. We run Snort 3 software experiments in an Intel i7-4790 CPU @ 3.60 GHz.

Traffic Generator: We installed both DPDK Pktgen [1] and Moongen [20] on a separate 4-core (Intel i7-4790 @ 3.6 GHz) machine with a 100Gbps Mellanox ConnectX-5 EN network adapter. DPDK Pktgen achieves higher throughput when replaying PCAP traces – up to 90Gbps – and hence we use the DPDK Packet Generator when running experiments with recorded traces. Moongen is better at generating synthetic traffic at runtime and can do so at up to the full 100Gbps offered by the underlying network. We specify in each experiment which traffic generator was used.

Traces and Ruleset: We test Snort and Pigasus both using the publicly available Snort Registered Ruleset (snapshot-29141) [6] and different traces from Stratosphere [44]: *CTU-Mixed-Capture-1–5*, *CTU-Normal-12*, and *CTU-Normal-7*. We refer to them as *mix-1–5*, *norm-1*, and *norm-2*, respectively. For the *mixed* traces, we use the `*before.infection` pcaps. We use Stratosphere traces because their packet captures contain the original payloads, which is essential when evaluating IDSes.

Measuring Throughput and Latency: We measure throughput in two ways: 1. The *Zero Loss* throughput is measured by gradually increasing the packet generator's transmission rate until the system (Snort or Pigasus) first starts dropping packets; 2. The *Average* throughput is computed as the ratio of the cumulative size of packets in the trace (in bits) to the total time required to process the trace. We measure latency (at low load) using DPDK Pktgen's built-in latency measurement routine. Unfortunately, DPDK embeds timestamps in the packet body, which never triggers the CPU-side Full Matching functionality. Instead, we measure the end-to-end latency for Pigasus on an empirical trace using FPGA-side counters, and then adding the baseline FPGA loopback latency to it.

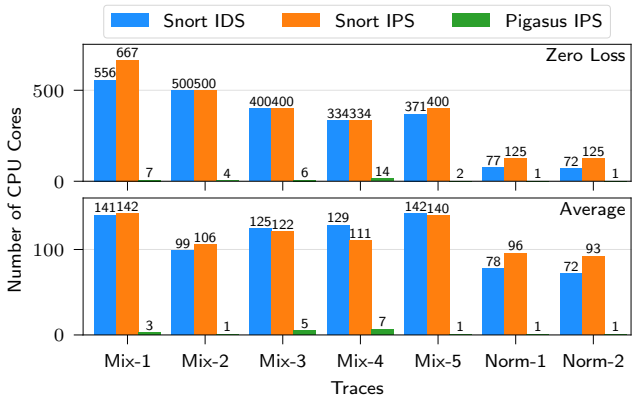


Figure 10: Number of cores required to process each trace at 100Gbps using Pigasus (FPGA + CPUs) and Snort (traditional CPUs alone). Pigasus numbers are based on implementation; Snort numbers are extrapolated from its single-core throughput and assume perfect linear scaling.

6.2 End-to-end performance and costs

In this section, we compare the performance, power, and cost of Pigasus vs. legacy Snort.

Provisioning for 100Gbps throughput: Figure 10 reports the number of server cores required to achieve 100Gbps for the evaluated Stratosphere traces for different settings. The top half is under the assumption of loss-free processing without buffering, while the bottom reports the steady-state core requirements based on the assumption that we could buffer packets during the peak periods and defer the full matching to allow the cores to catch up after the peak has passed.

The Pigasus results are based on experiments where the system is tested at increasing number of cores at maximum throughput, until we observe no packet loss. For the Snort experiments we run Snort in both IDS and IPS mode (with DPDK) on a single core and increase the throughput until it begins to drop packets. Note that while we report the actual number of cores required to run Pigasus, for Snort we extrapolate the single-core experiment to determine the number of cores that we would need to keep up with 100Gbps. This considers that Snort's throughput scales linearly with the number of cores and, therefore, represents an ideal *lower bound* to the actual number of cores needed to run Snort. Overall, we see that Snort in IDS mode requires $23-185\times$ more cores than Pigasus ($65\times$ on average), and in IPS mode requires $23-200\times$ more cores ($72\times$ on average).

Latency: Of course, in a practical IPS we care not only about throughput/provisioning but also per-packet latency. We plot the distribution of per-packet latency in Figure 11. We find

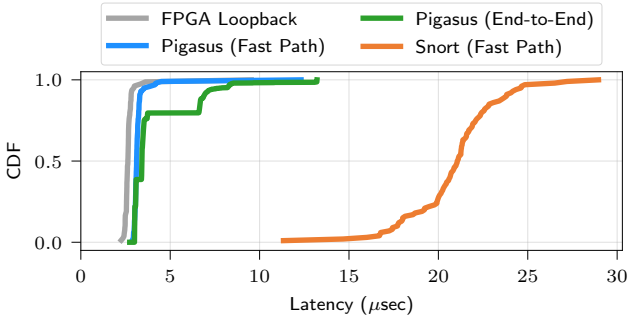


Figure 11: CDF of latency of Pigasus vs. Snort.

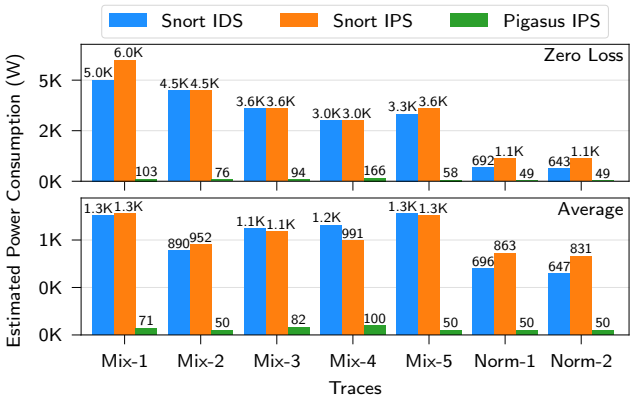


Figure 12: Estimated wattage to achieve 100Gbps.

that Pigasus yields almost an order of magnitude improvement in the median latency, and up to 3× improvement in the tail latency. As a point of comparison, we also show the baseline performance of a simple FPGA loopback measurement (*i.e.*, without any processing) and the Pigasus fast-path for packets that do not need further CPU processing. We find that the Pigasus fast path is very efficient and almost comparable to the baseline. We also find that Pigagus end-to-end latency only deviates substantially from the fast path for the tail. While we hypothesized some improvements in latency, we were puzzled by the magnitude of the improvement. Investigating why Snort was much slower revealed that on average, while Pigasus reduced the latency for the Reassembly (by 6μs), Parser (by 4μs), and the MSPM (by 3μs) as expected relative to software, the additional reduction came from avoiding Packet I/O overhead in software (around 5μs).

Power footprint: Figure 12 depicts the estimated power consumption required to achieve 100Gbps throughput for three configurations: Snort in IDS mode, Snort in IPS mode, and Pigasus in IPS mode. On the CPU side, we use Intel's Running Average Power Limit (RAPL) interface [23] to measure per-core power consumption in steady-state. To verify its accuracy we also measured the power utilization using an electricity usage monitor [4] and found consistent results. On the FPGA side, we use the Board Test System [2] (part of Intel's FPGA Development Kit) to measure power dissipation in the FPGA core and I/O shell. We observe that, across all traces, Snort (in *either* mode) has a $13-59\times$ higher power consumption than Pigasus (34× on average). We further note that the reported wattages for Pigasus represent a conservative estimate; while the total power consumption on the FPGA side is 40W, the core fabric accounts for just 13W, and the remainder is used for I/O (including Ethernet). Conversely, we only charge Snort for power consumed during compute tasks, ignoring other overages (such as Network I/O).

Cost: To estimate the Total Cost of Ownership (TCO), we consider both the capital investment and the power cost for each configuration. To estimate the capital investment, we use the per-core pricing data for the AMD EPYC 7452 CPU ($68.75 per core). For Pigasus, we also incorporate the market price of an Stratix 10 MX FPGA [2] ($10K). Assuming that the number of cores needed in practice is between the *Zero Loss* and *Average* in Figure 10, we estimate that the capital cost of the CPU-only solution is between $7,922 and $25,045, while the capital cost of Pigasus is between $10,189 and $10,344. To estimate the power costs we assume a lifetime of 3 years and electricity cost at $0.1334/kWh (average electricity rate in the US [3]). The power cost of the CPU-only solution at 9W/core is between $3,636 and $11,494, while the cost for Pigasus is between $227 and $298. Then, combining the capital investment and the power cost, the TCO of the CPU-only solution is between $11,558 and $36,539, while the TCO of Pigasus is between $10,416 and $10,642, saving between $1,142 and $25,897. We note that these estimates consider retail prices and do not account for other operational costs, such as cooling and rack space, which we expect to favor Pigasus. Moreover, for 100K flows and 10K rules we only use about half of a Stratix 10 MX; one may consider adapting the design to a smaller FPGA, further reducing the cost of Pigasus.

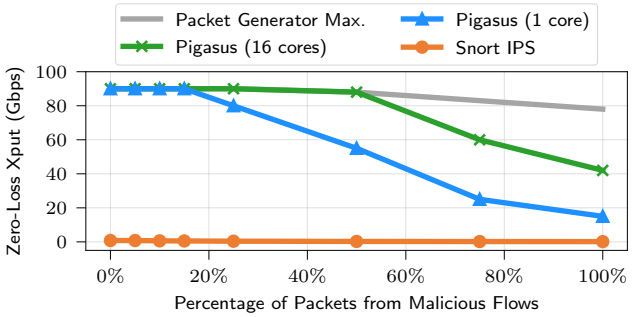
Recall that our original goal was to achieve 100Gbps supporting hundreds of thousands of flows matching tens of thousands of rules on a single server with a reasonable cost/resource footprint. The above results suggest that Pigasus indeed achieves this goal (with ample headroom).

6.3 Microbenchmarks and sensitivity analysis

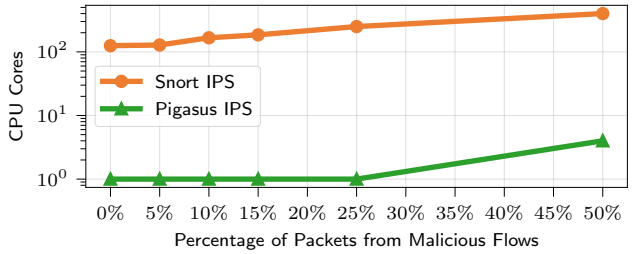
In this section, we present Pigasus' performance sensitivity to traffic characteristics. We probe deeper into Pigasus' performance under differing levels of malicious traffic. We further characterize the performance impact of *packet size*, and *out-of-order degree* of flows.

Dependence on CPU Offload: We construct semi-synthetic traffic traces by mixing malicious flows extracted from *mix-1* trace with innocent trace *norm-2* in different proportions.[8] Figure 13 (a) depicts the dependence of zero-loss throughput on the fraction of malicious flows (in terms of relative packet count). We report results for Pigasus (using both 1 and

[8] Note that not every packet in a malicious flow triggers a match.



(a) Zero-loss throughput.



(b) Number of cores required to achieve 100Gbps.

Figure 13: Impact of the fraction of malicious traffic on system throughput.

16 cores) and Snort IPS (with 1 core). We observe that, as long as the fraction of malicious traffic is smaller than 15%, Pigasus is able to process packets at line-rate using a single CPU core. With 16 cores Pigasus can process packets at line rate for up to 50% of malicious traffic. After the 50% mark, performance begins to degrade gradually. We repeated the same experiments disabling the software component of Pigasus and observed that the througput matches the 16-core experiment, suggesting that the hardware is the bottleneck. More specifically, the MSPM's rule reduction logic is stressed by the large number of potential rule matches.

Figure 13 (b) depicts the number of cores required to achieve 100Gbps as a function of the fraction of packets from malicious flows for up to 50%. Results for Snort are extrapolated from the single-core throughput. Despite the performance degradation observed in (a), Pigasus scales considerably better than Snort, requiring two orders of magnitude fewer cores. We also note that, while the hardware only becomes the bottleneck at an extreme fraction of malicious traffic, the design can be made even more robust using two hardware pipelines (discussed further in §6.4).

Dependence on Packet Size: We first consider the impact of packet size on Pigasus' performance stemming from the linked-list based TCP reassembler design. We configure the Moongen packet-generator to generate fixed-sized synthetic packets, and measure end-to-end, zero-loss throughput as we vary the packet size. Figure 14 illustrates this dependence. We observe that, for packets exceeding 500B (comparable to average packet sizes on the Internet [15]), Pigasus is capable of processing at line rate. (More generally, Pigasus by design

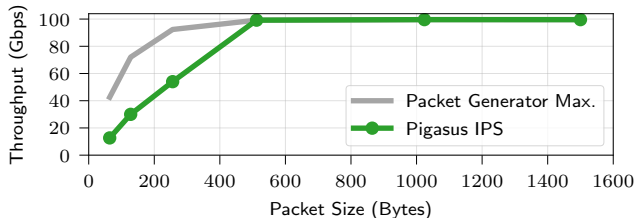


Figure 14: Zero-loss throughput achieved by Pigasus for a range of packet sizes.

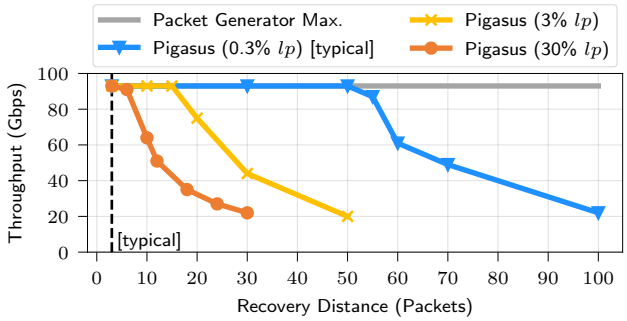


Figure 15: Zero-loss throughput achieved by Pigasus for a range of Loss Probabilities (lp) and Recovery Distances (rd).

can sustain 100Gbps as long as the average packet size is greater than 500B over a window of 87μs estimated base on buffer size.)

Dependence on Out of Order Degree: We characterize the OOO degree using randomly generated synthetic packet traces controlled by two independent variables: the packet loss probability (lp) [32] and the recovery distance (rd).[9] Figure 15 depicts the impact of these parameters on Pigasus' end-to-end, zero-loss throughput. We sweep the loss probability from 0.3% to 30%, and the recovery distance from 3 to 100. At typical values ($lp = 0.3\%$, $rd = 3$), Pigasus achieves a single-core throughput of 100Gbps, which degrades gradually with increasing packet loss and recovery distance. It is worthwhile to note that, at typical packet loss rates, the Reassembler can handle around 50 OOO packet arrivals without any degradation in end-to-end throughput.

6.4 Future outlook

Supporting 100Gbps with 100K flows and 10K rules requires only about half of the resources in our FPGA. We now explore what we can do with the additional capacity.

One option is to duplicate the existing processing pipeline (which runs at 100Gbps/25Mpps) each to serve a different subset of flows, increasing the throughput to 200Gbps, at the cost of creating additional copies of all the MSPM engines. Another option is to increase the number of supported flows

[9] Recovery distance is defined as the number of same-flow packets that arrive before a hole created by a lost packet is filled. In Pigasus, this value determines the amount of work (in cycles) that the OOO Engine must perform for each packet that arrives out of order.

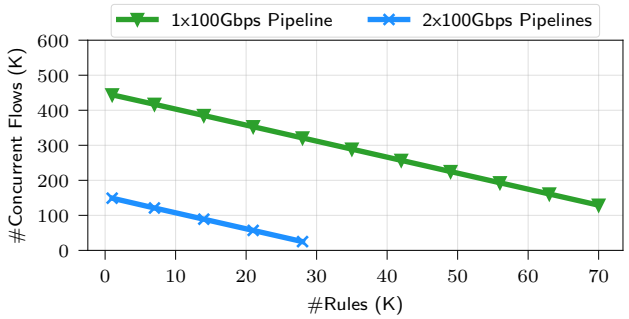


Figure 16: Tradeoff between the number of supported rules and concurrent flows when using one or two 100Gbps hardware pipelines.

or rules. Figure 16 depicts the three-way tradeoff between the scalability of the number of rules, concurrent flows, and replicated hardware pipelines. The design with two pipelines benefit from better throughput but have fewer room for storing rules or flows. There is plenty of scaling headroom in the Pigasus FPGA frontend design for more rules and flows.

7 Related Work

We now review some of the most related work, some of which served as inspiration for Pigasus.

Pattern matching: The design of the hash table filter in our Multi-String Pattern Matcher is similar to the filters used by DFC [17] and Hyperscan [47]. An important difference, however, is that instead of using a second hash table to associate potential string patterns with their identifiers, we directly use the matched index as a pattern identifier. This helps to reduce the amount of resources required by the hardware implementation. We also employ fewer, but much larger filters, since cache-friendliness is not a concern for FPGA design.

Using FPGAs to accelerate IDS/IPS: Many previous work have also made a case for using FPGAs to implement network functionality [21, 30, 33, 36, 45]. ClickNP [30] and FlowBlaze [36] present abstractions for making it easier to implement network functionality in FPGAs. However, they do not provide the necessary abstractions for searching bytestream nor they would be able to scale to meet our goals for throughput and number of rules. Some propose using FPGAs to accelerate IDS/IPS [10, 16, 18, 19, 34, 43, 46, 49]. However, all of these works do not implement a complete IDS/IPS and fail to meet our target for throughput or number of rules. Even though, Snort Offloader [43] proposes using an FPGA to implement an entire IDS/IPS, it only supports very simple operations, not including components that are essential for correct IPS operation, *e.g.*, TCP reassembly.

Other accelerators: Other works have looked at using hardware accelerators to improve some IDS/IPS components. Kargus [25] uses GPUs to accelerate exact-pattern and regular-expression matching. However, their use of GPUs contributes to increasing both power and latency. PPS [26] uses PISA switches to implement DFAs and accelerate arbitrary regular expressions. But are limited to only UDP and can only support a small number of string patterns. More important, however, we note that by only accelerating the latest IDS/IPS stages, these solutions are fundamentally limited in the throughput improvements they can achieve.

8 Conclusions

In many ways, IDS/IPS are one of the most stressful network workloads for both traditional software and hardware. As such, the gap between the workload demands and what was achievable on a single server always seemed elusive. The design of Pigasus is a singular proof point that a seemingly unattainable goal (100Gbps line rates for 100K+ flows matching 10K+ of complex rules) on a single server is well within our grasp. Looking forward, we believe that we can further unleash the potential benefits of FPGAs for this unique workload by further eliminating CPU bottlenecks and potentially moving additional functionality onto the FPGA. Given the future hardware roadmaps of FPGAs and SmartNICs, we believe that our insights and successes can more broadly inform in-network acceleration beyond IDS/IPS as well.

Acknowledgements

We thank the OSDI reviewers, Eriko Nurvitadhi, Aravind Dasu, and our shepherd Thomas Anderson and his students for comments and feedback on this work. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; in part by the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS); and finally in part by the project AIDA - Adaptive, Intelligent and Distributed Assurance Platform (reference POCI-01-0247-FEDER-045907), co-financed by the ERDF - European Regional Development Fund through the Operational Program for Competitiveness and Internationalisation - COMPETE 2020.

References

[1] DPDK-pktgen. https://github.com/Pktgen/Pktgen-DPDK.

[2] Intel Stratix 10 MX. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html.

[3] Microsoft Azure: Total cost of ownership (TCO) calculator. https://azure.microsoft.com/en-us/pricing/tco/calculator/. Accessed: 2020-10-01.

[4] PN1500 Watt meter. https://poniie.com/products/17.

[5] Snort 3. https://www.snort.org/snort3.

[6] Snort ruleset. https://www.snort.org/downloads/#rule-downloads.

[7] Zeek – network security monitor. https://www.zeek.org.

[8] Y. Arbitman, M. Naor, and G. Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Interna-*

tional Colloquium on Automata, Languages, and Programming, pages 107–118. Springer, 2009.

[9] R. Baeza-Yates and G. H. Gonnet. A New Approach to Text Searching. *Commun. ACM*, 35(10):74–82, Oct. 1992.

[10] Z. K. Baker and V. K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, ANCS '05, pages 193–202, New York, NY, USA, 2005. Association for Computing Machinery.

[11] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[12] BERTEN. GPU vs FPGA performance comparison. Technical Report BWP001, BERTEN Digital Signal Processing. http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf.

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[14] M. Branscombe. The year of 100GbE in data center networks. https://www.datacenterknowledge.com/networks/year-100gbe-data-center-networks, Aug. 2018.

[15] CAIDA. Packet length distributions. https://www.caida.org/research/traffic-analysis/AIX/plen_hist/.

[16] M. Ceška, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semric, and T. Vojnar. Deep packet inspection in FPGAs via approximate nondeterministic automata. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '19, pages 109–117, 2019.

[17] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating string pattern matching for network applications. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 551–565, Santa Clara, CA, Mar. 2016. USENIX Association.

[18] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, pages 249–257, 2004.

[19] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, Jan. 2004.

[20] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, Oct. 2015. Association for Computing Machinery.

[21] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.

[22] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. HARE: Hardware accelerator for regular expressions. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, Oct. 2016.

[23] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, Jan. 2012.

[24] Intel. FPGA design software – Intel Quartus Prime. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html. Accessed: 2020-10-14.

[25] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 317–328, New York, NY, USA, 2012. Association for Computing Machinery.

[26] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé. Fast string searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 21–28, New York, NY, USA, 2019. Association for Computing Machinery.

[27] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 171–186, Renton, WA, Apr. 2018. USENIX Association.

[28] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[29] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman. UNO: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, New York, NY, USA, 2017. Association for Computing Machinery.

[30] 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*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[31] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 363–378, Renton, WA, July 2019. USENIX Association.

[32] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

[33] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable router architecture for experimental research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 1–7, New York, NY, USA, 2008. Association for Computing Machinery.

[34] P. Orosz, T. Tóthfalusi, and P. Varga. FPGA-assisted DPI systems: 100 Gbit/s and beyond. *IEEE Communications Surveys & Tutorials*, 21(2):2015–2040, 2019.

[35] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[36] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 531–548, Boston, MA, Feb. 2019. USENIX Association.

[37] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron. Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAs. In *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pages 138–147. IEEE, 2020.

[38] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, USA, 1999. USENIX Association.

[39] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 323–336, San Jose, CA, Apr. 2012. USENIX Association.

[40] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 13–24, New York, NY, USA, Aug. 2012. Association for Computing Machinery.

[41] R. Sidhu and V. Prasanna. Fast regular expression matching using fpgas. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Los Alamitos, CA, USA, Apr. 2001. IEEE Computer Society.

[42] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. Scalable 10gbps TCP/IP stack architecture for reconfigurable hardware. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, pages 36–43, USA, 2015. IEEE Computer Society.

[43] H. Song, T. Sproull, M. Attig, and J. Lockwood. Snort offloader: a reconfigurable hardware NIDS filter. In *International Conference on Field Programmable Logic and Applications*, FPL '05, pages 493–498, 2005.

[44] Stratosphere. Stratosphere laboratory datasets, 2015. Retrieved March 13, 2020, from https://www.stratosphereips.org/datasets-overview.

[45] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference*, ATC '17, pages 459–471, Santa Clara, CA, July 2017. USENIX Association.

[46] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, page 112–122, USA, 2005. IEEE Computer Society.

[47] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 631–648, Boston, MA, Feb. 2019. USENIX Association.

[48] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. REAPR: Reconfigurable engine for automata processing. In *27th International Conference on Field Programmable Logic and Applications*, FPL '17, pages 1–8. IEEE, Sept. 2017.

[49] R. Yuan, Y. Weibing, C. Mingyu, Z. Xiaofang, and F. Jianping. Robust TCP reassembly with a hardware-based solution for backbone traffic. In *Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, NAS '10, pages 439–447, USA, 2010. IEEE Computer Society.

A Artifact Appendix

A.1 Abstract

Pigasus has a hardware component, that runs on an FPGA, and a software component which is adapted from Snort 3. The current version requires a host with a multi-core CPU and an Intel Stratix 10 MX FPGA (with 100 Gb Ethernet) [2]. Pigasus' artifacts are open source and publicly available.

We provide detailed instructions to reproduce Figure 10. This figure supports our main claim that Pigasus requires two-order of magnitude fewer cores than state-of-the-art Snort 3. In addition to the steps in this appendix and on the repository README, we also provide video archives that reproduce Figure 10 for both the Snort 3 Baseline[10] and the Pigasus[11] experiments.

A.2 Artifact check-list

- **Algorithm:** Pigasus Multi-String Pattern Matcher.
- **Program:** Snort 3 [5] for baseline experiments; DPDK pktgen [1] and Moongen [20] to generate packets.
- **Compilation:** Intel Quartus Prime [24].
- **Data set:** Stratosphere Laboratory Datasets [44].
- **Run-time environment:** System running Linux with Snort 3 [5] software dependencies installed. Quartus 19.3 with Stratix 10 device support is required to load the bitstream to the FPGA.
- **Hardware:** Two servers, one with an Intel Stratix 10 MX FPGA [2] and another with a DPDK-compatible 100 Gb NIC. Power-measurement experiments require either a CPU with a power measurement interface (*e.g.*, RAPL [23]) or an external electricity usage monitor.
- **Execution:** Disable power optimizations in the BIOS, isolate cores from the Linux scheduler, and pin processes to cores.
- **Experiments:** Experiments are run manually with Pigasus on one machine and a packet generator on another.
- **Public link:** https://github.com/cmu-snap/pigasus
- **Code licenses:** 'BSD 3-Clause Clear License' for the hardware component and 'GNU General Public License v2.0' for the software component. Check the repository for details.

A.3 Description

How to access

To access the artifact, clone the repository from GitHub:

```
$ git clone https://github.com/cmu-snap/pigasus.git
```

This repository also includes a README with the most up-to-date instructions on how to install and extend Pigasus.

[10] https://figshare.com/articles/media/snort_baseline_mp4/12922160

[11] https://figshare.com/articles/media/pigasus/12922178

Hardware dependencies

Pigasus requires a host with an Intel Stratix 10 MX FPGA [2]. This host should have PCIe Gen3 or greater and a slot with 16 lanes for the FPGA. Experiments require an extra host equipped with a DPDK-compatible 100 Gb NIC to be used as a packet generator. For the experiments, the two hosts are connected back to back. The power-measurement experiments require either a CPU with a power measurement interface (*e.g.*, RAPL [23]) or the use of an external electricity usage monitor.

Software dependencies

Pigasus' software component is adapted from Snort 3 [5] and inherits the same software dependencies. §A.4 provides instructions on how to install those. The provided implementation works on Linux only and was tested on Ubuntu 16.04 and 18.04. Experiments require the installation of vanilla Snort 3, for comparison, as well as DPDK pktgen and Moongen in the packet generator host. To be able to load the bitstream on the FPGA, an installation of Quartus 19.3 as well as the Stratix 10 device support are required.[12]

Data sets

To obtain the Stratosphere traces go to https://www.stratosphereips.org/datasets-overview.

A.4 Installation

These instructions assume that you already have the bitstream to be loaded on the FPGA. For instructions on how to synthesize the design, refer to the repository README.

Software Configuration

In a system running a fresh install of Ubuntu 18.04, with the Pigasus repository cloned to the home directory, start by setting the required environment variables and useful aliases by adding the following to your `.bashrc` or equivalent:

```
export pigasus_rep_dir=$HOME/pigasus
export pigasus_inst=$HOME/pigasus_install
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
export LUA_PATH="$pigasus_inst/include/snort/lua/?.lua;;"

alias pigasus="taskset --cpu-list 0 $pigasus_inst/bin/snort"
alias sudo='sudo '
```

Make sure you apply these changes:

```
$ source ~/.bashrc
```

Then install the dependencies using the provided script:

```
$ cd $pigasus_rep_dir
$ ./install_deps.sh
```

Once the dependencies are installed, build Pigasus as follows:

```
$ cd $pigasus_rep_dir/software
$ ./configure_cmake.sh --prefix=$pigasus_inst
     --enable-pigasus --enable-tsc-clock
     --builddir=build_pigasus
$ cd build_pigasus
$ make -j $(nproc) install
```

[12] Both can be obtained at: https://fpgasoftware.intel.com/19.3/.

Hardware Configuration

To load the bitstream make sure the Quartus tools are in your path by setting the following environment variables in your `.bashrc` or equivalent:

```
# quartus_dir should point to the Quartus installation dir.
export quartus_dir=
export INTELFPGAOCLSDKROOT="$quartus_dir/19.3/hld"
export QUARTUS_ROOTDIR="$quartus_dir/19.3/quartus"
export QSYS_ROOTDIR="$quartus_dir/19.3/qsys/bin"
export IP_ROOTDIR="$quartus_dir/19.3/ip/"
export PATH=$quartus_dir/19.3/quartus/bin:$PATH
export PATH=$quartus_dir/19.3/modelsim_ase/linuxaloem:$PATH
export PATH=$quartus_dir/19.3/quartus/sopc_builder/bin:$PATH
```

Make sure you apply these changes:

```
$ source ~/.bashrc
```

A.5 Evaluation and expected result

In what follows, we describe how to run the experiments to reproduce Pigasus results from Figure 10. Before every experiment we reload the bitstream on the FPGA and reboot the server. This ensures that we always start from the same FPGA state:

```
$ cd $pigasus_rep_dir/pigasus/hardware/hw_test/
$ ./load_bitstream.sh
$ sudo reboot
```

Once the machine is back, to run the software component, first insert the kernel module:

```
$ cd $pigasus_rep_dir/software/src/pigasus/pcie/kernel/linux
$ sudo ./install
```

Then, run Pigasus, using the following command:

```
$ cd $pigasus_rep_dir/software/lua
$ sudo pigasus -c snort.lua --patterns ~/rule_list
```

The `snort.lua` uses the same syntax as in Snort 3, you should modify it to include the Snort Registered Rule Set [6]. In our experiments, we modified the rules to remove some features currently not supported by Pigasus, including `services`, `file_data` and `nocase`. We also use the same modified rules in the baseline experiment.

When Pigasus finishes the startup process it will stop printing logs to the screen. Once this happens, you can invoke the FPGA JTAG console to configure the FPGA internal state. To do so, open another terminal and enter:

```
$ cd $pigasus_rep_dir/hardware/hw_test/
$ ./run_console
% source path.tcl
```

If the last command produces an error, exit the JTAG console with `Ctrl+C` and rerun the last two commands. Once the last command runs successfully type the following commands to configure the buffer size, set the number of cores, and check the FPGA internal state:

```
% set_buf_size 262143
% set_core_num 1
% get_results
```

This last command should return all zeros as no packets have been sent yet.

Now that Pigasus is running and properly configured, we can start the packet generator on another machine. Here we assume that DPDK pktgen is properly configured on the other machine and has been started.

You can specify the rate to send packets, where 100 means 100% line rate. To ensure that DPDK pktgen will only send the trace once, specify the number of packets to match the trace size. The example pcap we are using is the `norm-2.pcap`, which has 456,709 packets. After setting these parameters, you can start sending packets.

```
Pktgen:/> set 0 count 456709
Pktgen:/> set 0 rate 100
Pktgen:/> str
```

Once the packet generator finishes sending packets, go back to the JTAG console on the other host and type the following:

```
% get_results
```

This should return 456,709 received packets and 456,709 processing packets. This means that Pigasus processed all the packets sent at max rate, without loss.

Now stop Pigasus by going back to the first terminal and typing `Ctrl+C`. It will print `rx_pkt`, which should match the `dma_pkt` reported by the FPGA in second terminal. This means that all packets sent from the FPGA to the CPU for full evaluation were processed.

A.6 Experiment customization

Experiments may be customized to use different rule sets and different packet traces. Pigasus design can also be changed to support a different number of concurrent flows or rules.

A.7 Artifact Evaluation Methodology

Submission, reviewing and badging methodology: https://www.usenix.org/conference/osdi20/call-for-artifacts