Achieving 100Gbps Intrusion Prevention on a Single Server

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

Abstract

Intrusion Detection and Prevention Systems (ID/PS) are among the most demanding stateful network functions. Today’s network operators are facing with securing 100Gbps networks with 100K+ concurrent connections by deploying ID/PSes 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 ID/PS, 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 12 cores and 1 FPGA, using 25× less wattage than a CPU-only approach.

1 Introduction

Intrusion Detection and Prevention Systems (ID/PS) are a critical part of any operational security deployment [33, 34]. 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 rule sets are obtained through offline techniques (e.g., crafted by experts or obtained from proprietary vendor algorithms) [3].

A recurring theme in ID/PS 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 ID/PSes that can support line rates on the order of 100Gbps with hundreds of thousands of concurrent flows and capable of matching packets against tens of thousands of rules. This paper answers this challenge with the Pigasus FPGA-based ID/PS 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 [24, 26]. 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 FPGA for low power, low costs and high performance for some simpler network functions such as software switching in Microsoft’s AccelNet [17].

While many before us have integrated FPGAs with ID/PS processing [8, 12, 14, 15, 18, 29, 35–37, 39, 41, 42], 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 ID/PS 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 ID/PS processing. Here, the majority of packet processing for ID/PS is performed via a highly-parallel datapath on-board the SmartNIC FPGA, connected directly to 100Gbps transceivers. The Pigasus FPGA performs TCP re-assembly directly on board so it can immediately apply exact string matching algorithms over packet headers and payloads 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, 4.4 signatures/packet and 11% 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 multistring 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 [25, 31] 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 rule sets, we measured that supporting the Snort Register Ruleset would require 23MB of Block RAM (BRAM), more than the entire capacity of our FPGA (16 MB). We instead take inspiration from Hyperscan [40], 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 about 2MB of memory overall.

**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 [42] 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.

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 12 cores and 1 FPGA, requiring an average of 232 Watts. In contrast, Snort 3.0 – which uses Hyperscan [40] for string matching – would require 364 cores and 5,800 Watts.

## 2 Background & Motivation

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

### 2.1 ID/PS Functionality

The key goal of a signature-based ID/PS 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 ID/PS operates in detection mode, a triggered signature results in an alert or an event recorded to a log. When an ID/PS 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. ID/PSes, on the other hand, may operate asynchronously and are often deployed over a secondary traffic ‘tap’ which provides copies of the active traffic.

**Software ID/PS Performance:** One of the most widely-known ID/PSes is Snort [32] and our work aims to be compatible with Snort rule sets. In our experiments, we primarily work with the Snort Registered Ruleset, which contains roughly 10,000 signatures.

In 2018, Intel published Hyperscan [40], an x86-optimized library for performing both exact-match and regular-expression string matching. Hyperscan is the key new element in Snort 3.0, which is 8× faster than its predecessor which used more traditional algorithms (e.g. Aho-Corasick [5]) for search. 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. On average, Snort 3.0 can process 400Mbps loss-free on a single core. Generously assuming that Snort 3.0 is capable of perfect multicore scalability, this would require 250 cores to support 100Gbps of throughput, or 5-25 servers.

---

1 There exist other models of ID/PS which are ‘script based’ – executing arbitrary user code over scanned traffic – such as Zeek [4, 30]. These ID/PSes are out of scope for this work.
2.2 FPGA Basics

Why look to FPGAs to improve ID/PSes? While there are many platforms (`accelerators`) that offer highly parallel processing, we choose FPGAs because they are (a) energy-efficient (using 4-5× [9] fewer Watts than GPUs) 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 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 ID/PS Performance

We are not the first to integrate FPGAs into ID/PS processing. However, prior work 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 [8, 12, 14, 15, 18, 29, 35–37, 39, 41, 42]. Most research in this space targets offloading regular expression matching alone [18, 35, 41], although some target TCP reassembly [36, 42] or header matching [22] instead. Unfortunately, a basic analysis based on Amdahl’s law reveals why this approach fundamentally cannot bring ID/PS 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: header and string matching (as both are implemented in the same module in Snort 3.0, which is called Multi-String Pattern Matching or MSPM), full matching including regular expressions, TCP reassembly, and other tasks. As we can see, no single task dominates CPU time – at most, MSPM consumes 46% of CPU time for Mixed-4 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.

3 System Overview

We now present an overview of our system, 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 present our rationale behind Pigasus’ FPGA-first approach (§3.1) before presenting Pigasus’ packet processing datapath module by module (§3.2). Finally, we discuss considerations for making best use of FPGA memory resources (§3.3).
3.1 An FPGA-First Design

Following our analysis in §2.3, we argue for an FPGA-first design for ID/PS 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. To see why this approach is necessary, consider a simple application of Amdahl’s Law. To achieve, e.g., a 10× throughput speedup, one must parallelize at least 90% of the work in the running system. Any approach to speed up ID/PSes to this degree must hence 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 on FPGA provides lower performance benefits at a higher cost in terms of memory. The Full Match stage provides relatively lower performance benefits because, on average, it only sees 11% of packets. The MSPM first searches each packet for a partial rule match based on a single exact-match string; the Full Match stage is then responsible only for checking those (packet, rule) pairs which the MSPM flags suspicious (in our dataset, on average, each packet comes with only 4.4 of the original 10K rules to check). At the same time, moving the Full Match stage to the FPGA would come with substantial memory cost, requiring that the FPGA store the complete rule-set in parallel-searchable data structures; by keeping the Full Match off the FPGA only needs to store the header match and a single exact-match string for each rule. In short; the performance gains from moving the Full Match stage to FPGA are not necessary to meet our 100Gbps goal, and the cost of moving Full Match to FPGA steals memory resources from the Reassembler and MSPM thus hindering our ability to support 100K flows and 10K rules.

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. We now describe how these pieces work together.

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 continuously (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. This module decouples the data movement task from Reassembler packet processing tasks to give Reassembler more cycle budget on processing packets.

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 the exact-match filters for all 10,000 rules. Thus, it has the most demanding performance requirements among our modules. At a 400MHz clock rate, the payload matching alone requires searching 32 indices against all 10,000 signatures every clock cycle. We describe how Pigasus achieves this 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 on PCIe bandwidth, 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.

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 (regular expressions, secondary exact match strings) 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 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-Pattern String Matcher. Specifically, the Reassembler performs multiple accesses per packet as it needs to check for out-of-orderness, manages the out of order packet buffer, and checks and releases 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. 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 in advance of 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 11% 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 the full capacity of BRAM for the Reassembler and Multi-Pattern String 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 the next two sections, we describe our design optimizations to ensure that the Reassembler and MSPM both ‘fit’ on-board without compromising performance.

4 Reassembly

Reassembly refers to the process of reconstructing a TCP bytestream in the presence of packet fragmentation, network loss, and out-of-order packet 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 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 is not responsible, e.g. for producing ACKs; the ID/PS is a passive observer of traffic between two existing endpoints and 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, [42] allocates a fixed, 64KB packet buffer in DRAM and uses 7 pairs of pointers to track OOO state for each flow; similarly, [36] 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

---

2FPGA 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.

3We do use BRAM in some places for internal buffers/queues.
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 [36] 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 which amounts to at least 100Gbps so long as packet size is greater than 500B.

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. However, 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.

We leverage this idea in Pigasus’s TCP Reassembler (§4.2).

4.2 Pigasus TCP Reassembler

Pigasus takes the linked list approach. As mentioned above, this results in 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 identifier5 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 as packet size is greater than 500B.

OOO Engine: The OOO Engine does not run in constant time, instead dequeuing packets provided for it from the Fast Path as it finishes operating over the previous packet.5 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.

5Experimentally, 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.
**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 [31]: an FPGA-based hash table that employs deamortized cuckoo hashing [6, 23]. We illustrate this data structure in Figure 6. The design provides high memory density (up to 97% occupancy using 4 or more sub-tables [6, 23, 31]), 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 [6]. We implement the hash table using dual-port BRAM, and the Insertion Queue using a parallel shift-register (capable of storing 16 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 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 dequeued 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.

### 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. To the best of our knowledge, there are other no hardware or software projects reporting multistring matching of tens thousands of strings at 100 Gbps. Classically implemented with parallelized NFAs, the best string matcher for hardware that we know of [12] would require 23MB of BRAM to represent the exact match search strings alone (ignoring the additional header matches).

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 $8 \times$ [40] speedup to state-machine based string matchers in software. The challenge is in extending Hyperscan to even higher degrees of parallelism to reach our 100Gbps goal, while also fitting all datastructures in memory. Our MSPM reaches 100Gbps by parallelizing the Hyperscan approach $32 \times$ over – searching 32 indices of the bytestream simultaneously in one clock cycle – while fitting the entire MSPM module in under 2MB.

#### 5.1 MSPM in Software

**Role of MSPM in ID/PS**: 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. 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. This division is what allows Pigasus to do the
majority of processing on-board the FPGA: on average, only 11% of packets reach the full matcher in our implementation.

**Snort 3.0 + Hyperscan:** In Snort 3.0, the MSPM is implemented with Intel’s Hyperscan, illustrated in Fig. 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, 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 * l lookups for an l length packet.

To reduce the number of expensive sequential lookups, each string matcher contains an SIMD-optimized shift-or filter [7] prior to the hash table; this filter outputs a 0 or 1 value for every index of the packet marking whether or not that index matches any fast pattern in the hash tables; indices which result in a 0 output from the shift-or 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 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 show how Pigasus partitions and re-orders the Hyperscan algorithm to reach this high degree of parallelism within available resources.

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

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. Furthermore, the hash tables only store a 1-bit entry to indicate whether a given (index, length) 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 5x.

The shift-or and 1-bit hash table\(^3\) consume only 65KB and 25KB respectively, thus they are relatively cheap to replicate the 32× over required to scale to 100Gbps.

In theory, the filters can generate 32 * 8 matches per cycle, however, in the common case most packets and most indices do not match any rules, thus requiring no further lookups into additional data structures. This gives us opportunity to make subsequent pipeline stages narrower. We design a Rule Reduction logic to select non-zero rule matches from the filter’s 256-bit wide vector output and narrow it down to 8 values.

By performing this filter first, we are able to apply a much lower degree of replication of subsequent larger data structures, since most bytestream indices have already been filtered out by string matcher. This enables high (effective) parallelism with limited memory overheads.

After the rule reduction logic, we actually fetch the rule IDs and check the header fields that were matched. At this point, we only need to create 8 replicas of the 17KB Rule

---

\(^6\)Up to 8 bytes – longer fast patterns are truncated.

\(^3\)Subtly, this is not a true Bloom filter [10] because we only perform one hash per input; implementing multiple hashes increases resource utilization and complexity, we find, with little gain.
Table and well 68KB Port Grouping modules. We hash the packet index (for the length matched in the filtering stage) and look up the corresponding rule ID in the Rule Table as well as the corresponding port Group ID; we then look up the header pattern in the Port Grouping table and check that the packet matches the header fields in addition to the fast-pattern.

The packet is released along with the set of any rule IDs which it matched with both a fast pattern and a header match.

**Comparing Hardware vs Software**: Although Hyperscan and Pigasus use the same set of datastructures, their rearrangements reflects fundamental differences in how one optimizes software vs hardware.

Hyperscan first targets fitting table lookups in cache: this is the reason hash table lookups are partitioned in port groups. In contrast, on the FPGA, we have careful control of what data sits in fast BRAM versus slower eSRAM or DRAM – thus, in the hardware design, all memory lookups take only one cycle. Secondary to fitting in cache, Hyperscan aims to do as few table lookups as possible. In Pigasus on the other hand, any compute that can be easily parallelized is cheap – and hence the first round of our filter does 256 hash table lookups in one cycle.

Pigasus’ primary objective instead is to fit in as little memory as possible. Where Hyperscan can assume that caching and prefetching will provide a reasonable illusion of infinite, processor-local memory, Pigasus must in practice fit all of this data in BRAM and thus structures its hash tables to be memory-minimal in aggregate.

### 6 Evaluation

In this section, we evaluate Pigasus and show that:

- Pigasus is at least an order of magnitude better than state-of-art Snort running in software: using 7 – 99× fewer cores, and 5 – 65× less power;
- Pigasus performance gains are resilient to a variety of factors such as small packets, out of order execution, 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.

#### 6.1 Setup

**Implementation**: We implement Pigasus using an Intel Stratix 10 MX FPGA Development card as the SmartNIC in an 8-core (Intel i7-7820X @ 3.6 GHz) host machine. The Stratix 10 MX FPGA has 16MB of on-chip BRAM, 10MB of eSRAM, and 8GB of DRAM. To implement Pigasus’s slow path, we adapt Snort 3 to allow it to receive reconstructed PDU and rule IDs, coming from the FPGA, directly into its Full Matcher. For comparison, we run Snort 3 software experiments on the same server hardware as Pigasus.

**Traffic Generator**: We installed both DPDK Pktgen [1] and Moongen [16] 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 replayingPCAP 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.

**Ruleset**: We test Snort and Pigasus both using the publicly available Snort Registered Ruleset (snapshot-29141) [3].

**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.

#### 6.2 End-to-end performance and costs

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

![Figure 9: Extrapolated number of cores to process each trace at 100Gbps using Pigasus (FPGA + CPUs) and Snort (traditional CPUs alone).](image)
Provisioning for 100Gbps throughput: The green bars in Figure 9 report the estimated 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 final full matching decision to allow the cores to catch up after the peak has passed.

The Pigasus results are based on experiments where a one-core Pigasus system (configured for the standard Snort ruleset and provisioned for 500K concurrent flows) is tested at increasing data rates until it begins to drop packets. This is when the processor core is saturated by the full matching workload. In separate experiments, we have ascertained that the FPGA frontend and PCIe transfer can keep up all the way to the packet generator maximum data rate of 85+Gbps. Using the data rate to saturate one core, we estimate the minimum number of cores that would be needed to keep up with the FPGA frontend at 100Gbps. Similar estimates are made for Snort running in IDS mode (blue) and IPS mode (green), respectively for reference.

Overall, we see that Snort in IDS mode requires $7 - 99 \times$ more cores than Pigasus ($42 \times$ on average), and in IPS mode requires $6 - 124 \times$ more cores than Pigasus ($52 \times$ on average). To extrapolate the total cost of such a system, we use per-core pricing data for the AMD EPYC 7452 CPU from to estimate the total cost of each configuration. For Pigasus, we also incorporate the market price of a Stratix 10 MX FPGA to arrive at the final estimate. We observe that, when provisioning for the peak demand in the ‘mixed’ traces, the large FPGA cost is readily amortized and results in savings of 30-70% over either Snort configuration. Conversely, in the average scenario, the cost is 0.6-47% lower for Snort configurations compared to Pigasus. However, it is important to remember that these estimates represent capital cost investments; practical deployments should also consider long-term operational due to cooling and power (which, as we will show later, favor Pigasus).

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 10. We find 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 Pigasus 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 6 µs), Parser (by 4 µs), and the MSPM (by 3µs) as expected relative to software, the additional reduction came from avoiding Packet I/O overhead in software (around 5 µs).

Power footprint: Figure 11 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 [19] to measure per-core power consumption in steady-state. On the FPGA side, we use the Board Test System (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 $5 - 65 \times$ higher power consumption than Pigasus ($22 \times$ 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).

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’s performance sensitivity to traffic characteristics. We probe deeper into Pigasus’s performance under differing levels of malicious traffic.
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 mixed-1 trace with innocent trace normal-2 in different proportion. Figure 12 (a) depicts the dependence of single-core, zero-loss throughput on the fraction of malicious traffic (in terms of relative packet count). We report three sets of results: Pigasus (Hardware), which represents the hardware-only throughput achieved on the Pigasus FPGA frontend; Pigasus (End-to-End), which represents Pigasus’s overall, single-core throughput (including CPU-side full matching); and Snort IPS, which represents Snort’s single-core throughput. We observe that, as long as the fraction of malicious traffic is smaller than 5%, Pigasus is able to process packets at line-rate using a single CPU core. Ignoring the CPU bottleneck, the Hardware-only performance does not begin to degrade (and only gradually afterwards) until the the 15% mark.

Figure 12 (b) depicts the number of cores required to achieve 100Gbps as a function of the fraction of malicious traffic. Despite the performance cliff observed in (a), Pigasus scales considerably better than Snort: the relative difference in core counts is between 6× (at 100% malicious traffic) and 124× (at <5% malicious traffic). We also note that, while the Hardware-only throughput drops to 48Gbps when the traffic is entirely malicious – half our 100Gbps target – we can recover this difference by replicating the hardware pipeline up to two more times (discussed further in §6.4).

**Dependence on Packet Size:** We first consider the impact of packet size on Pigasus’s 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 13 illustrates this dependence. We observe that, for packets exceeding 500B (comparable to average packet sizes on the Internet [11]), Pigasus is capable of processing at line rate. (More generally, Pigasus by design can sustain 100Gbps as long as the average packet size is greater than 500B over a window of 87 μs 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 \(l_p\) [27] and the recovery distance \(rd\). Figure 14 depicts the impact of these parameters on Pigasus’s 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 \((l_p = 0.3\%, rd = 3)\), 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 Re-assembler can handle around 50 OOO packet arrivals without any degradation in end-to-end throughput.

---

12 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.
6.4 Future outlook

Figure 15 depicts the three-way tradeoff between the scalability of the number of rules, the number of concurrent flows, and the number of replicated hardware pipelines. Surprisingly, there is plenty of scaling headroom in the Pigasus FPGA frontend design on all three dimensions. The performance headroom give motivation for future work to improve or eliminate the CPU full checking bottleneck. One lesson is that the Pigasus FPGA frontend can actually be configured to operate at well beyond 100Gbps. We could replicate the design evaluated above up to three times, each to serve a different subset of flows. Hence, we could very well see a single server processing up to 200 or 300Gbps. In future work, we plan to explore moving more functionality from the full matcher onto the FPGA as well.

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 [13] and Hyperscan [40]. An important, 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 ID/PS: Many previous work have also made a case for using FPGAs to implement network functionality [17, 25, 28, 31, 38]. ClickNP [25] and Flow-Blaze [31] present abstractions for making it easy to implement network functionality in FPGA. 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 ID/PS [8, 12, 14, 15, 29, 37, 39, 42]. However, all of these works do not implement a complete ID/PS and fail to meet our target for throughput or number of rules. Even though, Snort Offloader [37] proposes using an FPGA to implement an entire ID/PS, 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 ID/PS components. Kargus [20] uses GPUs to accelerate exact-pattern and regular-expression matching. However, their use of GPUs contributes to increasing both power and latency. PPS [21] 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 ID/PS stages, these solutions are fundamentally limited in the throughput improvements they can achieve.

8 Conclusions

In many ways, ID/PS 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 ID/PS as well.
References


