Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization ☆

Charalampos Stylianopoulos[a,*], Magnus Almgren[a], Olaf Landsiedel[b,a], Marina Papatriantafilou[a]

[a] *Chalmers University of Technology, Sweden*
[b] *Kiel University, Germany*

Abstract

As both new network attacks emerge and network traffic increases in volume, the need to perform network traffic inspection at high rates is ever increasing. The core of many security applications that inspect network traffic (such as Network Intrusion Detection) is pattern matching. At the same time, pattern matching is a major performance bottleneck for those applications: indeed, it is shown to contribute to more than 70% of the total running time of Intrusion Detection Systems. Although numerous efficient approaches to this problem have been proposed on custom hardware, it is challenging for pattern matching algorithms to gain benefit from the advances in commodity hardware. This becomes even more relevant with the adoption of Network Function Virtualization, that moves network services, such as Network Intrusion Detection, to the cloud where scaling on commodity hardware is key for performance.

In this paper, we tackle the problem of pattern matching and show how to leverage the architecture features found in commodity platforms. We present efficient algorithmic designs that achieve good cache locality and make use of modern vectorization techniques to utilize data parallelism within each core. We first identify properties of pattern matching that make it fit for vector-

☆Preliminary results of this work were presented the 46th International Conference on Parallel Processing (ICPP) 2017 [1].

*Corresponding author

Email addresses: chasty@chalmers.se (Charalampos Stylianopoulos), magnus.almgren@chalmers.se (Magnus Almgren), ol@informatik.uni-kiel.de (Olaf Landsiedel), ptrianta@chalmers.se (Marina Papatriantafilou)

ization and show how to use them in the algorithmic design. Second, we build on an earlier, cache-aware algorithmic design and show how we apply cache-locality combined with SIMD gather instructions to pattern matching. Third, we complement our algorithms with an analytical model that predicts their performance and use it to easily evaluate alternative designs. We evaluate our algorithmic design with open data sets of real-world network traffic: Our results on two different platforms, Haswell and Xeon-Phi, show a speedup of 1.8x and 3.6x, respectively, over Direct Filter Classification (DFC), a recently proposed algorithm by Choi et al. for pattern matching exploiting cache locality, and a speedup of more than 2.3x over Aho-Corasick, a widely used algorithm in today's Intrusion Detection Systems. Finally, we utilize highly parallel hardware platforms and evaluate the scalability of our algorithms, achieving processing throughput of up to 45Gbps.

Keywords: `pattern matching, SIMD, vectorization, gather`

---

1. Introduction

Pattern matching is an essential building block for many security applications, such as antivirus programs or Network Intrusion Detection Systems (NIDS). In its core, pattern matching algorithms operate on two sets of input: (i) a predefined set of patterns and (ii) an incoming stream of data and attempt to detect if any of the patterns exist in the stream. In this work, we focus on the problem of fixed-string, *multiple* pattern matching, i.e. the patterns are string literals and, differently from single pattern matching [2, 3], we are simultaneously tracking the presence of many patterns. In the context of Network Intrusion Detection Systems, the set of patterns are *signatures* of known malicious attacks (usually in the order of thousands) that the system aims to detect and the data stream is the reassembled stream of packets captured from the network interface.

Motivation and Challenges. Pattern matching represents a major performance bottleneck in many security mechanisms, especially when there is a need to employ analysis on the full packet's payload (Deep Packet Inspection). In intrusion detection, for example, more than 70% of the total running time in spent on pattern matching [4, 5]. Moreover, with the increasing interest in Network Function Virtualization (NFV) [6, 7], applications like firewalls and Network Intrusion Detection are now expected to be placed in the application layer of the control plane [8], where they need to rely on

commodity hardware features for performance, like multi-core parallelism and vector processing pipelines.

In this paper, we introduce a vectorizable design of an exact pattern matching algorithm which nearly doubles the performance when compared to the state of the art, on SIMD-capable commodity hardware, such as Intel's Haswell processors or Xeon Phi [9]. *Vectorization* as a technique to increase throughput is gradually taking a more central role [10]. For example, architectures with SIMD instruction-sets now provide wider vector registers (256 bits with AVX) and introduce new instructions, such as gathers, that make vectorization applicable to a wider range of applications. Moreover, modern processor designs are shifting towards new architectures, like Intel's Xeon Phi [9], that, for example, supports 512 bit vector registers. On those platforms, vectorization is not just an option but a must, in order to achieve high performance [11]. In this work we introduce algorithmic designs to utilize these capabilities.

Approach and Contributions. The introduction of *gathers* and other advanced SIMD instructions (cf. section 3) allows even applications with irregular data patterns to gain performance from data parallelism. For example, SIMD can speed up regular expression matching [12, 13, 14]. Here, the input is matched against a single regular expression at a time, represented by a finite state machine that can fit in L1 or L2 cache. Working close to the CPU is crucial for these approaches, otherwise the long latency of memory accesses would hide any computation speedup through vectorization.

The domain of multiple pattern matching for Network Intrusion Detection has challenging constraints that limit the effectiveness of these approaches: applications need to simultaneously evaluate thousands of patterns and traditional state-machine-based algorithms, such as Aho-Corasick [15], use big data structures that by far exceed the size of the cache of today's CPUs. The size of the patterns varies greatly (from 1-byte to several hundred byte patterns) and can appear anywhere in the input. That is why SIMD techniques have not been previously considered for exact multiple pattern matching – with a few exceptions discussed in Section 7 – for Network Intrusion Detection.

Building upon recent work [16, 17] that take steps in addressing the cache-locality issues for this problem, our approach fills this gap: we propose algorithmic designs for multiple pattern matching that bring together cache locality and modern SIMD instructions, to achieve significant speedups when compared to the state of the art. Combining cache locality and vectorization

introduces new trade-offs on existing algorithms. Compared to traditional approaches that perform the minimum required number of instructions, but on data that is away from the processor, our approach, instead, performs more instructions, but these instructions find data close to the processor and can process them in parallel using vectorization.

Our work builds on a family of recent algorithms that take steps towards providing good cache locality for multiple exact pattern matching [16, 17]. They filter parts of the input streams using small, cache efficient data structures. We argue that, as a result, memory latencies are no longer the dominant bottleneck for this family of algorithms while their computational part becomes more significant. In this work, we follow a two-step approach. First, we propose a refined and extended method, which is able to benefit from vectorization while ensuring cache locality. Second, we design its vectorized version by utilizing SIMD hardware *gather* operations. To evaluate our approach, we apply our techniques to the DFC algorithm [16], as a representative example that outperforms existing techniques in Network Intrusion Detection applications, including [17], on which our proposed approach can be applied as well. We also include an analytical model that predicts the cost of both our scalar and vectorized algorithms, taking into account the number of malicious patterns given at startup. Finally, we deploy our algorithms on multi-core architectures and utilize all the available hardware parallelism, both within each core (with vectorization) and across many cores. A high-level illustration of our approach is shown in Figure 1.

In particular, we target the computational part of pattern matching for performance optimization and make the following contributions:

- We propose algorithmic designs for multiple pattern matching which (a) ensure cache locality and (b) utilize modern SIMD instructions.
- We devise a new pattern matching algorithm, based on these designs, that utilizes SIMD instructions to outperform the state of the art, while staying flexible with respect to pattern sizes.
- We introduce an analytical model to predict the performance of both our scalar and vectorized algorithms, based on the number of patterns. We evaluate the model with real-world data and find that it closely follows the observed trends.
- We (implement the algorithm and) thoroughly evaluate it under both real-world traces and synthetic data sets. We outperform the state of

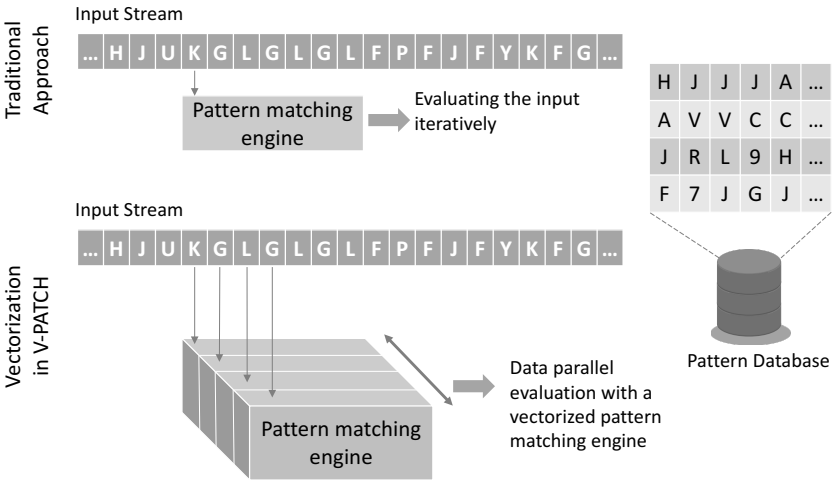


Figure 1: A general example of pattern matching at the top, and our proposed vectorized pattern matching approach at the bottom.

the art by up to 1.8x on commodity hardware and up to 3.6x on the Xeon-Phi platform.

- We evaluate the scalability of our algorithms when using all the parallelism offered by the platform and achieve up to 40 Gbps processing throughput on the Haswell platform and 45Gbps on the Xeon-Phi.

The remainder of the paper is organized as follows: Section 2 gives an overview of important pattern matching algorithms and background on vectorization. Section 3 describes our system model. In Section 4, we present our approach leading to a new, vectorized design. In Section 5 we introduce an analytical model to predict the performance of our scalar and vectorized algorithms. Section 6 presents our experimental evaluation on the performance of our algorithms under a variety of evaluations scenarios. In Section 7, we give an overview of other related work and we conclude in Section 8.

2. Background

In this section we present traditional approaches to pattern matching, followed by a brief description of the DFC algorithm (Choi et al. [16]) to which we apply our approach. Next, we introduce the required background on vectorization techniques.

2.1. Traditional Approach to Multiple-Pattern Matching

The most commonly used pattern matching algorithm for network-based intrusion detection is by Aho-Corasick [15]. It creates a finite-state automaton from the set of patterns and reads the input byte by byte to traverse the automaton and match multiple patterns. Even though it performs a small number of operations for every input byte, it implies– in practice and on commodity hardware – a low instruction throughput due to frequent memory accesses with poor cache locality [16]: As the number of patterns increases, the size of the state automaton increases exponentially and does not fit in the cache. Nevertheless, the method is heavily used in practice; e.g., both Snort [18], one of the best known intrusion detection systems, as well as CloudFlare's web application firewall [19], use it for string matching.

2.2. Filtering Approaches and Cache Locality in Multiple Pattern Matching

Besides state-machine based approaches, there is a family of algorithms that rely on *filtering* to separate the innocuous input from the matches. Recent work focuses on alleviating the problem of long latency lookups on large data structures. Choi et al. [16] present a novel algorithmic design called DFC (Direct Filter Classification), that replaces the state machine approach of Aho-Corasick with a series of small, succinct summaries called *filters*. Such a filter is a bit-array that summarizes only a specific part of each pattern, e.g. its first two bytes, having one bit for every possible combination of two characters that can be found in the patterns. The algorithm is structured in two phases, the *filtering* and *verification*:

- In the *filtering* phase, a sliding window of two bytes over the input goes through an initial filter, as described above, to quickly evaluate whether the current position is a possible starting point of a match. The two-byte windows that passed the initial filter are fed to other, similar filters, each specializing on a family of patterns depending on their length. Since the filters are small (8KB each), they usually fit in L1 cache. Thus, the main part of the algorithm differs from Aho-Corasick and uses only cache-resident data structures, resulting in up to 3.8 times less cache misses [16].

- If a window of two characters passed all filters, there is a strong indication that it is a starting point of a match. For this reason, in the next *verification* phase, the DFC algorithm performs lookups on specially designed hash tables, containing the actual patterns and performs exact matching on the input and the pattern, to verify the match.

Other algorithms in this family, like [17] as well as this work, operate on the same idea: the input is filtered using cache resident data structures, and only the "interesting" parts of the input is forwarded for further evaluation.

2.3. Vectorization

Single Instruction Multiple Data (SIMD) is an execution model for data parallel applications, which utilizes processing units that operate on a vector of elements simultaneously, instead of separate elements at a time. SIMD instructions utilize the vector execution units, a separate pipeline found in modern processors that operates on multiple registers with almost the same cost as the equivalent scalar instructions. SIMD vectorization is a desirable goal in computationally intensive, number-crunching applications, where computation is performed on independent data, *sequentially* stored in memory. However, until recently, most algorithms that did not follow this sequential access patterns were difficult to vectorize.

Vector instruction sets have evolved over time, introducing bigger registers and support for more complex instructions. Originally offering support for up to 128 bits, vector instruction sets are now extended to 256 bit-long vector registers and new generation platforms, such as the Xeon-Phi [9], support up to 512 bit-long vector registers, which indicates the vendor effort to accelerate applications that utilize data parallelism. Recently, vector instruction sets on commodity hardware have been enriched with the *gather* instruction [20] that enables accessing data from *non-contiguous memory locations* (described in detail in Section 3). Polychroniou et al. [21] study the effect of vectorization with the *gather* instruction on a series of data structures, such as Bloom-Filters, hash-table lookups, joins and selection scans, among others. We are building on these works with SIMD instructions and extend their design to pattern matching with the applications we focus on.

3. System model

In this section we introduce the assumptions and requirements that our approach makes on the hardware. We focus on mainstream CPUs, with vector processing units (VPUs) that support *gather* instructions. The latter make it possible to fetch memory from non-contiguous locations using only

SIMD instructions[1]

The semantics of *gather* are as follows: let W be the vector length, which is the maximum number of elements that each vector register can hold. The parameters to the instruction are a vector register (I) that holds W indexes and an array pointer (A). As output, *gather* returns a vector register (O) with the W values of the array at the respective indexes. It is important to note that *gather* does not parallelize the memory accesses; the memory system can only serve a few requests at a time. Instead, its usefulness lies in the fact that it can be used to obtain values from non-contiguous memory locations using only SIMD code. This increases the flexibility of the SIMD model and allows to efficiency employ it for workloads previously not considered, i.e., where the memory access patterns are irregular. The alternative is to load the values using scalar code, then transfer them one by one from the scalar registers into vector registers. Generally, switching between scalar and vector code is not efficient [22, 21].

Apart from *gather*, the rest of the instructions we use can be found across almost all the vector instruction sets available. Worth mentioning is the *shuffle* instruction, that makes it possible to permute individual elements within the vector register in any desired order. For example, we employ it for handling the input and output of the algorithm (cf. Section 4.2).

The size of the cache, especially the L1 and L2, is very important for the algorithmic design, as we describe later in Section 4. Common sizes in modern architectures is 32 KB of L1 data cache with 256 KB of L2 cache and we will use this as a running example. Our design is applicable to other cache sizes as well.

4. Algorithmic Design

In this section, we begin by introducing S-PATCH, an efficient algorithmic design for multiple pattern matching. It is designed with both cache locality and vectorizability in mind. Next, we propose our vectorization approach V-PATCH, Vectorized PATTern matCHing.

[1]In Intel processors, the *gather* instruction was introduced with the AVX2 instruction set and is included in the latest family of mainstream processors; *gather* also exists in other architectures, such as the Xeon Phi co-processor [9].

4.1. S-PATCH: a vectorizable version of DFC

To enable efficient vectorization, we introduce significant modifications to the original DFC design. The key insight for the modifications, explained later in detail, is that small patterns will be found frequently in real traffic, so they should be identified quickly without adding too much overhead. On the other hand, long patterns are found less frequently, but detecting them takes longer and requires more characters from the input to pinpoint them accurately.

As in the original DFC, our approach has two parts, but it is organized as two separate rounds. In the **filtering** round, we examine the whole input and feed it through a series of filters that bear some similarities to DFC, but adapted to consider properties of realistic traffic, as motivated above. The **verification** round is as in DFC and performs exact matching on the full patterns that are stored in hash tables. Compared with DFC, S-PATCH focuses on efficient filtering in the first round, because this is the computationally intensive part of the algorithm that, as we show, can be efficiently vectorized. Splitting the two parts in separate rounds improves cache locality, since the data structures used in each round do not evict each other and, as shown in Section 4.2, makes vectorization more practical.

4.1.1. Filtering

In this first phase the goals are to (i) quickly eliminate the parts of the input that cannot generate a match and (ii) store the input positions where there is indication for a match. In general, key properties of the filtering phase include:

- Good filtering rate. A big fraction of the input is filtered out at this stage. This is important, in order to avoid performing verification frequently, as it has higher cost than filtering. The achieved filtering rate is directly dependant on the number of patterns inserted in each filter (see also the cost and hit rate predicted by the model described in Section 5).

- Low overhead. Every filter introduces additional computations and memory accesses, so there needs to be a balance between its overhead and the amount of input that is filtered out. Later in Section 5, our model quantifies the filtering overhead and the filtering rate, to help us maintain that balance.

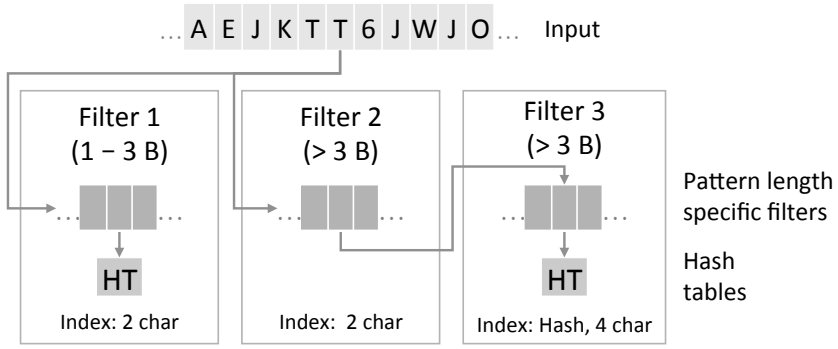


Figure 2: Filter Design of S-PATCH. HT stands for the *Hash Tables* that contain the full patterns.

- Size-efficiency. All the filters need to fit in L1 or L2 cache, while also leaving room for the input and the array for the intermediate results in cache. This is very important, because it ensures that the lookups on the filters will be fast and, as explained later, vectorization using the gather instruction will be feasible.

Our proposed filter design (cf. Figure 2) consists of three filters, each with a specific purpose. The first one stores information about the short patterns (less than 4 characters). It has one bit for every possible combination of two characters, and if a particular combination is the beginning of a pattern, the corresponding bit is set. Similarly, the second filter uses the same indexing and accounts for the longer patterns together with the third filter. An example of how filters are populated (in this example, Filter 2) is shown in Figure 3. In more detail on how we scan the input against the filters (cf. also Algorithm 1):

First filter: In the first part of the filtering, we examine two bytes of the input at a time and use them to calculate an index for filters 1 and 2. If the corresponding bit in the first filter is set, we directly store the current input position in an array for further processing (lines 5-7).

Second filter: We also perform a lookup on the second filter using the same index, at line 8. A hit may indicate that we have a match with a longer pattern, but it may also be a false positive (e.g. compare the strings "**at**tribute" and "**at**tack"). Thus, before storing the current input position after a match with the second filter, the algorithm uses more bytes (in our case four) from the input stream with a third filter to gain stronger indications

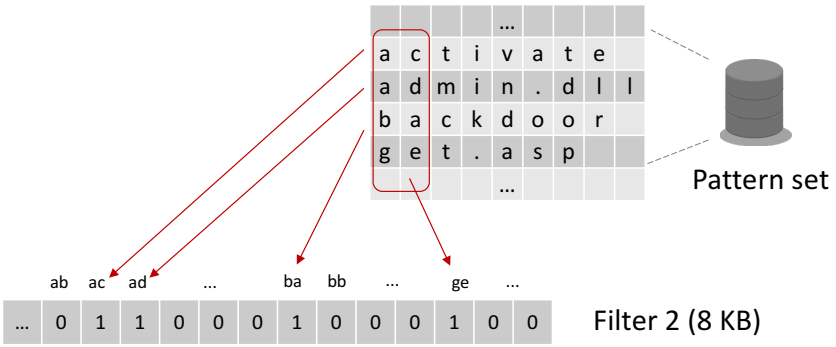


Figure 3: An example showing how Filter 2 is created, based on the patterns found in the pattern set.

whether there is actually a match. Only when the match in the second filter is corroborated with a match from the third filter is the current position in the input stream stored for further processing (line 11).

Third filter: For the third filter, the index is calculated differently; we cannot have a filter with all combinations of four bytes, due to cache-size limitations. Instead, we use a multiplicative hash function for the four bytes of input to compute the index in the filter, at line 9. There is a trade-off between having a large enough filter to avoid collisions (thus providing a good filtering rate) and having it small enough to fit in cache. The reason why we choose four bytes as input will become clear in the next section (4 bytes fit in each one of the 32-bit vector register values).

Note that the performance of the filtering phase is intrinsically tied to the filter designs and the type of input. The reason why our proposed design is more effective is twofold. Short patterns, although few,[2] are likely to generate many matches. As an example, if strings like `GET` and `HTTP` are part of the pattern set, they will frequently be found in real network traffic. Treating them separately in a dedicated filter allows us to focus on the longer patterns in other filters. Long patterns, found more rarely, require more information to be distinguished from innocuous traffic.

[2]21% of Snort's v2.9.7 patterns are 1-4 bytes long [16].

```
Data: D: data to inspect
# A_short : temporary array for short patterns
# A_long : temporary array for long patterns
for i=0, i <D.length, i++ do
    index = Read two bytes from pos i in D
    if (Filter1[index] is set) then
        Store i in A_short
    end
    if (Filter2[index] is set) then
        new_index = hash 4 bytes from input
        if Filter3[new_index] is set) then
            Store i in A_long
        end
    end
end
for i=0, i <A_short.length, i++ do
    Verification for small patterns
end
for i=0, i <A_long.length, i++ do
    Verification for big patterns
end
```

Algorithm 1: Pseudocode for S-PATCH.

4.1.2. Verification

After the filtering, all the possible match positions in the input have been stored in a temporary array. At this point, we need to compare the input at these positions with the actual patterns, before we can safely report a match. As mentioned before, the verification phase is as described by Choi et al. [16], except that it is now done in a separate round, after the current chunk of input has been processed by the filtering phase. For ease of reference we paraphrase here.

Among several optimizations, Choi et al. [16] use specially designed *compact hash tables* that are different for different pattern lengths. Translated to our improved filtering design, if the input at some position i passed the filtering, in the verification phase the algorithm will perform a match on the compact hash table that stores references to all the patterns of appropriate size. For example, if i passed the third filter that stores information on patterns that are four bytes or longer, in the verification phase, the algorithm

performs a match on the compact hash table that stores patterns of four bytes or longer (lines 18-20). Each hash table is indexed with as many bytes as the shortest pattern that the hash table contains (in this case, four bytes of the input will be used as an index to the hash table). Each bucket in the hash table contains references to the full patterns and the algorithm has to compare each one of them individually with the input, before reporting a match. Eventually, the algorithm identifies all the occurrences of all the patterns, producing the same output as Aho-Corasick.

In general, the compact hash tables as we use them in this phase, do not fit L1 or L2 cache (but they might fit L3 cache) and accessing them incurs high latency misses. However, the success of the approach lies in the fact that the filtering phase will reject most of the input, so the algorithm resorts to verification only when it is needed (when there is a high probability for a match). That is why our efforts focus on the filtering part, where the data structures are close to the processor and can benefit from vectorization.

4.2. V-PATCH: Vectorized algorithmic design

A basic issue when vectorizing S-PATCH is its non-contiguous memory accesses. The sequential version accesses the filters at nonadjacent locations for every window of two characters, whereas in a vectorized design W indexes are stored in a vector register (of length W), each pointing to a separate part of the data structure. For this reason, we use the SIMD *gather* instruction that allows us to fetch values from W separate places in memory and pack them in a vector register.

Algorithm 2 gives a high level summary of the filtering phase of V-PATCH. The first step towards vectorizing the algorithm is loading the consecutive input characters from memory and storing them in the appropriate vector registers. Figure 4 shows the initial layout of the input and the desired transformation to W elements, each holding a sliding window of two characters. The transformation is efficiently achieved with the use of the *shuffle* instruction, allowing to manually reposition bytes in the vector registers (Algorithm 2, line 8).

Data: D: input data to inspect
1 # W : the vector register length
2 # A_short : temporary array for short patterns
3 # A_long : temporary array for long patterns
4 # $\overrightarrow{M1}$: constant mask used to convert the input to 2 byte sliding window format
5 # $\overrightarrow{M2}$: constant mask used to convert the input to 4 byte sliding window format
6 **for** *i=0, i <D.length, i += W* **do**
7 $\quad \overrightarrow{R}$ = Fill register with raw input from D
8 $\quad \overrightarrow{Indexes}$ = `shuffle`($\overrightarrow{R}$,$\overrightarrow{M1}$)
9 $\quad \overrightarrow{V1}$ = `gather`(filter1_address, $\overrightarrow{Indexes}$)
10 $\quad$ **if** *at least one element in* $\overrightarrow{V1}$ *is set* **then**
11 $\quad\quad$ Store positions of matches in A_short
12 $\quad$ **end**
13 $\quad \overrightarrow{V2}$ = `gather`(filter2_address, $\overrightarrow{Indexes}$)
14 $\quad$ **if** *at least one element in* $\overrightarrow{V2}$ *is set* **then**
15 $\quad\quad \overrightarrow{NewIndexes}$ = `shuffle`($\overrightarrow{R}$,$\overrightarrow{M2}$)
16 $\quad\quad \overrightarrow{Keys}$ = hash($\overrightarrow{NewIndexes}$)
17 $\quad\quad \overrightarrow{V3}$ = `gather`(filter3_address, $\overrightarrow{Keys}$)
18 $\quad\quad$ **if** *at least one element in* $\overrightarrow{V3}$ *is set* **then**
19 $\quad\quad\quad$ Store positions of matches in A_long
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 **end**

Algorithm 2: Pseudocode for the V-PATCH filtering phase.

Once the vector registers are filled, the next step is to calculate the set of indexes for the filters. Note that every 2-byte input value maps to a specific *bit* in the filter, but the memory locations in the filter are addressable in *bytes*. A standard technique used in the literature [23, 16] is to perform a bit-wise right shift of the input value to the corresponding index in the filter. The remainder of the shift indicates which bit to choose from the ones returned. Having computed the indexes, we use them as arguments to the *gather* instruction that fetches the filter values at those locations (Algorithm 2, lines 9 and 13).

Regarding the number of *gather* instructions used, to optimize in latency,

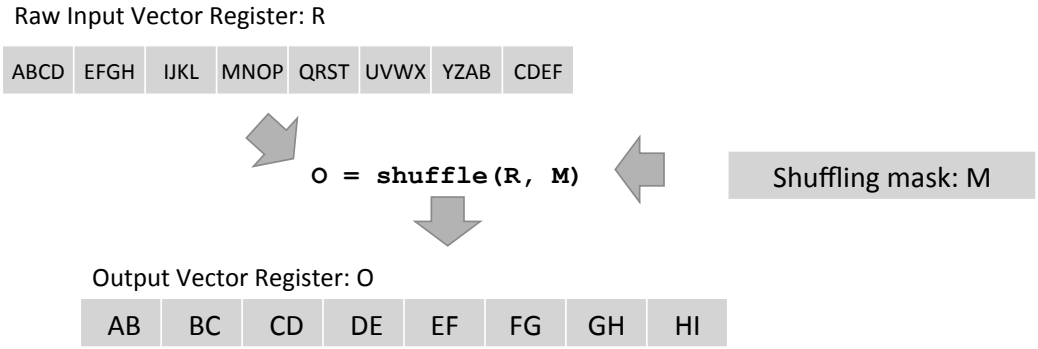


Figure 4: Input Transformation from consecutive characters to sliding windows of two characters.

note that the first two filters (lines 9 and 13) are specifically designed to use the same indexes for a given input value in *gather* but different base addresses for the filters. Thus, with the **filter merging** optimization where the filters are interleaved in memory (at the same base address), we can merge lines 9 and 13 into a single *gather*, to bring the information from both filters from memory simultaneously. This optimization is not shown in the pseudo-code but depicted in Figure 5, giving an example in which a single *gather* instruction fetches information from both filters. Using bit-wise operations we can choose one filter or the other, once the data is in the vector register.

If at least one of the W values has passed the second filter, they need to be further processed through the third filter. Remember that the third filter uses a window of four input characters as an index. Thus, we load a sliding window of four input characters in each vector element in the register (line 15) and create the hash values that we use as indexes in the third filter (lines 16-17).

Not all of the values in the vector register are useful; only the ones that passed the second filter need to be processed further by the third filter. This is a common challenge when vectorizing algorithms with conditional statements, since for different input we need to run different instructions. There are approaches [23] that manipulate the elements in the vector registers, so that they only operate on useful elements. For this particular algorithm, experiments with preliminary implementations showed that the cost of moving the elements in the registers out-weighted the benefits. Thus, we choose to speculatively perform the filtering on all the values and then mask out the ones that do not pass the second filter. In our evaluation (Section 6),

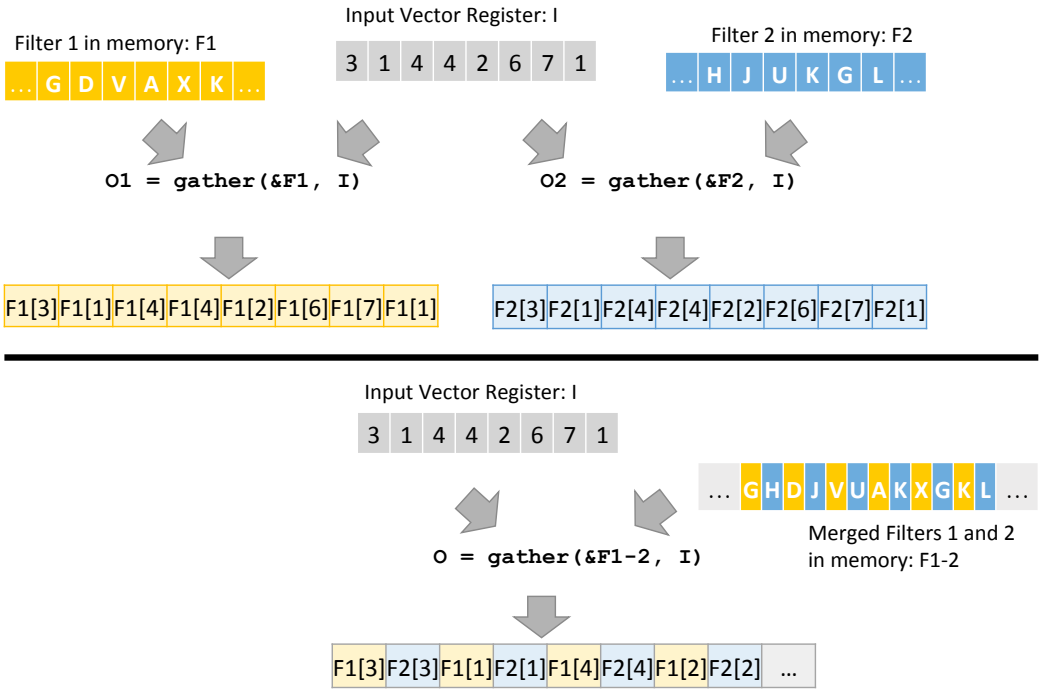


Figure 5: Figure describing the **filter merging** optimization. In the upper half, lookups on two filters require two gather invocations. Once the filters are merged in memory in the lower half, one gather brings information from both filters to the registers.

we observe that operating speculatively on all the elements is actually not a wasteful approach, especially with a large number of patterns to match.

As with the scalar algorithm, after a hit in the first or third filter we need to store the position of the input where a potential match occurred. We store the positions of the input that passed the filter from the set of W values in the register (lines 11 and 19). Here, we postpone the actual verification to avoid a potential costly mix of vectorized and scalar code, where the values from the vector registers need to be written to the stack and from there read into the scalar registers. Such a conversion can be costly and can negate any benefits we gain from vectorization [22].

Furthermore, to fully exploit the available instruction-level parallelism, we manually unroll the main loop of the algorithm by operating on two vectors (R_j) of W values instead of one, a technique that has proven to be efficient especially for SIMD code [23]. This has the benefit that, while the results of a *gather* on one set of W values are fetched from memory (line 9),

the pipeline can execute computations on the other set of values in parallel.

Scaling across multiple threads: The description of V-PATCH so far focuses on how to utilize data parallelism within each core using vector instructions, but we can easily extended them to use multiple threads. With respect to that, we inherit the easily parallelizable property from DFC. Contrary to, e.g. Aho-Corasick, that is inherently sequential, DFC (as well as S-PATCH and V-PATCH) can start processing from any point in the input stream. Based on that, the algorithms presented in this section can be parallelized by splitting the received input into equal chunks and distributing it across the available threads. Then, each thread processes its own chunk independently. The only corner case is when malicious patterns spawn across two different chunks: to remedy this we allow each thread to continue processing each neighbouring thread's chunk, for as along as the largest pattern in the pattern set. Usually, the size of the largest pattern is very small (323 bytes in our evaluation), compared to the size of the each chunk (several MB). In Section 6.7 we show that our algorithms can scale with the number of threads.

5. Performance Model

In order to better understand the runtime performance of the filter design we describe above, in this section, we introduce a simple model of the expected performance of the algorithm with respect to the number of patterns taken into account. We provide a model for both the scalar (S-PATCH) and the vectorized version (V-PATCH).

5.1. Usefulness

Our performance model is a useful tool to design and evaluate alternative filter architectures. As an example, for a given number of patterns, the model estimates the expected hit rate of the filters and the expected cost associated with filtering. Based on that, one can decide to add more filters in the design, or remove filters if their filtering ratio is low compared to the cost of accessing them. The model description that follows in this section refers to the filter design presented in Figure 2, but a similar analysis can be used for any other type of design.

5.2. Filter hit rates

We start by estimating the hit rate of the filters, then use these rates to derive the overall performance model. We assume, for now, that both

the input stream and the patterns are random. Then, if x is the number of patterns that are added to a filter, the probability that a bit in the filter is still zero is

$$p = (1 - \frac{1}{m})^x \tag{1}$$

where m is the size of the filter in bits (in the evaluation we use $m = 64K$ for all filters). This probability is derived by just considering the filter as a Bloom filter with a single hash function. In turn, the expected hit rate of a filter in the scalar case, i.e. the probability of accessing a single bit in the filter and finding it set to 1, is the complementary probability:

$$h(x) = 1 - p = 1 - (1 - \frac{1}{m})^x \tag{2}$$

Filter 1 in Figure 2 has a hit rate $h_1 = h(x_1)$ where x_1 is the number of patterns that are less than 4 bytes long. Note that, because filter 1 uses the first 2 bytes of the pattern as index, single-byte patterns need to be extended to 2 bytes. In order to do this, we create every possible combination of 2 byte characters starting with that single-byte pattern. For example, given the strings BC and A, we will set one bit at the index that corresponds to the position of BC and 256 bits on all indexes that start with A (AA, AB, AC etc.). As a result, x_1 accounts for all the patterns that are less than 4 bytes long and the number of extra patterns generated due to the presence of single-byte patterns.

Similarly, filter 2 in Figure 2 has a hit rate $h_2 = h(x_2)$, where x_2 is the number of patterns that are greater or equal to 4 bytes long. For filter 3, notice that: (i) it has the same size and number of patterns as filter 2, (ii) accessing filter 3 requires a hit in filter 2 (see Figure 2) and (iii) it uses a different hash function from filter 2, so a hit in filter 2 tells nothing about the probability of a hit in filter 3. Based on that, the overall probability of having a hit in filter 3 is $h_3 = (h_2)^2$.

Turning to the vectorized case, remember that we have a hit in the filter if *at least one* of the W elements in the register hits the filter. Thus, the hit rate h' of a filter in the vectorized case is:

$$h' = 1 - (1 - h)^W \tag{3}$$

since $(1 - h)^W$ is the probability of having W consecutive misses.

Figure 6 shows the expected hit rates of the filters in the scalar and vectorized case for a varying number of random patterns. Here we assume that the size of each pattern is uniformly distributed between 1 and 50 bytes.

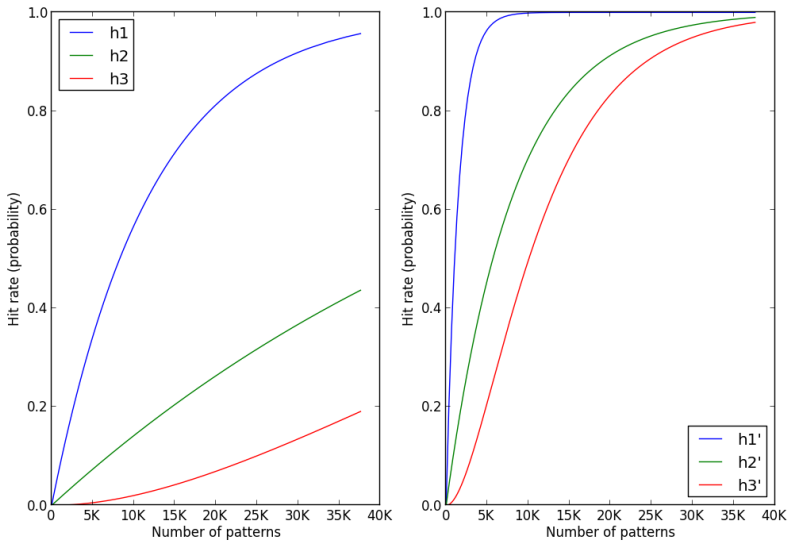


Figure 6: Expected hit rate for each filter in the scalar case (left) and the vectorized case (right).

5.3. Overall cost

Knowing the hit rates of the filters allows us to model the overall per-byte cost of the algorithm. We model the filtering and the verification phases separately.

For each byte of input processed by S-PATCH, we identify the following main operations that need to be performed in the filtering phase: (i) compute the indexes to filters 1 and 2 and access them, (ii) if there is a hit in filter 1, store the hit, (iii) if there is a hit in filter 2, compute the index for filter 3 and access it and (iv) if there is a hit in filter 3, store the hit. Those operations are the main factors in our model of the per-byte cost for the filtering phase of S-PATCH, which can be broken down as follows:

$$c_f = c_{1,2} + s_1 * h_1 + c_3 * h_2 + s_3 * h_3 \tag{4}$$

where $c_{1,2}$ and c_3 are the cost of computing the indexes and accessing for the first two ($c_{1,2}$) and the third filter (c_3) and s_1, s_3 are the cost of storing the indexes that produced a hit at filters 1 and 3, respectively. The cost of storing the hits is relatively small and we will exclude it from the model (but

Table 1: Estimated values (in cycles) for the constants involved in the model, for the Haswell platform, c.f. Section 6.

	$c_{1,2}$	c_3	$c'_{1,2}$	c'_3	V_{small}	V_{large}
Estimated value (cycles)	3.8	26.0	3.1	4.3	7.7	110.7

we will return to it in Section 6.4). Thus,

$$c_f = c_{1,2} + c_3 * h_2 \tag{5}$$

That leaves us with two constants that need to be computed, $c_{1,2}$ and c_3. We approximate these constants by measuring the cost for two numbers of patterns.

Similarly, the filtering cost for the vectorized case is

$$c'_f = c'_{1,2} + c'_3 * h'_2 \tag{6}$$

The cost of the verification phase is the same for both the scalar and the vectorized case. Remember that the algorithm reaches the verification phase when there is a hit on the first or the third filter. Verifying a hit involves a lookup in a hash table, the cost of which can be considered constant. Thus, the per-byte cost of verification can be modeled as follows:

$$c_v = c'_v = h_1 * V_{small} + h_3 * V_{large} \tag{7}$$

where V_{small}, V_{large} are the cost of the hash table lookups for verification of small and large patterns, respectively. Again, we approximate these two constants by measuring the cost of verification for two numbers of patterns.

In summary, the per-byte cost for S-PATCH is

$$c = c_f + c_v = c_{1,2} + c_3 * h_2 + h_1 * V_{small} + h_3 * V_{large} \tag{8}$$

and for V-PATCH:

$$c' = c'_f + c'_v = c'_{1,2} + c'_3 * h'_2 + h_1 * V_{small} + h_3 * V_{large} \tag{9}$$

The values we use for the constants are given in Table 1 (measured for the Haswell platform, c.f. Section 6). In Section 6 we evaluate the cost predicted by the model and show that it is accurate with respect to the one observed in practice.

6. Evaluation

In this section, we evaluate the benefits that our vectorization techniques bring to pattern matching algorithms. Our evaluation criteria are the processing throughput and the performance under varying number of patterns. We show the improvements of V-PATCH with both realistic and synthetic datasets, as well as with changing number of patterns. For a comprehensive evaluation, we compare the results from five different algorithms: the original Aho-Corasick ([15]; implementation directly taken from the Snort source code [18]), DFC (Choi et al. [16], summarized in Section 2.2), Vector-DFC (a direct vectorization of DFC done by us), S-PATCH (the scalar version of our algorithm, described in Section 4.1, that facilitates vectorization and addresses properties of realistic traffic that were not addressed before), and V-PATCH (the final vectorized algorithm described in Section 4.2).

6.1. Experimental setup

Systems: For the evaluation we use both Intel Haswell and Xeon-Phi. More specfically, the first system is an Intel Xeon E5-2695 (Haswell) CPU with 32KB of L1 data cache, 256KB of L2 cache and 35MB of L3 cache. The platform has 14 cores on a single socket, with up to 2 threads per core, using hyperthreading. We use the ICC compiler (version 16.0.3) with -O3 optimization under the operating system CentOS. Unless otherwise noted, the experiments in this section are run on this platform. The second system is the Intel Xeon-Phi 3120 co-processor platform. Xeon-Phi has 57 simple, in-order cores at 1.1 GHz each, with 512-bit vector processing units. Each core supports up to 4 threads with hyperthreading. The memory subsystem includes a L1 data cache and a L2 cache (32KB and 512KB respectively) private to each core, as well as a 6GB GDDR5 memory, but no L3 cache. We compile with ICC -O3 (version 16.0.3) under embedded Linux 2.6. We are only using Xeon-Phi in native mode as a co-processor. The next versions of Xeon-Phi are standalone processors, so the problem of processor-to-co-processor communication is alleviated. In the following experiments, we first focus on the speedup achieved by a single hardware thread, through vectorization, then we discuss experiments with multiple threads.

Patterns: We use two sets of patterns: a smaller one, named *S1*, consisting of approximately $2,500$ patterns that comes with the standard distribu-

tion of Snort[3] [24] – the de-facto standard for network intrusion detection systems – and a larger one, named *S2*, with approximately 20,000 patterns, that is distributed by `emergingthreats.net` The patterns affect the performance of the algorithm and this is analyzed in detail in Section 6.3.

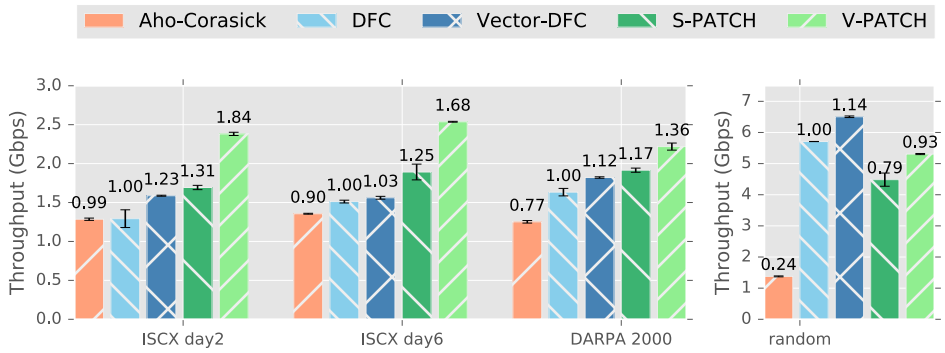
Data sets: In our evaluation, we use both real-world traces and synthetic data-sets. The real-world traces are the ICSX dataset [25, 26] (created to evaluate intrusion detection systems) and the DARPA intrusion detection dataset [27]. From ICSX, we randomly take 1GB of data from each of days 2 and 6 (thereafter named ICSX day 2 and ICSX day 6, respectively) and we also use 300MB of data from the DARPA 2000 capture. We are aware of the artifacts in the latter set, and the discussions in the community about its suitability for measuring the *detection capability* of intrusion detection systems [28]. In our experiments, we use it only for the purpose of comparing throughput between algorithms, allowing for future comparisons on a known dataset. The synthetic data set consists of 1GB of randomly generated characters.

An important point, considering the evaluation validity, is that, typically, not all the patterns are evaluated at the same time. In a Network Intrusion Detection System such as Snort, patterns are organized in groups, depending on the type of traffic they refer to. When traffic arrives in the system, the reassembled payload is matched only against patterns that are relevant (e.g. if the stream has HTTP traffic, it is checked against HTTP related patterns, as well as more general patterns that do not refer to a specific protocol or service). To evaluate our algorithm in a realistic setting, we also pair traffic with relevant patterns. Since, in our datasets, most of the traffic is HTTP [25], we focus on HTTP traffic and match it against the patterns that are applicable based on the rule definitions. A similar approach can be used for other protocols (e.g. DNS, FTP), but we focus on HTTP traffic as it typically dominates the traffic mix and many attacks use HTTP as a vector of infection.

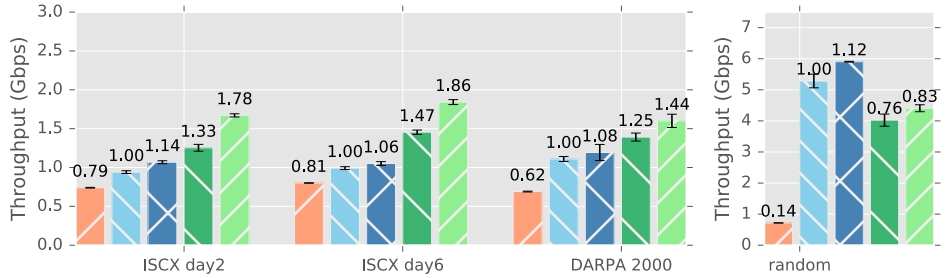
6.2. Overall Throughput

In this section we compare the overall performance between the different algorithms. Using the HTTP-related patterns of each set gives us 2K patterns from pattern set *S1* and 9K patterns from pattern set *S2*. All al-

[3]We used version 2.9.7 for our experiments.



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).

Figure 7: Performance comparison between the different algorithms for public and random data sets, on the Xeon platform.

gorithms count the number of matches. We use 10 independent runs of each experiment. We report the average throughput values, as well as standard deviation as error bars.

Figure 7a shows the throughput of all algorithms under realistic traffic traces and synthetic traces, when matched against the small pattern set (*S1*). In Figure 7b we use the bigger pattern set (*S2*). The numbers above the bars indicate the relative speedup compared to the original DFC algorithm.

We first discuss the results by only considering each pattern set and each traffic set separately. For realistic traffic traces, our vectorized implementation consistently outperforms the DFC algorithm by up to 1.86x (left parts of Figure 7), due to the parallelization we introduce in the filtering phase. The direct vectorization of the original DFC algorithm (Vector-DFC) has limited performance gain, because much of the running time of DFC is spent on verification and not filtering. This is the main motivation for introducing

a modified version of DFC, in Section 4.1, focused on improving the filtering phase. By treating small, frequently occurring patterns separately and by examining more information in the case of long patterns, S-PATCH outperforms the original by up to 1.47x. More importantly, it allows for much greater vectorization potential, since the biggest portion of the algorithm's running time is shifted to efficient filtering of the input, and verification is done much more seldom.

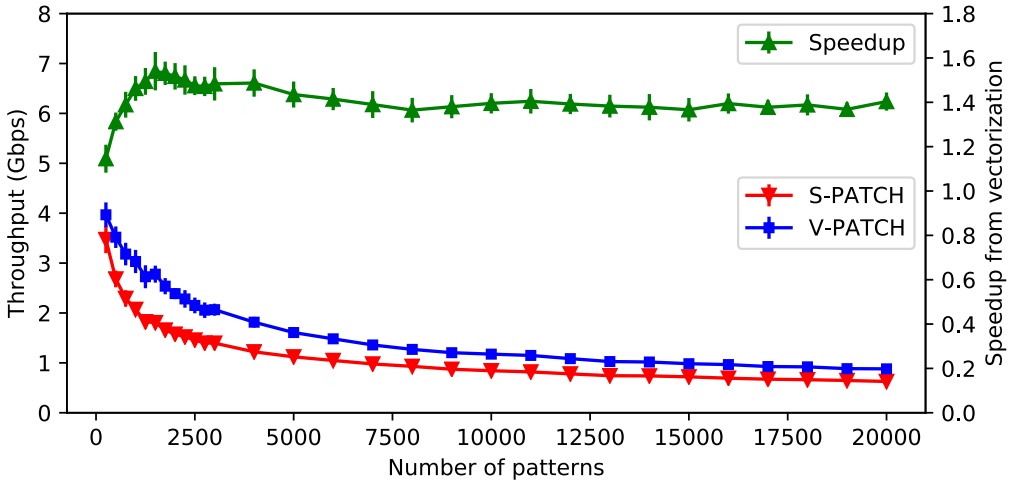
Next, we evaluate the impact of the size of the ruleset on the overall throughput (comparing Figure 7a with Figure 7b). The overall throughput of the algorithms decreases, since the input is more likely to match and identifying every match consumes extra cycles. The performance of Aho-Corasick, in particular, decreases by more than 40%, because the extra patterns greatly increase the size of the state machine. The rest of the algorithms experience a 23-34% drop in performance.

It is important to note that the performance gain of the algorithms (DFC versus Aho-Corasick, V-PATCH versus DFC) is influenced by the input as follows: when feeding the algorithms a data set that contains random strings, DFC significantly outperforms AC (right part of Figure 7). In this case, we do not expect to find many matches in the input and the filtering phase will quickly filter out up to 95% of the input. This is also the reason why the modified versions of the algorithm (S-PATCH and V-PATCH) perform less efficiently compared to what they do in the different input scenarios; the design of the two separate filters as described in Section 4 shows its benefits in more realistic traffic mixes. In turn, this poses interesting questions for the future in how to best design the filters based on the expected traffic mix. Still, the vectorized versions provides speedups over the scalar ones.

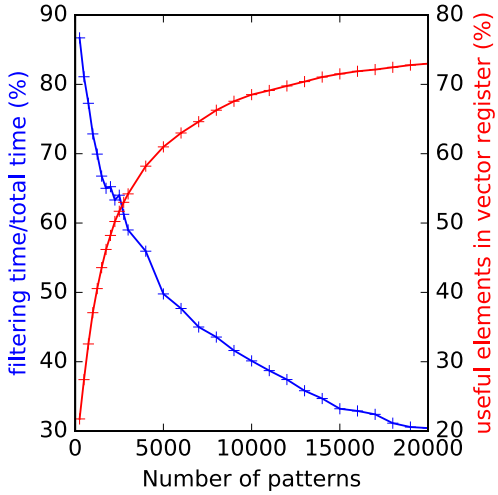
6.3. The effects of the number of patterns

As shown in Section 6.2, it is important to account for the actual traffic mix the algorithms are expected to run upon when designing the filtering stage, as it has a large impact on the performance. As new threats emerge, more malicious patterns are introduced and the performance of the algorithm must adapt to that change.

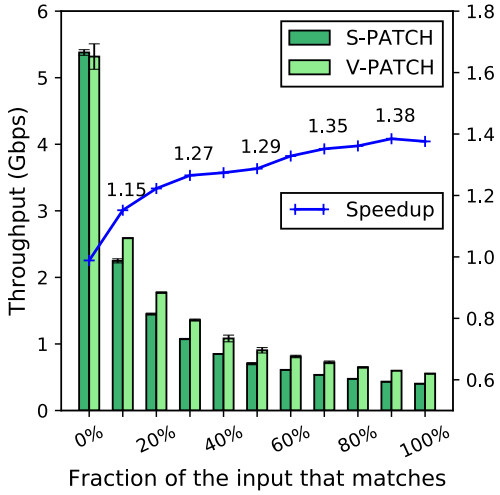
We measure the effects of the number of patterns on the two best performing algorithms and summarize the results in Figure 8a, also including the overall speedup of V-PATCH compared to S-PATCH. In this experiment, we randomly select the number of patterns from the complete set *S2* (20,000 patterns) in order to test our algorithms with as many patterns



(a) Throughput as the number of patterns increases.



(b) Filtering to verification ratio and vectorization efficiency.



(c) Speedup from vectorization, as the numbers of matches in the input increases.

Figure 8: Figure a) compares the scalar and vectorized versions of our approach, as the number of patterns increases. Figure b) shows the filtering-to-verification ratio (left axis), as well as the average number of useful elements in the vector registers after filter 2 (right axis), as the number of patterns increases. Figure c) compares the scalar and vectorized approach, as the fraction of matches in the input increases.

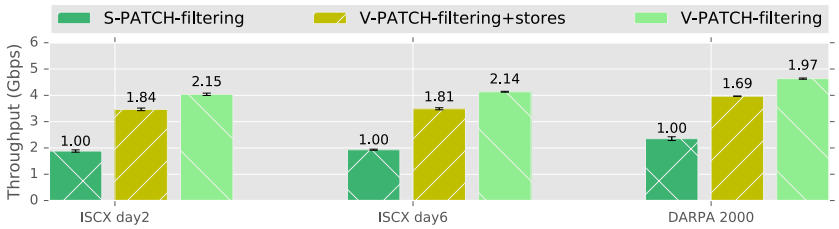
as possible. V-PATCH consistently performs better compared to S-PATCH, regardless of the number of patterns considered. Observe that:

- As the number of patterns increases, so does the input fraction that passes the filters. This causes the verification part, which is not vectorized, to take up more of the running time, essentially reducing the parallel portion and, by Amdahl's law [29], the benefit of vectorization. The portion of the running time spent in filtering, over the total running time is shown in Figure 8b (blue line).

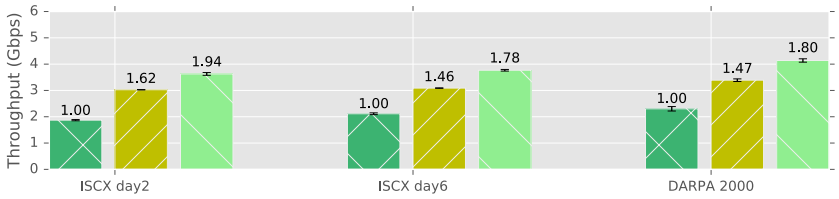
- As the number of patterns increases, the vectorization of the filtering becomes more efficient. Remember that V-PATCH will proceed with the third filter if at least one of the values in the vector register block passes the second filter. With a small number of patterns, we will seldom pass the second filter. When we do, it is likely we only have a single match, meaning that the rest of the values in the register are disabled and any computation performed for those values is wasteful work. Increasing the number of patterns results in more potential matches in the second filter and, as a consequence, less disabled values for the third filter and thus more useful work. In Figure 8b (red line) we measure this effect and show the average number of useful items inside the vector register every time we reach the third filter. Clearly, with an increasing number of patterns, the vectorization is performed mainly on useful data and therefore becomes more efficient.

- The two trends essentially cancel each other out, keeping the overall performance benefit of V-PATCH compared to S-PATCH constant after a point (Figure 8a), even though the optimized filtering gradually becomes a smaller part of the total running time. Eventually, the vector registers will always be full and we will not benefit from having more patterns. At this point the relative performance will stay constant. Our results indicate that this point is far beyond the number of patterns that current intrusion detection systems utilize.

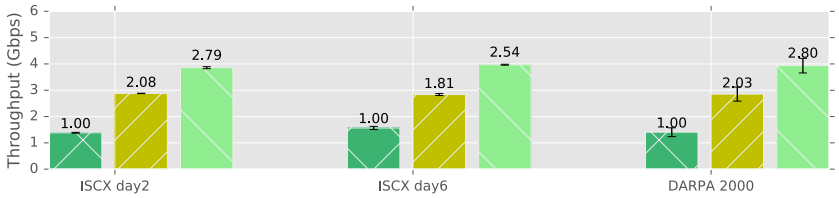
- A similar effect is observed when we keep the number of patterns constant, but increase the amount of matches in the dataset (Figure 8c). For this experiment, we created a synthetic input that contains increasingly more patterns, randomly selected from a ruleset of $2,000$ patterns. As more matching strings are inserted into the input, our vec-



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).



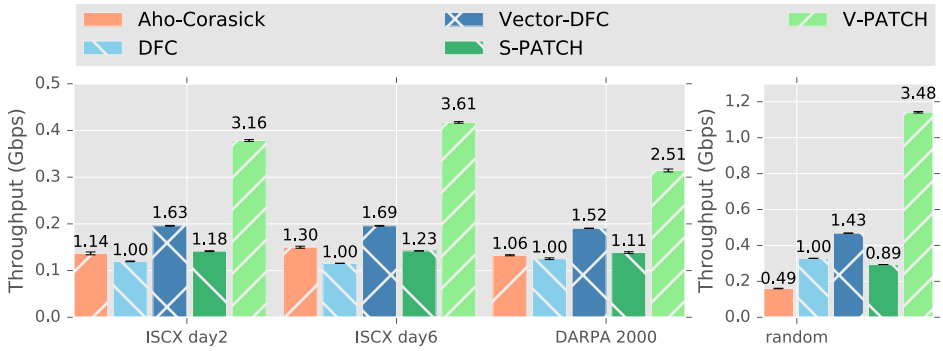
(c) Full pattern-set (20K).

Figure 9: Measuring the performance of the filtering part only. V-PATCH-filtering+stores includes the cost of storing the results of the filtering phase to temporary arrays.

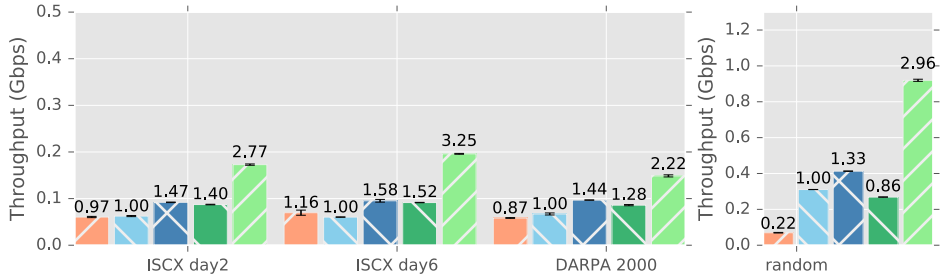
torized portion of the algorithm becomes more efficient and the relative speedup compared to the scalar version slowly increases.

6.4. Filtering Parallelism

In this section, in order to gain better insights about the benefits of vectorization, we measure the speedup gained in the filtering part in isolation. Figure 9 compares the filtering throughput of the scalar S-PATCH and V-PATCH, for pattern sets S1, S2, as well as the full pattern set (20K patterns). In the same figure, we also report the performance of the vectorized filtering, where we exclude the cost of storing the matches in the filtering phase in the temporary arrays. As we can see from the graph, the throughput of the filtering part is increased by up to a factor of 1.84x, on the small pattern set.



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).

Figure 10: Performance comparison between the different algorithms for public and random data sets on the *Xeon-Phi platform*.

Storing the matches of the filtering part in arrays comes with a cost; when it is removed, performance increases up to 2.15x for small pattern sets and up to 2.80x for the full pattern set. Even though there is a small decrease at the pattern set with 9K patterns (Figure 9b), the relative speedups of vectorized filtering increase with the number of patterns (Figure 9c).

6.5. Changing the vector length: Results from Xeon-Phi

We have also evaluated the effectiveness of our approach on an architecture with a wider vector processing pipeline. The Xeon-Phi [9] co-processor from Intel supports vector instructions that operate on 512-bit registers, thus able to perform two times more operations in parallel, in the filtering phase.

Figure 10 summarizes the results from Xeon Phi, where the experiments are identical with those described in Section 6.2. Note that we report the throughput of a single Xeon-Phi thread. V-PATCH takes advantage of the

wider vector registers and outperforms the original scalar DFC algorithm, up to a factor of 3.6x on real data and 3.5x on synthetic random data.

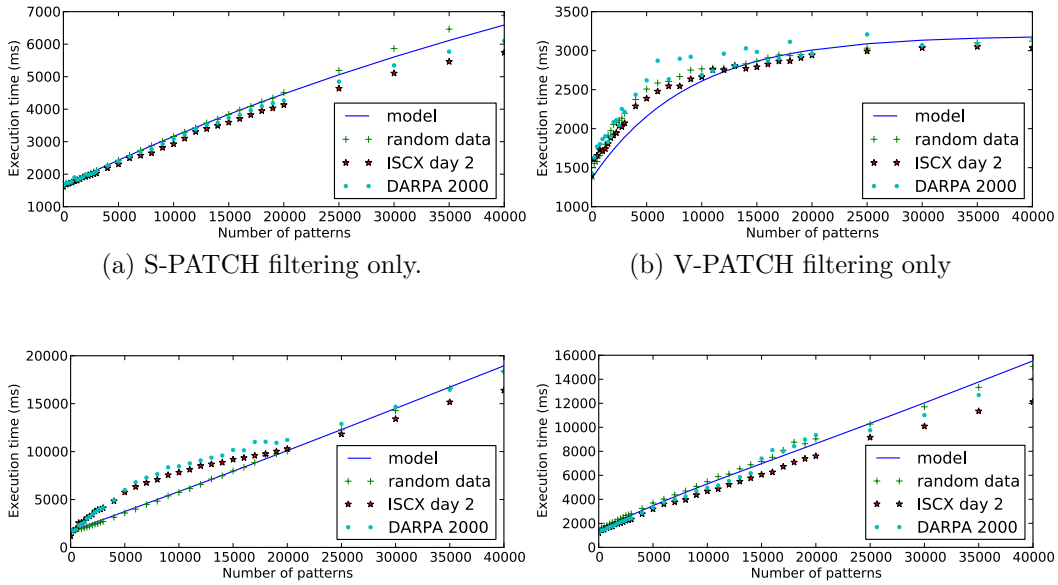
As Xeon-Phi threads have much slower clock (1.1 GHz) and the pipeline is less sophisticated (e.g. there is no out-of-order execution), it is not surprising that the absolute throughput sustained by a single Phi thread is smaller than that of the single thread performance of the Xeon platform used in the previous experiments. When dealing with multiple streams in parallel, due to the higher degree of parallelism, the aggregated gain will naturally be higher, as indicated later in Section 6.7.

An interesting observation is that the DFC algorithm is sometimes slightly slower than AC on real data, where the number of matches in the input is significantly higher. In the original DFC algorithm, the filters are small and can easily fit L1 or L2 cache, and the hash tables containing the patterns are bigger, but still expected to fit L3 cache. In Xeon-Phi there is no L3 cache, so accesses to the hash tables in the verification phase are typically served by the device memory, negating the benefits of cache locality that is part of the main idea of the algorithm. Nonetheless, our *improved filtering* design reduces the number of times we resort to verification and access the device memory, thus resulting in 1.1x-1.5x increased throughput on realistic traffic, compared to the original DFC design.

6.6. Model evaluation

In this section, we evaluate the accuracy of our analytical model presented in Section 5. In the following experiments, we randomly generate up to 40K patterns and use different data sets, both real and synthetic. We show the normalized execution time for S-PATCH and V-PATCH, along with the cost predicted by the model.

Figures 11a and 11b show the cost of filtering for S-PATCH and V-PATCH, respectively. The figures show both the cost predicted by the model (given by Equations 5 and 6) as well as the cost measured using real and synthetic data. As predicted by the model, the cost of filtering for both versions is mostly affected by the hit rate of filter 2 (see also Figure 6). The cost of S-PATCH increases with the number of patterns, while the cost of V-PATCH flattens quickly (in this case, the hit rate of filter 2 is already close to 90% for more than 20K patterns and the vector registers are filed with mostly useful elements). Notice the different range in the vertical axis between S-PATCH and V-PATCH and the fact that, as the model predicts, the filtering part



(a) S-PATCH filtering only.

(b) V-PATCH filtering only

(c) S-PATCH filtering and verification.

(d) V-PATCH filtering and verification

Figure 11: Real and predicted performance of S-PATCH and V-PATCH for different number of patterns.

of V-PATCH is much faster than that of S-PATCH across any number of patterns.

Similar to the above, Figures 11c and 11d show the total cost (in terms of execution time), including the cost of verification. The total cost for both follows an almost linear curve and is mostly dominated by the cost of verification, as predicted by the model (given by Equations 8 and 9). Since the model is fitted to random data, it predicts the cost of processing random data more closely compared to using realistic data (ISCX and DARPA data sets) where the traffic distribution is different. In this case of realistic data there is deviation from the model at around ten thousands patterns for the case of S-PATCH. Surprisingly, such deviation is not present for the case of V-PATCH. Also notice that, in most cases, processing real traffic is slightly faster than what is predicted by the model, most likely due to the different distribution of traffic.

Alternative filter designs: Having an accurate model to predict the overall performance of our algorithms allows us to easily evaluate different filtering

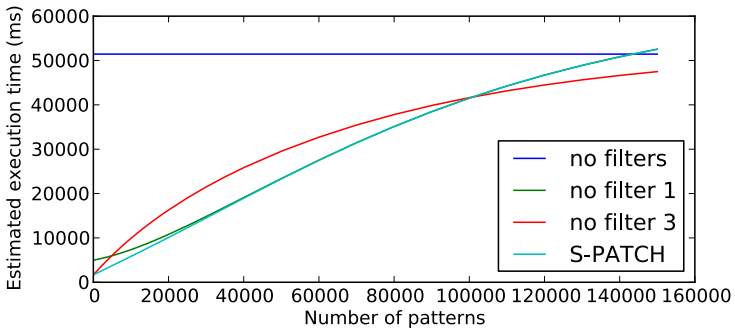
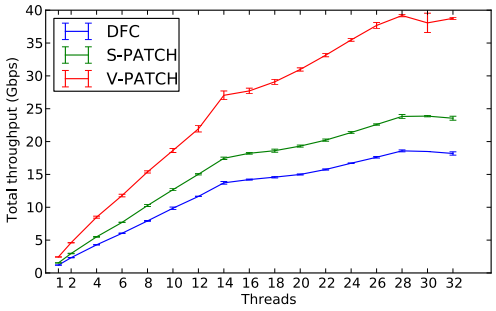


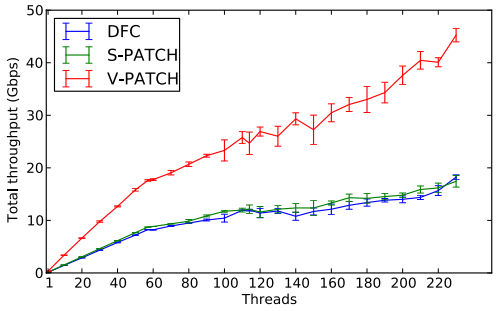
Figure 12: Prediction of the execution time of different filtering designs for S-PATCH, including designs where one or several of the filters are removed. Note the increased maximum number of patterns used in the horizontal axis.

architectures than the one we use for S-PATCH and V-PATCH (see Figure 2). We alter the model from Section 5 to predict a series of alternative designs, namely designs where we remove: (i) the filter for small patterns (Filter 1), (ii) one of the filters for long patterns (e.g. Filter 3) or (iii) all filtering whatsoever. By altering the model to cover these alternative designs, we can predict if, and at what number of patterns, it is beneficial to change our filtering design.

In Figure 12 we include the expected total execution time for 1GB of random data as predicted by the original model for S-PATCH, as well as the predictions for the alternative filtering designs discussed above. Note that we have extended the x-axis (number of patterns) to capture the trends at very large numbers of patterns, much larger than what is typically used in NIDS. Compared to our design (S-PATCH), removing Filter 1 has a small impact which is noticeable when less than twenty thousand patterns are used. Removing Filter 3 has initially a negative effect on performance, but the model predicts that it is a preferable choice when more than one hundred thousand patterns are used. This is reasonable since, when using so many patterns, filters are likely to be fully populated and have high hit-rates. In this case, the overhead of accessing the filter is not compensated by reducing the times we reach verification. If we remove all filters, we go to expensive verification for every input byte and the cost is prohibitively high, expect for the case of using more than one hundred and forty thousand patterns and all the filters are saturated. The trends also indicate that, for the number of patterns that are typically used in NIDS (one to ten thousand patterns)



(a) Parallel execution on the Haswell platform.



(b) Parallel execution on the Xeon-Phi platform.

Figure 13: Parallel execution on the Haswell and Xeon-Phi platforms.

our original filtering design is a good choice, validating the design choices explained in Section 4.1. The respective alternative designs for V-PATCH follow trends similar to the ones in Figure 12.

6.7. Parallel execution

The experiments presented so far focus on the data parallelism achieved within a single thread, i.e. using vectorization and data parallelism within each core. In this section, we present experiments from a multi-threaded execution and demonstrate the scalability of our approach. As already mentioned in Section 4.2, we can easily parallelize DFC, S-PATCH and V-PATCH by splitting the available input in equal chunks. Nonetheless, it is important to evaluate the scalability of algorithms using multiple threads to show the effect of the underlying architecture, e.g., resource sharing under hyper-threading.

For the following experiments, we used the ISCX day 2 data set and the S1 pattern set of 2K patterns. We split the input evenly across the available threads and report the total achieved throughput. We do not include the Aho-Corasick algorithm because it is inherently sequential. We experiment on both the Haswell platform (14 cores, 28 threads) and the Xeon-Phi platform (57 cores, 228 threads). In all cases, our thread placement policy is to spread threads as much as possible, i.e. we first place each thread in each own core, then start placing up to two threads per core, etc.

Figures 13a and 13b show the results from the Haswell and the Xeon-Phi platforms respectively. In both platforms, all algorithms scale linearly

while there is only one thread per core (up to 14 threads for Haswell and 57 threads for Xeon-Phi). After that, the scaling factor decreases, since threads that reside on the same core must share resources, such as parts of the execution units and the caches. For the case of the Haswell platform, we have also included tests where we spawn more software threads than the available hardware threads (over-subscription) and validate that we cannot get any more performance benefit. Nonetheless, all algorithms benefit from using the available thread-level parallelism in the system. V-PATCH achieves up to 40 and 45 Gbps on the Haswell and Xeon-Phi platforms respectively.

7. Other related Work

7.1. Pattern matching algorithms

Pattern matching has been an active field of research for many years and there are numerous proposed approaches. Aho-Corasick, explained before in Section 2.1 is one of the fundamental algorithms in the fields. There are variants of Aho-Corasick that decrease the size of the state transition table (for example [30]) by changing the way it is mapped in memory, but they come at an increased search cost, compared to the standard version of Aho-Corasick used in our evaluation. Other approaches apply heuristics that enable the algorithm to skip some of the input bytes without examining them at all, such as Wu-Manber [31] where a table is used to store information of how many bytes one can skip in the input. The main issue with these approaches is that they perform poorly with short patterns. For the problem domain investigated here, the patterns can be of any length and the algorithm must handle all of them gracefully. Moreover, in both Aho-Corasick and Wu-Manber algorithms, there is no data parallelism because there are dependencies between different iterations of the main loop over the input.

Recent algorithms [16, 17] follow a different idea: Using small data structures that hold information from the patterns (directly addressable bitmaps in the case of [16], Bloom filters in the case of [17]), they quickly filter out the biggest parts of the input that will not match any patterns and fallback to expensive verification when there is an indication for a match. Our work is inspired by this family of algorithms, showing how they can be modified to perform better under realistic traffic and gain significant benefit from vectorization.

7.2. Regular expression matching

Apart from exact signature matching, intrusion detection systems also employ regular expression matching to detect attacks, because they offer more flexibility when describing the patterns. Regular expression matching usually utilizes finite automata, either deterministic (DFA) or non-deterministic (NFA). DFA's are fast, because every byte of input leads to only one state and their search complexity is O(n). However, the size of the state machine can grow exponentially with the number of regular expressions [32]. NFA's, on the other hand, construct a significantly smaller state in memory, but the search time is increased, because the state machine needs to evaluate several paths before finding a match. There has been significant work trying to find a compromise between search time and memory use (for example [33]). Because regular expression matching is generally slow, Snort, a widely used NIDS, first applies exact pattern matching on the sub-strings that a regular expression contains, so most of the regular expressions do not have to be considered. The same approach is also followed in many proposed algorithms that target antivirus systems [34]. Thus, by improving the performance of exact pattern matching, we increase also the effectiveness of regular expression matching.

7.3. SIMD approaches to pattern matching

Even though pattern matching algorithms are characterized by random access patterns, SIMD approaches have been used before for pattern matching, especially in the field of regular expression matching. HyperScan [35] is a mature pattern matching framework that heavily relies on vector instructions for regular expression and fixed string pattern matching. Mytkowicz et al. [12] enumerate all the possible state transitions for a given byte of input to break data dependencies when traversing the DFA. Then they use the *shuffle* instruction to implement gathers and to compute the next set of states in the DFA. The algorithm is applied on the case where the input is matched against a single regular expression with a few hundreds of states and does not scale for the case of multiple pattern matching where we need to access thousands of states for every byte of input. Sitaridi et al. [13] use the same hardware gathers as we do, but apply them on database applications where the multiple, independent strings need to be matched against a single regular expression. There have been approaches that use other SIMD instructions for multiple exact pattern matching, but have constraints that make them impractical for the case of Network Intrusion Detection. Faro

et al. [36] create fingerprints from patterns and hash them, but they require that the patterns are long, which is not always true for the typical set of patterns found e.g. in Snort.

7.4. Other architectures

Outside the range of approaches that target commodity hardware, there is rich literature on network intrusion detections systems that are customised for specific hardware. For example, SIMD approaches that target DFA-based algorithms have been applied on the Cell processor [37], as well as FPGAs [38, 39, 40]. Most notably, Graphics Processing Units (GPUs) are a popular target platform for pattern matching applications. GPUs are highly parallel architectures and are typically a good match for algorithms that are easily parallelizable, such as pattern matching. Lin et al. [41] present a parallelizable version of Aho-Corasick that removes the failure transitions (transitions taken in the state machine when a pattern is only partially matched). The algorithms begins the state-machine traversal at every input byte, in parallel. Bellekens et al. [42] compress the size of Aho-Corasick's state machine to reduce the communication cost between the CPU and the GPU. Aragon et al. [43] experiment with pattern matching on embedded GPUs that share the same physical memory as the CPU. Kouzinopoulos and Margaritis also experiment with pattern matching algorithms on GPUs and apply them on genome sequence analysis [39].

There is also significant work on GPUs that addresses pattern matching as part of a Network Intrusion Detection System. Vasiliadis et al. [38] build a GPU-based intrusion detection system that uses Aho-Corasick as the core pattern matching engine. Go et al. [44] use integrated GPUs and show that they are successful platforms for packet processing and Network Intrusion Detection. Jahmsed et al. [45] present Kargus, a custom NIDS that uses multiple GPUs and CPU cores. Papadogiannaki et al. [46] present a similar system and enhance it with a scheduler that dynamically decides the placement of packet processing tasks.

GPU parallelization has many similarities with vectorization; in fact GPUs offer more parallelism that can hide memory latencies. At the same time, it introduces additional challenges e.g. long latencies when transferring data between the host and the GPU. In this work we utilize vector pipelines that are already part of modern commodity architectures. Moreover, vectorization with CPUs requires careful algorithmic design that makes use of

caches and advanced SIMD instructions. A main part of our work is showing how this problem can be tackled for the case of intrusion detection.

8. Conclusion

In this paper, we address the problem of multiple pattern matching and present an efficient algorithm that utilizes the architectural features of commodity hardware to improve the processing throughput of Network Intrusion Detection Systems or other similar applications that employ pattern matching, e.g. antivirus systems. Specifically we introduce V-PATCH, a cache efficient filtering design, coupled with modern vectorization techniques that allow data parallelism within each processing core. We also provide an analytical model for our algorithm that predicts the expected performance and can be used to create and evaluate new designs on-the-fly.

We thoroughly evaluate V-PATCH and its algorithmic design with both open data sets of real-world network traffic and synthetic ones in the context of network intrusion detection. Our results on Haswell and Xeon-Phi show a speedup of 1.8x and 3.6x, respectively compared to the state of the art and a speedup of more than 2.3x over Aho-Corasick, a widely used algorithm in today's Intrusion Detection Systems. We also show that our approach can scale across many cores, achieving up to 40 and 45 Gbps processing throughput on the Haswell and Xeon-Phi platforms, respectively. Our experimental study provides fine-grained insights on different scenarios, including stress-tests under malicious traffic and thousands of malicious patterns. Finally, we show that our analytical model closely follows the experimental results and can thus be used as a valuable tool to create new filtering designs.

Acknowledgements

The research leading to these results has been partially supported by the Swedish Energy Agency under the program Energy, IT and Design, the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC and the project LoWi, by the Swedish Research Council (VR) through the project ChaosNet, and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

References

[1] C. Stylianopoulos, M. Almgren, O. Landsiedel, M. Papatriantafilou, Multiple pattern matching for network security applications: Acceleration through vectorization, in: 2017 46th International Conference on Parallel Processing (ICPP), 2017, pp. 472–482 (Aug 2017). doi:10.1109/ICPP.2017.56.

[2] D. Knuth, J. Morris, Jr., V. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (2) (1977) 323–350 (1977). arXiv:https://doi.org/10.1137/0206024, doi:10.1137/0206024.
URL https://doi.org/10.1137/0206024

[3] R. S. Boyer, J. S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (1977) 762–772 (Oct. 1977). doi:10.1145/359842.359859.
URL http://doi.acm.org/10.1145/359842.359859

[4] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, Generating realistic workloads for network intrusion detection systems, SIGSOFT Softw. Eng. Notes 29 (1) (2004) 207–215 (Jan. 2004). doi:10.1145/974043.974078.
URL http://doi.acm.org/10.1145/974043.974078

[5] J. B. D. Cabrera, J. Gosar, W. Lee, R. K. Mehra, On the statistical distribution of processing times in network intrusion detection, in: 2004 43rd IEEE Conf. on Decision and Control (CDC), Vol. 1, 2004, pp. 75–80 Vol.1 (Dec 2004). doi:10.1109/CDC.2004.1428609.

[6] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, R. Boutaba, Network function virtualization: State-of-the-art and research challenges, IEEE Communications Surveys & Tutorials 18 (1) (2015) 236–262 (2015).

[7] Y. Li, M. Chen, Software-defined network function virtualization: a survey, IEEE Access 3 (2015) 2542–2553 (2015).

[8] J. Kurose, K. Ross, Computer networks: A top down approach featuring the internet, Peorsoim Addison Wesley (2010).

[9] Intel Xeon Phi product family, http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html, accessed: 2016-12-10 (2016).

[10] Intel vectorization tools, https://software.intel.com/en-us/articles/intel-vectorization-tools, accessed: 2016-12-10 (2015).

[11] The importance of vectorization for Intel Many Integrated Core Architecture (Intel MIC architecture), https://software.intel.com/en-us/articles/the-importance-of-vectorization-for-intel-many-integrated-core-architecture-intel-mic, accessed: 2016-12-10 (2013).

[12] T. Mytkowicz, M. Musuvathi, W. Schulte, Data-parallel finite-state machines, in: Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, ACM, New York, NY, USA, 2014, pp. 529–542 (2014). doi:10.1145/2541940.2541988.
URL http://doi.acm.org/10.1145/2541940.2541988

[13] E. Sitaridi, O. Polychroniou, K. A. Ross, SIMD-accelerated regular expression matching, in: Proc. of the 12th Int. Workshop on Data Management on New Hardware, DaMoN '16, ACM, 2016, pp. 8:1–8:7 (2016). doi:10.1145/2933349.2933357.
URL http://doi.acm.org/10.1145/2933349.2933357

[14] P. Jiang, G. Agrawal, Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation, in: Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '17, ACM, New York, NY, USA, 2017, pp. 179–191 (2017). doi:10.1145/3018743.3018760.
URL http://doi.acm.org/10.1145/3018743.3018760

[15] A. V. Aho, M. J. Corasick, Efficient string matching: An aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340 (Jun. 1975). doi:10.1145/360825.360855.
URL http://doi.acm.org/10.1145/360825.360855

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

URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/choi

[17] I. Moraru, D. G. Andersen, Exact pattern matching with feed-forward bloom filters, J. Exp. Algorithmics 17 (2012) 3.4:3.1–3.4:3.18 (Sep. 2012). doi:10.1145/2133803.2330085.
URL http://doi.acm.org/10.1145/2133803.2330085

[18] Snort rules and IDS software download, https://www.snort.org/downloads, accessed: 2016-12-10 (2016).

[19] Scaling CloudFlare's massive WAF, https://www.scalescale.com/scaling-cloudflares-massive-waf/, accessed: 2016-12-10 (2014).

[20] Gather Scatter operations, http://insidehpc.com/2015/05/gather-scatter-operations/, accessed: 2016-12-10 (2015).

[21] O. Polychroniou, A. Raghavan, K. A. Ross, Rethinking SIMD vectorization for in-memory databases, in: Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD '15, ACM, 2015, pp. 1493–1508 (2015). doi:10.1145/2723372.2747645.
URL http://doi.acm.org/10.1145/2723372.2747645

[22] J. Hofmann, J. Treibig, G. Hager, G. Wellein, Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips, in: Proc. of the 2014 Workshop on Programming Models for SIMD/Vector Processing, WPMVP '14, ACM, New York, NY, USA, 2014, pp. 57–64 (2014). doi:10.1145/2568058.2568068.
URL http://doi.acm.org/10.1145/2568058.2568068

[23] O. Polychroniou, K. A. Ross, Vectorized Bloom filters for advanced SIMD processors, in: Proc. of the Tenth Int. Workshop on Data Management on New Hardware, DaMoN '14, ACM, New York, NY, USA, 2014, pp. 6:1–6:6 (2014). doi:10.1145/2619228.2619234.
URL http://doi.acm.org/10.1145/2619228.2619234

[24] M. Roesch, Snort - lightweight intrusion detection for networks, in: Proc. of the 13th USENIX Conf. on System Administration, LISA '99, USENIX Association, Berkeley, CA, USA, 1999, pp. 229–238 (1999).
URL http://dl.acm.org/citation.cfm?id=1039834.1039864

[25] A. Shiravi, H. Shiravi, M. Tavallaee, A. A. Ghorbani, Toward developing a systematic approach to generate benchmark datasets for intrusion detection, Computers & Security 31 (3) (2012) 357 – 374 (2012). doi:http://dx.doi.org/10.1016/j.cose.2011.12.012.
URL http://www.sciencedirect.com/science/article/pii/S0167404811001672

[26] UNB ISCX intrusion detection evaluation dataset, https://www.unb.ca/cic/datasets/ids.html, accessed: 2016-12-10 (2012).

[27] DARPA intrusion detection data sets, https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-datasets, accessed: 2016-12-10 (2012).

[28] M. V. Mahoney, P. K. Chan, An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection, in: Int. Workshop on Recent Advances in Intrusion Detection, Springer, 2003, pp. 220–237 (2003).

[29] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proc. of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), ACM, New York, NY, USA, 1967, pp. 483–485 (1967). doi:10.1145/1465482.1465560.
URL http://doi.acm.org/10.1145/1465482.1465560

[30] M. Norton, Optimizing pattern matching for intrusion detection, Sourcefire, Inc., Columbia, MD (2004).

[31] S. Wu, U. Manber, A fast algorithm for multi-pattern searching, Tech. Rep. TR-94-17, University of Arizona. Department of Computer Science (1994).

[32] G. Berry, R. Sethi, From regular expressions to deterministic automata, Theoretical computer science 48 (1986) 117–126 (1986).

[33] R. Smith, C. Estan, S. Jha, S. Kong, Deflating the big bang: fast and scalable deep packet inspection with extended finite automata, in: ACM SIGCOMM Computer Communication Review, Vol. 38, ACM, 2008, pp. 207–218 (2008).

[34] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, D. G. Andersen, SplitScreen: Enabling efficient, distributed malware detection, Journal of Communications and Networks 13 (2) (2011) 187–200 (Apr. 2011). doi:10.1109/JCN.2011.6157418.

[35] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, H. Zhu, Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 631–648 (2019).
URL https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[36] S. Faro, M. O. Külekci, Fast Multiple String Matching Using Streaming SIMD Extensions Technology, Springer, Berlin, Heidelberg, 2012, pp. 217–228 (2012). doi:10.1007/978-3-642-34109-0_23.
URL http://dx.doi.org/10.1007/978-3-642-34109-0_23

[37] D. P. Scarpazza, O. Villa, F. Petrini, Peak-performance DFA-based string matching on the Cell processor, in: 2007 IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 1–8 (March 2007). doi:10.1109/IPDPS.2007.370634.

[38] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, S. Ioannidis, Gnort: High Performance Network Intrusion Detection Using Graphics Processors, Springer, Berlin, Heidelberg, 2008, pp. 116–134 (2008). doi:10.1007/978-3-540-87403-4_7.
URL http://dx.doi.org/10.1007/978-3-540-87403-4_7

[39] C. S. Kouzinopoulos, K. G. Margaritis, String matching on a multicore GPU using CUDA, in: Informatics, PCI'09. 13th Panhellenic Con. on, IEEE, 2009, pp. 14–18 (2009).

[40] I. Sourdis, D. Pnevmatikatos, Pre-decoded CAMs for efficient and high-speed nids pattern matching, in: Field-Programmable Custom Computing Machines, FCCM 2004. 12th Annual IEEE Symposium on, IEEE, 2004, pp. 258–267 (2004).

[41] C. H. Lin, C. H. Liu, L. S. Chien, S. C. Chang, Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs, IEEE

Transactions on Computers 62 (10) (2013) 1906–1916 (Oct 2013). doi:10.1109/TC.2012.254.

[42] X. J. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, T. Kirkham, A highly-efficient memory-compression scheme for gpu-accelerated intrusion detection systems, in: Proceedings of the 7th International Conference on Security of Information and Networks, ACM, arXiv, 2014, p. 302 (2014).

[43] E. Aragon, J. M. Jiménez, A. Maghazeh, J. Rasmusson, U. D. Bordoloi, Pattern matching in opencl: Gpu vs cpu energy consumption on two mobile chipsets, in: Proceedings of the International Workshop on OpenCL 2013 & 2014, IWOCL '14, ACM, New York, NY, USA, 2014, pp. 5:1–5:7 (2014). doi:10.1145/2664666.2664671.
URL http://doi.acm.org/10.1145/2664666.2664671

[44] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, K. Park, APUNet: Revitalizing GPU as Packet Processing Accelerator, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA, 2017, pp. 83–96 (2017).
URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[45] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, 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, ACM, New York, NY, USA, 2012, pp. 317–328 (2012). doi:10.1145/2382196.2382232.
URL http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2382196.2382232

[46] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, S. Ioannidis, Efficient software packet processing on heterogeneous and asymmetric hardware architectures, IEEE/ACM Transactions on Networking 25 (3) (2017) 1593–1606 (June 2017). doi:10.1109/TNET.2016.2642338.