Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs

Citation for the original published paper (version of record):
Stylianopoulos, C., Kindström, S., Almgren, M. et al (2019). Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs. ACM International Conference Proceeding Series, 2019-January: 17-27. http://dx.doi.org/10.1145/3359789.3359811

Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs

Charalampos Stylianopoulos
chasty@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Simon Kindström
simonki@student.chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Magnus Almgren
magnus.almgren@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Olaf Landsiedel[*]
olafl@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Marina Papatriantafilou
ptrianta@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

Pattern matching is an important building block for many security applications, including Network Intrusion Detection Systems (NIDS). As NIDS grow in functionality and complexity, the time overhead and energy consumption of pattern matching become a significant consideration that limits the deployability of such systems, especially on resource-constrained devices. On the other hand, the emergence of new computing platforms, such as embedded devices with integrated, general-purpose Graphics Processing Units (GPUs), brings new, interesting challenges and opportunities for algorithm design in this setting: how to make use of new architectural features and how to evaluate their effect on algorithm performance. Up to now, work that focuses on pattern matching for such platforms has been limited to specific algorithms in isolation.

In this work, we present a systematic and comprehensive benchmark that allows us to co-evaluate both existing and new pattern matching algorithms on heterogeneous devices equipped with embedded GPUs, suitable for medium- to high-level IoT deployments. We evaluate the algorithms on such a heterogeneous device, in close connection with the architectural features of the platform and provide insights on how these features affect the algorithms' behavior. We find that, in our target embedded platform, GPU-based pattern matching algorithms have competitive performance compared to the CPU and consume half as much energy as the CPU-based variants. Based on these insights, we also propose *HYBRID*, a new pattern matching approach that efficiently combines techniques from existing approaches and outperforms them by 1.4x, across a range of realistic and synthetic data sets. Our benchmark details the effect of various optimizations, thus providing a path forward to make existing security mechanisms such as NIDS deployable on IoT devices.

CCS CONCEPTS

• **Security and privacy** → **Network security**.

KEYWORDS

NIDS, GPU computing, embedded devices, pattern matching

ACM Reference Format:
Charalampos Stylianopoulos, Simon Kindström, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. 2019. Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3359789.3359811

1 INTRODUCTION

With the widespread adoption of Internet of Things (IoT) technologies, an increasing number of devices are equipped with the ability to communicate and connect to the Internet. While promising increased efficiency and flexibility, connected devices are vulnerable, as shown by recent attacks that specifically targeted IoT devices such as connected cameras and thermostats [31, 42]. Protecting such devices with well-established security mechanisms such as network intrusion detection systems (NIDS) is necessary. Yet, such mechanisms are hard to deploy on these devices, because their core function depends on pattern matching, a bottleneck that needs significant resources (more than 70% of the running time of the system may be spent on pattern matching [2]).

In the context of NIDS, pattern matching algorithms scan the packet payload (Deep Packet Inspection) and detect any occurrence of malicious string signatures (known in advance) in the stream of packets. Pattern matching algorithms are often studied in close relation with the hardware platforms, because the characteristics of the target platform play an important role on performance. This is evident by the number of algorithms in the literature that target specific platforms or build on optimizations that utilize specific characteristics, e.g., CPU caches [10], vector instructions [37], FPGAs [36] or hardware accelerators [21].

[*] Also with Kiel University, Germany.

ACSAC, December 9–13, 2019, Puerto Rico, USA

ACM ISBN 978-1-4503-7628-0/19/12...$15.00
https://doi.org/10.1145/3359789.3359811

In IoT deployments, one can find significant hardware diversity from the edge to the core of the network. For example, the introduction of new computing approaches with new hardware diversity in the fog increases the design space for pattern matching algorithms and offers new capabilities for improvements of performance. This is leveraged in recent research [32] with algorithms tailored to medium-ranged embedded platforms, such as Raspberry Pi or Odroid [6]. The latter platform offers an interesting combination of an IoT board equipped with an embedded, programmable Graphics Processor Unit (GPU). Such medium/high range IoT devices can take the role of NIDS boxes, protecting a network of resource-constrained devices closer to the edge of that network.

However, challenges remain and the feasibility of such platforms for NIDS is not established yet, especially with respect to performance, since their hardware characteristics are different from the well-studied high-end platforms. First, significant effort is required to take an algorithm for a particular hardware and change it to run well on another type of hardware. Given the development effort, it would be favorable to better understand what algorithms to port and how. Secondly, it is not clear how different optimizations translate across hardware. For example, the design of many pattern matching algorithms (e.g. [10, 37, 41]) is driven by specific features of the target architecture, such as cache sizes and vector execution units, so it is unclear how they perform on a different system. Thirdly, most work is documented in isolation, with specific data, and within a specific framework.

As such, a common methodology for benchmarking algorithms tailored for fog-layer devices is needed, to understand possibilities and limitations of the hardware itself, as well as the effects of algorithm engineering and, most importantly, the interplay of the algorithm and the hardware features.

Contributions: In this work we present a co-evaluation of various pattern matching algorithms on a heterogeneous computing platform. We target a medium range embedded device (an Odroid XU3) that is equipped with a programmable GPU, i.e., a GPU that can support general-purpose computing.

- We design a benchmark that facilitates a systematic comparison between different pattern matching algorithms, using realistic data sets that capture real NIDS workloads.
- We co-evaluate well-known CPU and GPU based algorithms, including our own GPU adaptation of a state of the art CPU based algorithm (DFC).
- We evaluate the effect of different platform-specific optimizations and parameters in the performance of the algorithms, both in terms of execution time as well as energy consumption, to guide the community in future research.
- Based on our methodology, we were also able to create a new algorithm, *HYBRID*, that effectively combines the benefits of existing approaches and achieves up to 1.4x speedup in pattern matching compared to the best GPU baseline.

The remainder of the paper is organized as follows: in Section 2 we discuss the general aim and design considerations of the benchmark. Section 3 summarizes the algorithms included in our benchmark, both existing as well as new ones. Section 4 provides details on the target hardware and discusses relevant optimizations. In Section 5 we present and discuss the benchmark results. We present related work in Section 6 and conclude in Section 7.

2 BENCHMARKING AIM AND CONSIDERATIONS

This section discusses the high-level design considerations for our benchmark and serves as a guideline for the general methodology followed in our work. We motivate the aim of the benchmark, the choice of algorithms and the steps we consider towards a fair and useful comparison between them.

Utilization of the target platform. The aim of this benchmark is to analyze the performance of pattern matching algorithms on a specific set of newly introduced hardware platforms: embedded devices with integrated GPUs that support General Purpose GPU computing (GPGPU). Originally designed for processing graphics, in the last decade, GPUs have been proven particularly successful in accelerating general-purpose workloads as well, mostly due to their highly parallel architecture. Their popularity increased further with the introduction of libraries that simplify the writing of GPU programs, namely CUDA [28] and OpenCL [16]. We focus on algorithms that are written or can be ported to OpenCL, since it is the library that the hardware supports (c.f. Section 4).

Choice of algorithms. Each pattern matching algorithm follows a different approach, but most of them fall into two main categories: *state machine* based approaches and *filtering* based ones. The first category involves variants of the Aho-Corasick algorithm (c.f. next section), where a state machine is created out of the patterns and traversed based on the input. On the contrary, filtering based approaches try to quickly isolate parts of the input that do not contain any matches and spend more resources on the parts that might potentially match with one of the patterns.

In the benefit of covering a wide spectrum of approaches and gaining insights from how different algorithms perform on the device we target in this work, we pick representative algorithms from both families. The description of the chosen algorithms follows in the next section.

General and platform-specific parameters and characteristics. The design and performance of the algorithms mentioned above are greatly affected by the system parameters and the characteristics of the target architecture. Each family of algorithms is affected by those characteristics to a different extent. Thus, it is important to examine the effect of a variety of parameters on algorithms from different families, especially when targeting architectures with unique characteristics such as the one we use in our work. In Section 4 we discuss the architecture characteristics and the parameters that are relevant to examine.

Use of realistic data sets and patterns. The performance of pattern matching algorithms for NIDS is often highly data-dependent and fluctuates based on i) the number and size of patterns used, and ii) the type of traffic that is monitored, e.g., randomly generated data versus captured traffic from actual deployments. Knowing the type of traffic and how the performance of each algorithm will be affected beforehand is hard. However, it is important to experiment with sets of traffic and patterns that are as close to the ones found in real life as possible. We use publicly available data sets that simulate traffic from real deployments, as well as patterns taken directly from the official pattern distributors for Snort [35], the de-facto NIDS. We refer to Section 5 for more information on the choice of data sets and patterns.

Identical functionality. Putting different algorithms together under the same fair and meaningful framework is not trivial. Often, algorithms are designed with different requirements in mind, e.g., reporting all the matches and their positions in the input, versus just reporting how many matching patterns the input contains. It is hard to judge which functionality to implement, since different applications of those pattern matching algorithms might have different requirements. However, it is important to ensure that we keep the same functionality for the algorithms we compare in this benchmark. For this reason, we have manually inspected the compared versions and rewritten parts of them to ensure that the different implementations have identical functionality.

In later sections we will return to the considerations mentioned above and discuss some of their implementation details.

3 CONSIDERED ALGORITHMS & NOVEL DESIGNS

Since this benchmark focuses on NIDS applications of pattern matching, we consider algorithms that have been designed or used in NIDS workloads. As previously mentioned, we cover algorithms from the two main pattern matching algorithm families: *state machine* based and *filter* based.

The algorithms chosen from the literature are representative of methods that can benefit from different architectural features of the computing platform, such as the high parallelism/latency hiding of the GPU as well as cache memories. Based on insights from the existing approaches and their properties in heterogeneous computing platforms, we also introduce a new method, which we call *HYBRID*, that aims to combine the benefits of the two. This algorithm will also be part of our benchmark. The algorithms are summarized in Table 1, where we also specify the code repository (if applicable) as well as the effort to adapt the code to the evaluation.

3.1 State Machine Based Algorithms: Aho-Corasick and PFAC

One of the most well-known algorithms for pattern matching is Aho-Corasick (AC) [1]. Aho-Corasick has a preprocessing stage where it builds a *Finite State Automaton (FSA)*, in other words a state machine, with all patterns. However the FSA differs from a normal FSA as *failure transitions* are also added. These failure transitions occur when there is a mismatch between the input and the state machine, and point to the state sharing the longest common prefix with the current state. During processing, one character at a time is read from the input and used to determine the next state, using a state transition table that represents the state machine. An example of Aho-Corasick's state machine is shown in Figure 1. The arrows between each branch indicate failure transitions.

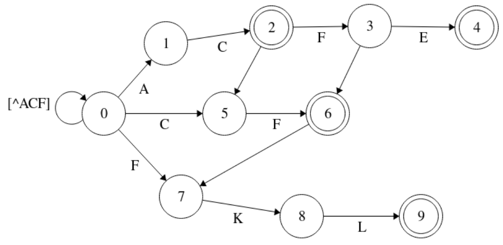


Figure 1: Aho-Corasick state machine for the patterns AC, ACFE, CF and FKL.

Algorithm 1: High level pseudo-code of PFAC.

```
for each character C in input stream do
    find the first state based on character C
    while no patterns found AND state not reset to 0 do
        read next character
        traverse the state machine
    end
end
```

Aho-Corasick is a simple way of performing multiple pattern matching that requires a small number of operations per byte. However, storing all the states and their transitions requires significant memory [26, 27]. Because of the large memory requirements, many cache misses occur during state transitions [26]. Nonetheless, AC is often used in practice and a variant of AC is used in NIDS such as Snort [35]. We include Aho-Corasick as a CPU-based baseline in our benchmark, due to its widespread use.

Parallel Failureless-AC (PFAC) is a parallel implementation of Aho-Corasick with a focus on GPUs [21]. In PFAC, every GPU thread starts from a single character and follows the state transitions until a match is detected or the state machine has returned to the original state, indicating there was not a match starting from that character. The state machine in PFAC is simplified compared to that of Aho-Corasick in that the failure transitions are removed and each pattern is an individual branch in the state machine. PFAC spawns many threads (up to one thread per input character) but most of them will quickly detect no matches and exit. Algorithm 1 shows a highly simplified pseudo-code version of PFAC.

One reason that makes PFAC an interesting algorithm to include in this benchmark is the fact that it has been evaluated on resource-constrained embedded GPUs, similar to the ones we target in this paper. Aragon et al. [4] implement PFAC in OpenCL and evaluate it on an ARM Mali GPU, considering various optimizations. In this work, we build upon this evaluation and extend it with more algorithms and insights.

3.2 Filter Based Algorithms: DFC and V-Patch

The motivation behind filter based algorithms is to create small filters that can quickly determine and discard parts of the input that do not contain any matches, in a quick and cache-efficient way, contrary to the cache-inefficient data structures of state machine based algorithms.

Acronym	Algorithm	Family	Code	Effort	Comment
AC (CPU)	Aho-Corasick [1]	state machine	Snort repo [34]	low	CPU baseline, used in Snort
DFC (CPU)	Direct Filter Classification [10]	filter	[10, 37]	low	CPU baseline (filter-based)
PFAC (GPU)	Parallel Failureless AC [21]	state machine	[4]	medium	code required some work to adapt to benchmark
DFC (GPU)	Direct Filter Classification [37]	filter	[37]	high	our own implementation (the first implementation of DFC for the GPU) based on [37]
HYBRID (GPU)	Mix of DFC and PFAC	mixed	this paper	high	our own design: a hybrid combining DFC and PFAC, implemented for the GPU

Table 1: A summary of the evaluated algorithms and their acronyms.

Direct Filter Classification (DFC) is a state of the art, memory and cache efficient pattern matching algorithm implemented on CPUs, using Direct Filters (DFs) [10]. A Direct Filter is a bitmap that summarizes some consecutive bytes from the pattern, and is small enough to be cache resident. A 2 byte DF will use 2 consecutive bytes from a pattern, e.g., the first or last two bytes, to index a bit in the bitmap.

DFC performs matching in multiple phases: *filtering* and *verification*. In the filtering phase, a window of two bytes is slid over the input, summarized and matched to the filter. If the window matches, additional DFs requiring more bytes for indexing may be used to check that it really is a match. The verification phase performs exact matching using a compact hash table with efficient indexing. Unlike other filter based algorithms (e.g. FFBF [26]), DFC works with patterns of any length and avoids expensive hash computations for indexing the filters. Algorithm 2 shows a highly simplified pseudo-code version of DFC.

Algorithm 2: High level pseudo-code of DFC.

```
for each character C in input stream do
    feed C (and neighbouring character) through a series of filters
    if there is a hit in the filters then
        do verification
    end
end
```

Since its inception, DFC has been the basis for further improvements on its filter design and its ability to utilize features of the architecture, such as vectorization. Stylianopoulos et al. [37] redesign the filter architecture of DFC (S-Patch) to better fit realistic traffic scenarios and allow for a vectorizable design (V-Patch), which leads to an increased throughput of up to 3.6x compared to the original DFC algorithm.

In this work, we implemented our own GPU version of DFC. We based our filter design on the CPU filtering design of S-Patch [37]. Such a GPU implementation is not straight-forward and involves re-factoring the CPU version and orchestrating the communication between the CPU and the GPU. We briefly mention here that the data structures that hold the series of hash tables used in the verification version of DFC need to be serialized so that they can be transferred to the GPU. Moreover, the results of the pattern matching kernel on the GPU are stored in a buffer that indicates whether or not there was a match, for each character in the input. Those results are later transferred to the CPU and processed there.

We include in our experiments both the CPU version of DFC, as a baseline, as well as our own GPU algorithm implementation. Upon these implementations, we evaluate and discuss the effect of different optimizations and algorithm engineering methods.

3.3 A Hybrid Approach

In addition to algorithms from the two different families described above, in this work we also introduce a new approach, *HYBRID*, that borrows the benefits of both families.

The new, albeit simple idea behind this approach is to filter the input using one filter, similar to the ones used in DFC and then perform PFAC-based pattern matching only on the parts of the input that cause a hit in the filters. The motivation behind this scheme is that it combines: (i) the good cache locality of filter based approaches, allowing us to quickly filter out most of the input (c.f. Section 5.5) and (ii) the ability to avoid the costly verification part of DFC by falling back to PFAC which uses only a minimum number of operations (jumps from one state to the other). The fact that, in PFAC, we can traverse the automaton starting from any individual character in the input (contrary to, e.g., Aho-Corasick), makes it possible to pipeline the execution of PFAC with that of DFC, by starting an automaton traversal only where there is a hit in the filter. Algorithm 3 shows a highly simplified pseudo-code version of *HYBRID*.

Algorithm 3: High level pseudo-code of *HYBRID*.

```
for each character C in input stream do
    feed C (and neighbouring character) through a series of filters
    if there is a hit in the filters then
        find the first state based on character C
        while no patterns found AND state not reset to 0 do
            read next character
            traverse the state machine
        end
    end
end
```

The *HYBRID* algorithm is actually a result of the insights gained from the benchmark described in this paper and came as a later addition to the paper. However, to ease the presentation, we evaluate the *HYBRID* algorithm together with the other algorithms included in this benchmark in Section 5.

4 HARDWARE-ORIENTED ALGORITHM OPTIMIZATIONS

In this section, we provide details on the architecture of the target platform and discuss relevant algorithm engineering methods that make use of the architectural features.

4.1 Overview of the Target Platform

We use the ODROID-XU3 [6] to execute all tests. The XU3 uses the Exynos 5 Octa (5422) chip that has a quad-core ARM Cortex-A15 and quad-core ARM Cortex-A7, along with 2GB of RAM. The Exynos 5422 is used in one variant of the Samsung Galaxy S5, showing that the XU3 is a reasonable choice for a more powerful embedded device today. It also suggests that hardware and the pattern matching algorithms considered in this paper could also be used, e.g., in scanning malicious mobile application. The most important reason as to why we use the XU3 is because it possesses a GPU that allows General-Purpose computing on Graphics Processing Units (GPGPU). Further reasons are that it has a high-speed Ethernet port, allowing for high-speed network sniffing. We chose the Odroid XU3 over the newer XU4 model (that has the same CPU and GPU) because the former has on-board energy sensors that allow us to easily measure the energy consumed.

The XU3's variant of the GPU is a Mali-T628 MP6 [5] that has 6 cores (much less than the typical high-end discrete GPUs). These shader cores may be programmed using OpenCL. The GPU does not have a separate device memory but share the same physical memory as the CPU [7]. Moreover, any shared or local memory on the GPU is actually mapped in the global memory instead. Each core has L1 and L2 memory caches to remedy the cost of always accessing the global memory. These caches have a 64-byte cache line. There are two 16KB L1 caches for each core, one used for generic memory accesses and one for texture memory. Another unique feature of the Mali GPU is that there is Single Instruction Multiple Data (SIMD) parallelism supported within each GPU thread. Finally, each GPU thread has a separate instruction counter, meaning that divergent execution is not a problem on the Mali-GPU [7]. Architectures with similar features (e.g. shared memory between the CPU and the GPU) are also available from NVIDIA [29] and we expect similar performance trends.

Based on the above the description, the Odroid XU3 platform is a relatively powerful single board computer that belongs to the medium/high range of IoT devices. Although the architecture and the capabilities of the device are far different from those of the typical end-point, resource-constrained sensors (that are usually powered by ARM Cortex-M or similar processors), medium/high range devices have a central role in the IoT context, as they can play the role of cluster-heads or gateways to a network of less powerful end-nodes. Also, platforms like the Odroid XU3 fit in the fog computing [11] context, as an intermediate layer between the cloud and the edge network.

4.2 Relevant Algorithm Optimizations

In this section, we discuss optimizations that are relevant to explore and relate to (i) features of the architecture and (ii) possibilities for the evaluated algorithms to make good use of those features. A similar discussion on some of these parameters can also be found in the work by Aragon et al. [4]. We extend their discussion with more parameters to also consider the use of texture memory for storing the relevant data structures, as well as the use of vectorization, as it has shown promise for CPU-based algorithms [37]. In Section 5.2 we evaluate the effect of each of the optimizations described in detail in the following sections. They are also summarized in Table 2, with a comment on how they were realized and the corresponding effort.

Reducing memory transfers (MAP): Memory transfers between an OpenCL host (CPU) and OpenCL device (GPU) is often the bottleneck in GPGPU applications [13]. In most cases, memory transfers include a copy of data from the physical memory of the host to that of the GPU. The platform we target here offers an interesting way to alleviate this problem. The GPU shares the same physical address spaces as the CPU, making memory copies redundant. A naive use of the OpenCL API would still force the driver to perform memory copies of the data to be transferred. Instead, it is possible to map a memory region (through the OpenCL API) to make it accessible from the GPU and then map it back to the CPU to read the results.

Increasing work per thread (THR): An issue with having each thread handle a single character, is that many threads are spawned. This is not a free operation, costing time and energy. By having each thread process multiple characters, fewer threads are needed.

Utilizing local memory (MEM LOCAL/GLOBAL): The GPU local memory is shared between each work-item in a workgroup and is often faster to access than the global memory. As described earlier, the target platform does not have dedicated local memory and references to local memory are served by the global memory instead. Nonetheless, we experiment with local memory to show how optimizations that would be beneficial on high-end GPUs can actually have a negative impact on embedded platforms like ours.

Utilizing texture memory (MEM TEXTURE): An additional optimization strategy used in this work is to utilize the texture memory, an on-chip memory designed to quickly serve addresses that have spatial locality. As there is a separate L1 cache for textures (16KB) in the XU3, one may increase the cache hits further by storing one or more DF as a texture. It is worth noting that storing the filters in texture memory results in the need for some additional registers and computations to retrieve the one bit of interest, potentially reducing the gain from better cache locality.

Vectorized execution (VEC): As mentioned earlier, each GPU thread can operate on multiple elements at a time in a SIMD fashion. In this work we implement and evaluate a vectorizable version of DFC, based on V-Patch [37], and *HYBRID*. In this version, the filtering is done on multiple (in this case eight) elements at the same time. Notice however that some parts of the filtering are still done in scalar code, due to the lack of special vector instructions such as the gather instruction [25], which is important for the implementation of V-Patch.

Altering OpenCL workgroup size (WG): Another variable is the size of the OpenCL workgroup [7]. Fewer workgroups should result in a lower overhead for maintaining them, but more results in better latency hiding. The size of a workgroup is usually a maximum of 256, as is the case on the ODROID-XU3.

Configurations			
Acronym	Description	Effort	Comment
MAP	The use (or not) of memory mapping to avoid data transfers.	low	Done through the OpenCL API
THR	Thread granularity.	low	Done through the OpenCL API
MEM	Type of memory used to store the filters of DFC (GLOBAL/LOCAL/TEXTURE).	high	Using texture memory requires data structure re-arrangement
VEC	The use (or not) of vectorization in the GPU kernel.	high	Vectorization requires code refactoring
WG	Work group size.	low	Done through the OpenCL API

Table 2: A list of optimizations and their acronyms used throughout the evaluation.

5 EVALUATION

In this section we co-evaluate the algorithms presented in Section 3 under the same benchmark. We start by describing the methodology followed throughout the experiments. Then we focus on DFC (which has not been previously studied in a GPU context) and study the effects of different optimizations. We then report and discuss the overall performance comparison between the different versions, as well as how the performance changes across different data sets and number of patterns. Finally, we discuss some insights from the execution of *HYBRID*.

5.1 Evaluation Methodology

In our study we use the Odroid XU3, presented in Section 4, to run all experiments. We test the algorithms using the three following publicly available data sets that represent realistic traffic traces:[1]

- 162MB of HTTP traffic from the DARPA 2000 data set [20] (unless otherwise stated, this data set is used)
- 100MB of traffic from the ISCX 2012 data set [12, 33], and
- 100MB of traffic from the BigFlows data set [3].

We also test against 100MB of randomly generated data. In all cases the algorithms read the data from a file in chunks of 25MB each and the cost of reading the data is included in the measurements.

The set of malicious patterns to be matched are taken directly from Snort v2.9: we use a set of 2,183 HTTP-related patterns from the default Snort distribution, as well as 5,000 randomly chosen patterns from *emergingthreats.net*. Unless otherwise stated, the set of 2,183 patterns is used for experiments.

Due to restrictions in OpenCL, statically allocating enough space to fit the longest possible pattern would be wasting memory resources. For this reason, we use no pattern longer than sixty-four (64) characters and remove any instance longer than this threshold. This choice is motivated by the distribution of pattern lengths in the first and second pattern data set, which is shown in Figures 2 and 3 respectively. In the case of the patterns from emergingthreats.net, the longest pattern is 513 characters, while most are much much shorter than that. Removing any pattern longer than 64 characters removes approximately 80 and 500 from the first and second pattern data set respectively. This decision to remove excessively long patterns is similar to what happens in systems such as Snort, where long patterns are truncated and only shorter versions of them are used in the pattern matching engine.

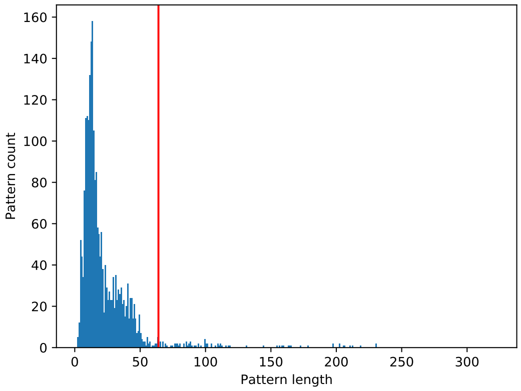


Figure 2: Snort HTTP patterns

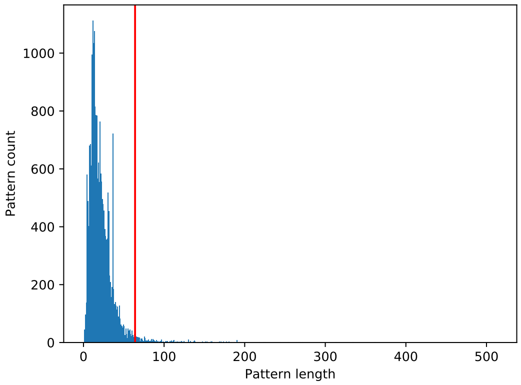


Figure 3: All patterns from emergingthreats.net

Figure 4: Distribution of pattern lengths. Red line signifies 64 characters

In order to ensure a fair comparison between the algorithms based on the considerations mentioned in Section 2, we have ensured that all versions have identical functionality, i.e., they all count the number of patterns that are matched in the input.

Finally, our main *performance criterion* used to compare the different versions is the *execution time* of the algorithm as a whole, which includes: reading data from a file, performing pattern matching, counting the number of matches and, for the GPU versions,

[1] We are aware of the artifacts in the DARPA 2000 set, and the discussions in the community about its suitability for measuring the *detection capability* of intrusion detection systems [23, 24]. In our experiments, we use it only for the purpose of comparing execution time and energy usage between algorithms, allowing for future comparisons on a known and easily-available dataset.

mapping memory between the CPU and the GPU. We do not include the cost of pre-processing, e.g., building the state machines of PFAC or the filters of DFC, since this happens offline before deployment. When measuring energy consumption, we gather measurements, using the on-board energy sensors of the device, at a rate of 100Hz.

5.2 Deciding Parameters for DFC

It is important to understand how the characteristics of the target platform affect the performance of the algorithms and allow for different optimizations. This analysis has been done already for PFAC by Aragon et al. [4], but not for DFC since we are the first to implement a GPU version of it. Therefore, we follow the approach of Aragon et al. and we present, in Table 3, the effects of the optimizations discussed in Section 4 on DFC's performance.

In order to limit the number of different configurations that need to be examined, we follow a greedy approach: we change one parameter until we find the value for which we get the best performance, i.e., the smallest TOTAL execution time, then keep that value and move on with the next parameter. The configurations and their effect in Table 3 are described below. In the results section of the table we report the time it takes to write/read data to/from the device (WRITE/READ), the execution time of the kernel (KERNEL) that implements pattern matching on the GPU, and the total measured time (TOTAL), as well as the energy consumed (ENERGY). Yellow boxes indicate configurations that improve the overall performance (TOTAL time) compared to the previous configuration, while red boxes indicate changes in configuration that do not have a positive effect on performance.

Reducing Memory transfers (MAP): The MAP configuration option is binary (yes/no) reflecting whether we use *map* to reduce memory copies. Overall, mapping memory instead of copying it has the most significant effect, seen from the reduction in the time it takes to read/write data to/from the device. This is expected since, as previously mentioned, the CPU and the GPU share the same physical memory which makes memory copies redundant.

Increasing work per thread (THR): The THR configuration option controls the thread-granularity, i.e., how many characters each GPU thread processes. It is varied from 1 character per thread to 48. Increasing the thread granularity to more than one character per thread resulted in a great decrease in the kernel execution time, since there is now more work for each thread to do, instead of exiting early. However, at a certain point there are too few threads to keep the hardware pipeline busy and performance decreases.

Utilizing global, local or texture memory (MEM): The MEM configuration option has three cases, reflecting the type of memory where the filters are stored, i.e. global, local, or texture memory. As expected, using local memory to store the filters had a negative effect on performance, because the local memory in the Mali GPU is simulated by global memory. Moreover, this simulation seems to come at a cost in that the local memory is more expensive than the global memory with the same overall settings (which was also observed by Aragon et al. [4]). Surprisingly, using texture memory did not have a beneficial effect on performance either. This is likely due to the extra instructions needed to access and isolate the relevant bits from the filter when it is stored as a texture.

Vectorized execution (VEC): The VEC configuration option controls whether the kernel is vectorized. In related work [37] for the CPU, vectorization played a large role in improving the performance. However, vectorizing the kernel in our setting did not have a beneficial effect, likely due to the lack of proper *gather* operations, which means that we incur a penalty when switching between vectorized and scalar code. However, future architectural support for vectorized operations would likely bring improvements similar to what is seen in [37] for the CPU.

Altering OpenCL workgroup size (WG): Finally, the WG configuration option varies the workgroup size. The best configuration for the work-group size is 128 work-items per work-group.

5.3 Overall Comparison

After establishing a set of parameters that works best for DFC (the best configurations from Section 5.2), in addition to the parameters that work best for PFAC from Aragon et al. [4], we put all algorithms to the test in this section. Figure 5 shows the execution time of all algorithms when processing the DARPA data set with the default set of 2,183 patterns. We run each experiment five times and report the average execution time. In Figure 5 we have broken down the cost to its individual components. Post-processing refers to counting the number of matches, based on the results read from the GPU. Even though the main focus is the execution time of pattern matching itself (light blue bars), we still present the additional costs for completeness.

When comparing results between the performance of the CPU baseline and the GPU, keep in mind that a direct comparison is not always straightforward. Here, we compare against the performance of parallel algorithms that have been transformed to operate on the GPU, against single-threaded algorithms that operate on the CPU. Using all the available threads in the CPU is likely to improve CPU performance, assuming there is an efficient way to parallelize the algorithms on the CPU and taking into account bottlenecks involved in CPU parallelization. Still, our comparison allows us to express the performance of the GPU in terms of something easily understood: the performance of a single CPU thread. Moreover, note that the GPU and CPU can complement each other: it is possible to have both the GPU and the CPU threads working simultaneously, either on disjoint parts of the input data, or on different tasks (e.g. the post-processing can be done in parallel by the CPU).

CPU vs GPU: First, comparing the CPU (the first two bars in Figure 5) and the GPU versions (the rest) shows that all GPU versions perform significantly better than the CPU versions, up to 2X less total execution time when comparing Aho-Corasick with *HYBRID*. That result supports the claim that embedded accelerators such as the Mali GPU can effectively offload pattern matching for Network Intrusion Detection applications, regardless of the family of algorithms used. Particularly, we notice that the execution cost of the pattern matching part alone (light blue bars) is greatly reduced on the GPU by up to 7X, suggesting that pattern matching kernels make great use of the high degree of parallelism offered by the GPU. However, notice that the extra costs of data transfers and post-processing of the results are significant and offset, to some extent, the total benefit of offloading pattern matching on the GPU.

Configurations					Results					Improvement	
MAP	THR	MEM	VEC	WG	WRITE (ms)	READ (ms)	KERNEL (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	1	GLOB	NO	128	431	4870	2596	11170	2867	1	1
YES	1	GLOB	NO	128	198	945	2661	6787	1806	1.65	1.59
YES	8	GLOB	NO	128	195	678	1829	4118	991	2.71	2.89
YES	16	GLOB	NO	128	195	654	1599	3790	923	2.95	3.11
YES	24	GLOB	NO	128	196	653	1488	3670	875	3.04	3.28
YES	32	GLOB	NO	128	194	644	1427	3451	854	3.24	3.36
YES	40	GLOB	NO	128	196	648	1406	3440	845	3.25	3.39
YES	48	GLOB	NO	128	196	647	1412	3464	845	3.22	3.39
YES	40	LOC	NO	128	194	643	6267	8411	2238	1.33	1.28
YES	40	TEX	NO	128	197	646	1777	3897	996	2.87	2.88
YES	40	GLOB	YES	128	191	637	2065	4172	1095	2.68	2.62
YES	40	GLOB	NO	64	196	645	1674	3784	905	2.95	3.17
YES	40	GLOB	NO	256	196	645	1435	3501	870	3.19	3.29

Table 3: Summarized configuration impact for GPU version of DFC, similar to the evaluation methodology followed in [4].

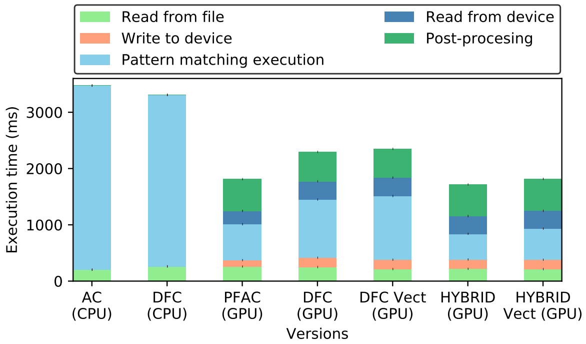


Figure 5: Execution time (broken down to its different components) of the different versions.

Even though data copies are avoided, the overhead of mapping/unmapping the memory region every time the buffer is full is still significant in this application. As such, a new design that would minimize this effect further, e.g., by overlapping the CPU and the GPU execution, would be interesting to explore.

We now focus on the GPU versions and compare them with each other.

PFAC vs DFC (GPU): Comparing PFAC with DFC (GPU), we find the pattern matching part of PFAC is 1.62x faster than DFC (Figure 5). This is likely due to two reasons. First, the cost of verification in DFC is high, because it includes accesses to hash tables containing the patterns as well as exact matching. Second, DFC was originally designed to have an increased instruction count compared to Aho-Corasick (more instructions needed to access filters and hash tables) but a better cache utilization. In GPUs, the benefit from high cache utilization is not as important as in CPUs, due to the large number of threads spawned and the device's ability to switch between threads quickly and hide the high memory latency incurred when there is a cache-miss. As a result, the benefits of cache utilization are offset from the cost of the extra instructions and the costly verification phase.

Interestingly, the vectorized version of DFC (DFC Vect (GPU)) fails to bring any speedup in the GPU. As already mentioned in Section 5.2, vectorizing the kernel for each GPU thread is not efficient for this workload, due to the lack of *gather* operations.

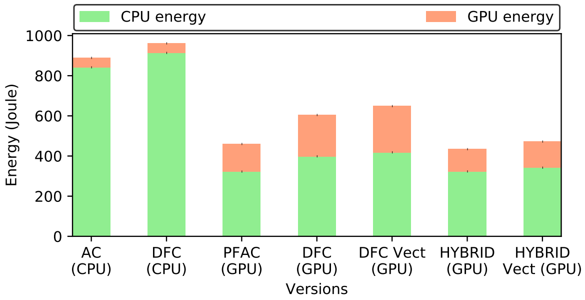


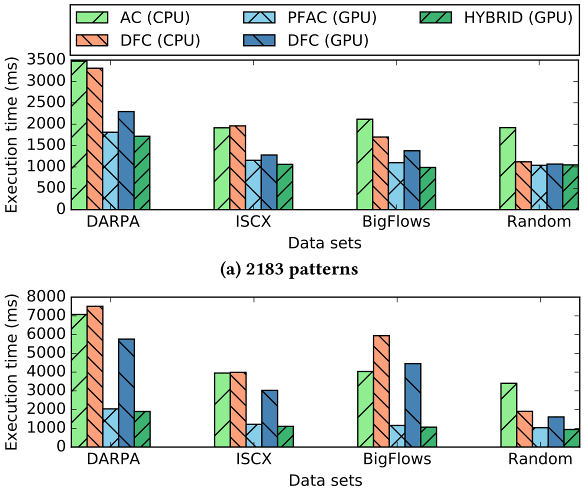
Figure 6: The CPU and GPU energy consumed by the different versions.

The *HYBRID* approach: Among the GPU versions, *HYBRID* manages to execute the pattern matching part faster than both PFAC and DFC (1.3X faster than PFAC and 2.2X faster than DFC). Contrary to DFC, *HYBRID* avoids the costly verification by switching to PFAC when there is a hit in the filters, while still keeping the benefits of using filters to quickly filter out parts of the input that cannot contain a match. As in DFC though, the vectorized version (*HYBRID* Vect (GPU)) does not bring a speedup.

Energy Comparison: Finally, in Figure 6 we report the total consumed energy for the same experiment, across the different versions. Overall, we see that the reduction in execution time from the previous figure translates, in almost all cases, directly to reduction in consumed energy, with the best GPU version (*HYBRID*) consuming 2X less energy than Aho-Corasick. In this figure we also show separately the energy consumed by the CPU (A15) and the GPU. Notice that the GPU versions still consume significant CPU energy, mostly due to post-processing and idly waiting for the GPU execution to finish.

5.4 Varying the Data Sets and the Number of Patterns

In this section, we evaluate the behavior of the different algorithms across different data sets and number of patterns. Insights about this behavior are important when considering real packet processing deployments where the characteristics of the incoming traffic might change or new malicious patterns might be added in the database.



(a) 2183 patterns

(b) 5000 patterns

Figure 7: Execution time across different data sets when using a) the default set of 2183 patterns and b) 5000 randomly chosen patterns.

Varying the data sets: Figure 7a shows the overall cost (including reading from file, data transfers etc.) for different real and synthetic data sets, when using the default set of patterns. Here, we have omitted the vectorized versions since they do not bring a significant change in performance. In all cases, the GPU versions outperform the CPU, with *HYBRID* having the smallest total execution time in almost all cases.

An interesting observation is that, when using randomly generated data, the CPU version of DFC performs significantly better than with real traffic. This is because randomly generated characters are very unlikely to cause hits in the filters, reducing the need for verification and most of the memory accesses are now served by the CPU cache [10]. This stresses the need to use realistic data sets when comparing algorithms in a benchmark.

Increasing the number of patterns: In Figure 7b, we increase the number of malicious patterns to 5,000. Overall, we see that the execution time of all versions increases (notice the different range on the y-axis between Figures 7a and 7b). Aho-Corasick nearly doubles in execution time, mostly due to the fact that the state machine grows and does not fit the CPU cache.

Interestingly, GPU versions that also use a state machine, i.e., PFAC and *HYBRID*, do not get affected to such a great extent. This is, again, due to the fact that GPU cache misses are not detrimental if the GPU is able to hide the latency of accessing the main memory, by switching between many active threads. We also notice that both DFC versions increase significantly in execution time, likely due to the exact matching part of the verification step of DFC, which is costly and happens more often when the number of patterns increases.

5.5 Deciding a Filter Size for *HYBRID*

In this section, we take a closer look at the *HYBRID* approach and specifically at the effect of the filter size on the performance of the algorithm. Intuitively, a larger filter will be more sparsely populated, meaning that we expect fewer hits (therefore fewer times that we need to resort to the state machine of PFAC) when filtering the input. On the other hand, a small filter can fit in the GPU cache (16KB per shader core), making it easier to access.

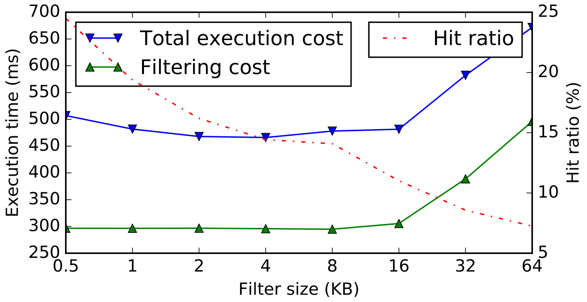


Figure 8: Cost of filtering and total GPU execution time of the *HYBRID* approach (left y-axis), as well as the effectiveness of the filtering (hit ratio, right y-axis), as we increase the filter size.

In the following experiment, we use the first three characters from each pattern to populate the filter, after hashing them with a simple multiplicative hash function. We vary the effective size of the filter by masking the hash value appropriately, effectively bounding the size of the filter. In Figure 8 we report: (i) the cost of only accessing the filter (green line, left y-axis) (ii) the hit ratio of the filter, i.e., the number of hits in the filter compared to the total number of input characters (red line, right y-axis) and (iii) the total cost of the *HYBRID* execution on the GPU, i.e., the cost of both filtering and traversing the state machine of PFAC when there is a hit in the filter (blue line, left y-axis).

As expected, the hit ratio decreases as the filter becomes larger, ranging from 25% hit rate for a half-KB filter to 5% hit rate for a 64KB filter. The cost of accessing the filter remains small while the filter is smaller than the size of the cache (16KB) and increases rapidly afterwards. The best performing configuration is when the size is much less than the size of the cache, because we still need to save space for the state machine for when there is a match.

5.6 Summary of the Results

In this section, we presented the results of our benchmark where multiple pattern matching algorithms are brought to the test on a medium/high range IoT device with an embedded GPU. Experiments using real data sets and patterns showed that: (i) the GPU is a viable alternative for pattern matching on these devices, both in terms of execution time and energy consumption and (ii) there were significant differences in performance between the GPU based algorithms, uncovering the strength and weaknesses of each approach. Stemming from this analysis, it was possible to identify new meaningful combinations (*HYBRID*) that combine techniques from existing work and outperform them. The co-evaluation of CPU and GPU algorithms also uncovered how different algorithms utilize the hardware's resources differently: in the CPU, having good cache locality proved important, whereas it mattered less in the GPU. Finally, using both synthetic and real data sets showed that most pattern matching algorithms are highly data-dependent, raising interesting future directions about the feasibility to adapt to the distribution of the data.

6 RELATED WORK

Pattern matching has been an active field of research for decades and the literature offers numerous algorithms for a variety of different settings. On single string pattern matching, Boyer-Moore [9] and Knuth-Morris-Pratt [18] are two well-known algorithms that skip over parts of the input and perform pattern matching in sublinear time. However, such algorithms do not work well in the context of pattern matching for intrusion detection where there are many patterns to search for simultaneously. An important multiple string matching algorithm, other than the Aho-Corasick [1] we already summarised in Section 3, is the Wu-Manber [43] algorithm that keeps a table to store information on how many bytes we can skip from the input. Hyperscan [41] is an open source regular-expression library that also includes many optimizations for fixed string pattern matching. However, these optimizations are built around Intel's high-end vector instruction set extensions and are not available in the ARM-based platform we use in this work.

Apart from the pattern matching algorithms included in this work (and summarized in Section 3), there are others that target GPU platforms. FFBF [26] is a filter based approach by Moraru and Andersen that uses Bloom filters to find a subset of the input and the patterns that should be matched together. However, FFBF imposes restrictions on the pattern size and requires long patterns in order to work effectively. On the contrary, the approaches we consider are flexible with respect to the number of patterns. Kouzinopoulos et al. [19] also experiment with pattern matching algorithms on GPUs, using the CUDA framework. In [8], Bellekens et al. present a compressed Aho-Corasick algorithm that improves the bandwidth of data transfers on both NIDS and DNA sequencing workloads. All of the above-mentioned work targets high-end, desktop GPUs. In this work, we focus on embedded GPUs that have a significantly different architecture, as described in Section 4.

On the topic of embedded GPUs, as already mentioned, Aragon et al. [4] implement PFAC in OpenCL and report its performance on two embedded GPUs that have almost the same architecture as the one used in this paper. In [22], Maghazeh et al. benchmark various GPGPU applications on an embedded GPU, concluding that the high energy efficiency of such devices makes them a promising choice for a wide range of workloads such as genetic algorithms and vector similarity (referred to as pattern matching in their work). In [14], Grasso et al. present optimizations techniques for the Mali GPU that allow them to gain significant speedups (both in terms of execution time, as well as energy) for various benchmarks.

Even though this work focuses on the pattern matching algorithms themselves, it is important to mention work that considers the role of GPUs in the NIDS as a whole. Vasiliadis et al. [38] use Aho-Corasick to build a GPU based NIDS. In [39], they also integrate more network processing workloads into a general GPU framework, including flow state management and TCP stream reconstruction. Go et al. [13] experiment with integrated GPUs and show that they provide an appealing alternative for packet processing workloads, including pattern matching. There is also work that considers both the CPU and the GPU and how they can coordinate to better serve the needs of the NIDS. Kim et al. [17] propose NBA, a framework that abstracts the GPU offloading from the programmer and includes load balancing and batching. Their NIDS implementation is also based on Aho-Corasick. Vasiliadis et al. [40] present a system based on Snort that uses all CPU threads and multiple GPU devices. Jamshed et al. [15] present Kargus, a similar parallel design that is based on their own custom NIDS. Papadogiannaki et al. [30] extend the existing work in the field with a scheduler that decides the placement of the different computing tasks of the NIDS, across a heterogeneous platform.

7 CONCLUSION

In this paper, we introduce a fair and thorough co-evaluation of pattern matching algorithms for network intrusion detection on embedded devices with GPUs with the aim to methodologically investigate the algorithm behavior in conjunction with the architectural features available on medium to high-level devices used in deployments for the Internet of Things. We present results from existing approaches, in-house implementations of state of the art algorithms, as well as a new algorithm, *HYBRID*, that combines the benefits from existing designs.

We conclude that GPUs on embedded devices are an attractive alternative to CPUs when it comes to pattern matching for intrusion detection, since the GPU-based algorithms in our benchmarks managed to reduce the overall execution time and energy consumption of pattern matching workloads by up to 2 times. We investigate how algorithm engineering approaches that are based on the platform's architectural features, e.g., shared GPU and CPU memory, affect the performance of various algorithms. Finally, we show that *HYBRID* is a good fit for the GPU, achieving up to 1.4x speedup compared to the best GPU-based baseline. This investigation provides a baseline for the community to further develop algorithms and make standard security tool deployable in IoT devices. Implementations used in the benchmark are available online[2].

ACKNOWLEDGEMENTS

The research leading to these results has been partially supported by 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, by the Swedish Research Council (VR) through the project ChaosNet and the project AgreeOnIT, the Vinnova-funded project "KIDSAM", and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

REFERENCES

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

[2] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. 2004. Generating Realistic Workloads for Network Intrusion Detection Systems. *SIGSOFT Softw. Eng. Notes* 29 (2004), 9.

[3] Appneta. [n. d.]. Sample Captures. http://tcpreplay.appneta.com/wiki/captures.html/ [Accessed: 2018-09-18].

[4] Elena Aragon, Juan M. Jiménez, Arian Maghazeh, Jim Rasmusson, and Unmesh D. Bordoloi. 2014. Pattern Matching in OpenCL: GPU vs CPU Energy Consumption

[2] https://bitbucket.org/mpastyl/acsac_pattern_matching_benchmark_opencl/src/master/

on Two Mobile Chipsets. In *Proceedings of the International Workshop on OpenCL 2013 &#38; 2014 (IWOCL '14)*. ACM, New York, NY, USA, Article 5, 7 pages. http://doi.acm.org/10.1145/2664666.2664671

[5] ARM. [n. d.]. ARM Mali-T628 product page. https://www.arm.com/products/multimedia/mali-cost-efficient-graphics/mali-t628.php. Accessed: 2018-03-14.

[6] Arm. [n. d.]. ODROID-XU3. https://developer.arm.com/graphics/development-platforms/odroid-xu3. Accessed: 2018-05-25.

[7] ARM. 2018. ARM Mali GPU OpenCL, Version 3.0, Developer Guide. https://static.docs.arm.com/100614/0300/arm_mali_gpu_opencl_developer_guide_100614_0300_00_en.pdf. Accessed: 2018-03-14.

[8] Xavier JA Bellekens, Christos Tachtatzis, Robert C Atkinson, Craig Renfrew, and Tony Kirkham. 2014. 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, 302.

[9] Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772. http://doi.acm.org/10.1145/359842.359859

[10] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. 2016. 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, 551–565.

[11] Cisco. 2015. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. White Paper https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf. Accessed: 2018-05-07.

[12] Canadian Institute for Cybersecurity. 2012. UNB ISCX Intrusion Detection Evaluation DataSet. http://www.unb.ca/research/iscx/dataset/iscx-IDS-dataset.html. Accessed: 2016-12-10.

[13] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 83–96. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[14] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez. 2014. Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 123–132.

[15] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. 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, 317–328.

[16] Khronos Group. [n. d.]. OpenCL Overview. https://www.khronos.org/opencl/. Accessed: 2018-03-11.

[17] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 22, 14 pages. http://doi.acm.org/10.1145/2741948.2741969

[18] D. Knuth, J. Morris, Jr., and V. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. arXiv:https://doi.org/10.1137/0206024 https://doi.org/10.1137/0206024

[19] Charalampos S Kouzinopoulos and Konstantinos G Margaritis. 2009. String matching on a multicore GPU using CUDA. In *Informatics, PCI'09. 13th Panhellenic Con. on*. IEEE.

[20] Lincoln Laboratory. 2000. DARPA Intrusion Detection Data Sets. https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-data-sets. Accessed: 2018-09-20.

[21] C. H. Lin, C. H. Liu, L. S. Chien, and S. C. Chang. 2013. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Trans. Comput.* 62, 10 (Oct 2013), 1906–1916.

[22] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng. 2013. General purpose computing on low-power embedded GPUs: Has it come of age?. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 1–10.

[23] Matthew V Mahoney and Philip K Chan. 2003. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *Int. Workshop on Recent Advances in Intrusion Detection*. Springer.

[24] John McHugh. 2000. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by lincoln laboratory. *ACM Tran. on Information and System Security (TISSEC)* 3, 4 (2000), 262–294.

[25] MichaelS. 2015. Gather Scatter Operations. http://insidehpc.com/2015/05/gather-scatter-operations/. Accessed: 2016-12-10.

[26] Iulian Moraru and David G. Andersen. 2012. Exact Pattern Matching with Feed-forward Bloom Filters. *J. Exp. Algorithmics* 17, Article 3.4 (Sept. 2012), 1.08 pages. http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2133803.2330085

[27] Marc Norton. 2004. *White paper: Optimizing pattern matching for intrusion detection*. Technical Report. Snort.

[28] Nvidia. [n. d.]. About CUDA. https://developer.nvidia.com/about-cuda. Accessed: 2018-03-11.

[29] NVIDIA. [n. d.]. Jetson Nano Brings AI Computing to Everyone. https://devblogs.nvidia.com/jetson-nano-ai-computing/. Accessed: 2019-04-17.

[30] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis. 2017. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. *IEEE/ACM Transactions on Networking* 25, 3 (June 2017), 1593–1606.

[31] David E. Sanger and Nicole Perlroth. 2016. A New Era of Internet Attacks Powered by Everyday Devices. https://nytimes.com/2016/10/23/us/politics/a-new-era-of-internet-attacks-powered-by-everyday-devices.html. Accessed: 2018-03-04.

[32] Alessandro Sforzin, Félix Gómez Mármol, Mauro Conti, and Jens-Matthias Bohli. 2016. RPiDS: Raspberry Pi IDS - A Fruitful Intrusion Detection System for IoT. In *UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld, Toulouse, France, July 18-21, 2016*. 440–448.

[33] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. 2012. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security* 31, 3 (2012).

[34] Snort. [n. d.]. Snort++. https://github.com/snort3/snort3. Accessed: 2018-12-21.

[35] Snort [n. d.]. Snort Network Intrusion Detection and Prevention System. https://www.snort.org. Accessed: 2018-09-21.

[36] Ioannis Sourdis and Dionisios Pnevmatikatos. 2003. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In *Field Programmable Logic and Application*, Peter Y. K. Cheung and George A. Constantinides (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 880–889.

[37] C. Stylianopoulos, M. Almgren, O. Landsiedel, and M. Papatriantafilou. 2017. Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization. In *2017 46th International Conference on Parallel Processing (ICPP)*. 472–482.

[38] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. 2008. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Recent Advances in Intrusion Detection*, Richard Lippmann, Engin Kirda, and Ari Trachtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 116–134.

[39] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 321–332. https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis

[40] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. MIDeA: A Multi-parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 12.

[41] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[42] Wang Wei. 2018. Casino Gets Hacked Through Its Internet-Connected Fish Tank Thermometer. https://thehackernews.com/2018/04/iot-hacking-thermometer.html. Accessed: 2019-01-16.

[43] Sun Wu and Udi Manber. 1994. *A fast algorithm for multi-pattern searching*. Technical Report TR-94-17. University of Arizona. Department of Computer Science.