Regular Expression Matching on Graphics Hardware for Intrusion Detection

Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis

Institute of Computer Science, Foundation for Research and Technology – Hellas,
{gvasil,mikepo,antonat,markatos,sotiris}@ics.forth.gr

Abstract. The expressive power of regular expressions has been often exploited in network intrusion detection systems, virus scanners, and spam filtering applications. However, the flexible pattern matching functionality of regular expressions in these systems comes with significant overheads in terms of both memory and CPU cycles, since every byte of the inspected input needs to be processed and compared against a large set of regular expressions.

In this paper we present the design, implementation and evaluation of a *regular expression matching engine* running on graphics processing units (GPUs). The significant spare computational power and data parallelism capabilities of modern GPUs permits the efficient matching of multiple inputs at the same time against a large set of regular expressions. Our evaluation shows that regular expression matching on graphics hardware can result to a 48 times speedup over traditional CPU implementations and up to 16 Gbit/s in processing throughput. We demonstrate the feasibility of GPU regular expression matching by implementing it in the popular Snort intrusion detection system, which results to a 60% increase in the packet processing throughput.

1 Introduction

Network Intrusion Detection Systems (NIDS) are an efficient mechanism for detecting and preventing well-known attacks. The typical use of a NIDS is to passively examine network traffic and detect intrusion attempts and other known threats. Most modern network intrusion detection and prevention systems rely on deep packet inspection to determine whether a packet contains an attack vector or not. Traditionally, deep packet inspection has been limited to directly comparing the packet payload against a set of string literals. One or more string literals combined into a single *rule* are used to describe a known attack. By using raw byte sequences extracted from the attack vector, it is easy to maintain signature sets that describe a large number of known threats and also make them easily accessible to the public.

However, the existence of *loose* signatures [28] can increase the number of false positives. Signatures that fail to precisely describe a given attack may increase the number of matches in traffic that do not contain an actual attack.

Moreover, string literals that are shared between two or more rules will probably conflict at the matching phase and increase the number of false positives. Thus, a large number of well and carefully designed strings may be required for precisely describing a known attack.

On the other hand, regular expressions are much more expressive and flexible than simple byte sequences, and therefore can describe a wider variety of payload signatures. A single regular expression can cover a large number of individual string representations, and thus regular expressions have become essential for representing threat signatures for intrusion detection systems. Several NIDSes, such as Snort [21] and Bro [20] contain a large number of regular expressions to accomplish more accurate results. Unfortunately, regular expression matching, is a highly computationally intensive process. This overhead is due to the fact that, most of the time, every byte of every packet needs to be processed as part of the detection algorithm that searches for matches among a large set of expressions from all signatures that apply to a particular packet.

A possible solution is the use of hardware platforms to perform regular expression matching [9, 24, 7, 18]. Specialized devices, such as ASICs and FPGAs, can be used to inspect many packets concurrently. Both are very efficient and perform well, however they are complex to modify and program. Moreover, FPGA-based architectures have poor flexibility, since most of the approaches are usually tied to a specific implementation.

In contrast, commodity graphics processing units (GPUs) have been proven to be very efficient for accelerating the string searching operations of NIDS [14, 30, 10]. Modern GPUs are specialized for computationally-intensive and highly parallel operations—mandatory for graphics rendering—and therefore are designed with more transistors devoted to data processing rather than data caching and flow control [19]. Moreover, the ever-growing video game industry exerts strong economic pressure for more powerful and flexible graphics processors.

In this paper we present the design, implementation, and evaluation of a GPU-based *regular expression matching engine* tailored to intrusion detection systems. We have extended the architecture of Gnort [30], which is based on the Snort IDS [21], such that *both pattern matching and regular expressions* are executed on the GPU. Our experimental results show that regular expression matching on graphics hardware can provide up to 48 times speedup over traditional CPU implementations and up to 16 Gbit/s of raw processing throughput. The computational throughput achieved by the graphics processor is worth the extra communication overhead needed to transfer network packets to the memory space of the GPU. We show that the overall processing throughput of Snort can be increased up to eight times compared to the default implementation.

The remainder of the paper is organized as follows. Background information on regular expressions and graphics processors is presented in Section 2. Section 3 describes our proposed architecture for matching regular expressions on a graphics processor, while Section 4 presents the details of our implementation in Snort. In Section 5 we evaluate our prototype system. The paper ends with an outline of related work in Section 7 and some concluding remarks in Section 8.

2 Background

In this section we briefly describe the architecture of modern graphics cards, and the general-purpose computing functionality they provide for non-graphics applications. We also discuss some general aspects of regular expression matching and how it is applied in network intrusion detection systems.

2.1 Graphics Processors

Graphics Processing Units (GPUs) have become powerful and ubiquitous. Besides accelerating graphics-intensive applications, vendors like NVIDIA[1] and ATI,[2] have started to promote the use of GPUs as general-purpose computational units complementary to the CPU.

In this work, we have chosen to work with the NVIDIA GeForce 9 Series (G9x) architecture, which offers a rich programming environment and flexible abstraction models through the Compute Unified Device Architecture (CUDA) SDK [19]. The CUDA programming model extends the C programming language with directives and libraries that abstract the underlying GPU architecture and make it more suitable for general purpose computing. CUDA also offers highly optimized data transfer operations to and from the GPU.

The G9x architecture, similarly to the previous G80 architecture, is based on a set of *multiprocessors*, each comprising a set of *stream processors* operating on SPMD (Single Process, Multiple Data) programs. A unit of work issued by the host computer to the GPU is called a *kernel* and is executed on the GPU as many different *threads* organized in *thread blocks*. A fast *shared memory* is managed explicitly by the programmer among thread blocks. The *global*, *constant*, and *texture memory spaces* can be read from or written to by the host, are persistent across kernel launched by the same application, and are optimized for different memory usage [19]. The constant and texture memory accesses are cached, so a read from them costs much less compared to device memory reads, which are not being cached. The texture memory space is implemented as a read-only region of device memory.

2.2 Regular Expressions

Regular expressions offer significant advantages over exact string matching, providing flexibility and expressiveness in specifying the context of each match. In particular, the use of logical operators is very useful for specifying the context for matching a relevant pattern. Regular expressions can be matched efficiently by compiling the expressions into state machines, in a similar way to some fixed string pattern matching algorithms [3].

A state machine can be either a deterministic (DFA) or non-deterministic (NFA) automaton, with each approach having its own advantages and disadvantages. An NFA can compactly represent multiple signatures but may result to

[1] http://developer.nvidia.com/object/cuda.html

[2] http://ati.amd.com/technology/streamcomputing/index.html

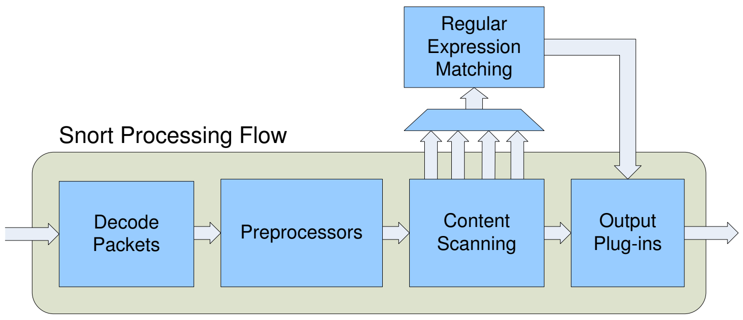


Fig. 1. Regular expression matching in the Snort IDS.

long matching times, because the matching operation needs to explore multiple paths in the automaton in order to determine whether the input matches any signatures.

A DFA, on the other hand, can be efficiently implemented in software—a sequence of n bytes can be matched with $O(n)$ operations, which is very efficient in terms of speed. This is achieved because at any state, every possible input letter leads to at most one new state. An NFA in contrast, may have a set of alternative states to which it may backtrack when there is a mismatch on the previously selected path. However, DFAs usually require large amounts of memory to achieve this performance. In fact, complex regular expressions can exponentially increase the size of the resulting deterministic automaton [6].

2.3 Regular Expression Matching in Snort

Regular expression matching in Snort is implemented using the PCRE library [1]. The PCRE library uses an NFA structure by default, although it also supports DFA matching. PCRE provides a rich syntax for creating descriptive expressions, as well as extra modifiers that can enrich the behavior of the whole expression, such as case-insensitive or multi-line matching. In addition, Snort introduces its own modifiers based on internal information such as the position of the last pattern match, or the decoded URI. These modifiers are very useful in case an expression should be matched in relation to the end of the previous match.

Each regular expression is compiled into a separate automaton that is used at the searching phase to match the contents of a packet. Given the large number of regular expressions contained in Snort's default rule set, it would be inefficient to match every captured packet against each compiled automaton separately. 45% of the rules in the latest Snort ruleset perform regular expression matching, half of which are related to Web server protection.

To reduce the number of packets that need to be matched against a regular expression, Snort takes advantage of the string matching engine and uses it as

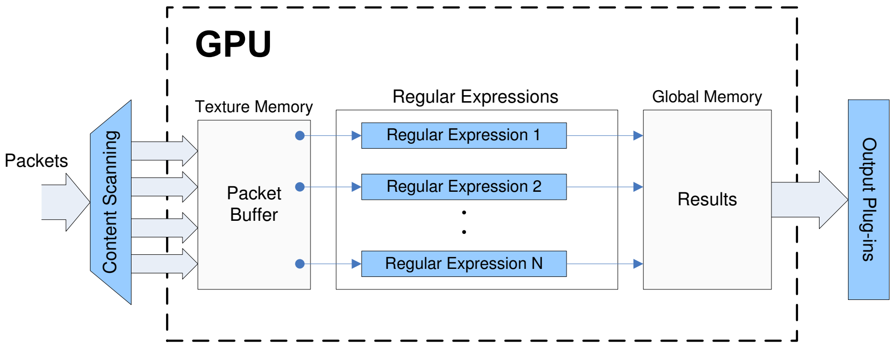


Fig. 2. Overview of the regular expression matching engine in the GPU.

a first-level filtering mechanism before proceeding to regular expression matching. Rules that contain a regular expression operation are augmented with a string searching operation that searches for the most characteristic fixed string counterpart of the regular expression used in the rule.

The string matching engine consists of a set-wise pattern matching algorithm that searches in advance for the fixed string subparts of all regular expressions simultaneously. For a given rule, if the fixed string parts of the regular expressions are not present in a packet, then the regular expression will never match. Thus, fixed string pattern matching acts as a pre-filtering mechanism to reduce the invocation of the regular expression matching engine, as shown in Figure 1.

There are also 24 rules in the latest Snort rule set that do not perform this pre-filtering, but we believe these are cases of poorly written rules. The matching procedure for regular expression matching is invoked only when the subparts have been identified in the packet. For example, in the following rule:

```
alert tcp any any -> any 21 (content:"PASS"; pcre:"/^PASS\s*\n/smi";)
```

the `pcre:` pattern will be evaluated only if the `content:` pattern has previously matched in the packet.

3 Regular Expression Matching on Graphics Processors

We extend the architecture of Snort to make use of the GPU for offloading regular expression matching from the CPU, and decreasing its overall workload. Figure 2 depicts the top-level diagram of our regular expression pattern matching engine. Whenever a packet needs to be scanned against a regular expression, it is transferred to the GPU where the actual matching takes place. The SPMD operation of the GPU is ideal for creating multiple instantiations of regular expression state machines that will run on different stream processors and operate on different data.

Due to the overhead associated with a data transfer operation to the GPU, batching many small transfers into a larger one performs much better than

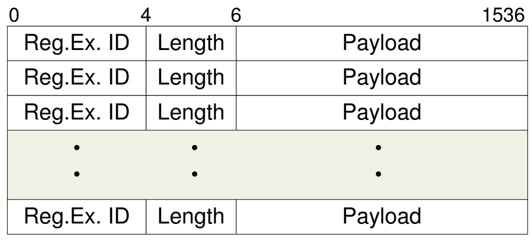


Fig. 3. Packet buffer format.

making each transfer separately, as shown in Section 5.3. Thus, we have chosen to copy the packets to the GPU in batches. We use a separate buffer for temporarily storing the packets that need to be matched against a regular expression. Every time the buffer fills up, it is transferred to the GPU for execution.

The content of the packet, as well as an identifier of the regular expression that needs to be matched against, are stored in the buffer as shown in Figure 3. Since each packet may need to be matched against a different expression, each packet is "marked" so that it can be processed by the appropriate regular expression at the search phase. Therefore, each row of the buffer contains a special field that is used to store a pointer to the state machine of the regular expression the specified packet should be scanned against.

Every time the buffer is filled up, it is processed by all the stream processors of the GPU at once. The matching process is a kernel function capable of scanning the payload of each network packet for a specific expression in parallel. The kernel function is executed simultaneously by many threads in parallel. Using the identifier of the regular expression, each thread will scan the whole packet in isolation. The state machines of all regular expressions are stored in the memory space of the graphics processor, thus they can be accessed directly by the stream processors and search the contents of the packets concurrently.

A major design decision for GPU regular expression matching is the type of automaton that will be used for the searching process. As we have discussed in Section 2, DFAs are far more efficient than the corresponding NFAs in terms of speed, thus we base our design of a DFA architecture capable of matching regular expressions on the GPU.

Given the rule set of Snort, all the contained regular expressions are compiled and converted into DFAs that are copied to the memory space of the GPU. The compilation process is performed by the CPU off-line at start-up. Each regular expression is compiled into a separate state machine table that is transferred to the memory space of the GPU. During the searching phase, all state machine tables reside in GPU memory only.

Our regular expression implementation currently does not support a few PCRE keywords related to some look-around expressions and back references. Back references use information about previously captured sub-patterns which is not straightforward to keep track of during searching. Look-around expressions scan the input data without consuming characters. In the current Snort default rule set, less than 2% of the rules that use regular expressions make use of these features. Therefore our regular expression compiler is able to generate automata for the vast majority of the regular expressions that are currently contained in the Snort rule set. To preserve both accuracy and precision in attack detection, we use a hybrid approach in which all regular expressions that fail to compile into DFAs are matched on the CPU using a corresponding NFA, in the same way unmodified Snort does.

4 Implementation

In this section, we present the details of our implementation, which is based on the NVIDIA G9X platform using the CUDA programming model. First, we describe how the gathered network packets are collected and transferred to the memory space of the GPU. The GPU is not able to directly access the captured packets from the network interface, thus the packets must be copied by the CPU. Next, we describe how regular expressions are compiled and used directly by the graphics processor for efficiently inspecting the incoming data stream.

4.1 Collecting packets on the CPU

An important performance factor of our architecture is the data transfers to and from the GPU. For that purpose, we use page-locked memory, which is substantially faster than non-page-locked memory, since it can be accessed directly by the GPU through Direct Memory Access (DMA). A limitation of this approach is that page locked memory is of limited size as it cannot be swapped. In practice though this is not a problem since modern PCs can be equipped with ample amounts of physical memory.

Having allocated a buffer for collecting the packets in page-locked memory, every time a packet is classified to be matched against a specific regular expression, it is copied to that buffer and is “marked” for searching against the corresponding finite automaton. We use a double-buffer scheme to permit overlap of computation and communication during data transfers between the GPU and CPU. Whenever the first buffer is transferred to the GPU through DMA, newly arriving packets are copied to the second buffer and vice versa.

A slight complication that must be handles comes from the TCP stream reassembly functionality of modern NIDSs, which reassembles distinct packets into TCP streams to prevent an attacker from evading detection by splitting the attack vector across multiple packets. In Snort, the Stream5 preprocessor aggregates multiple packets from a given direction of a TCP flow and builds a single packet by concatenating their payloads, allowing rules to match patterns that span packet boundaries. This is accomplished by keeping a descriptor for each active TCP session and tracking the state of the session according to the semantics of the TCP protocol. Stream5 also keeps copies of the packet data

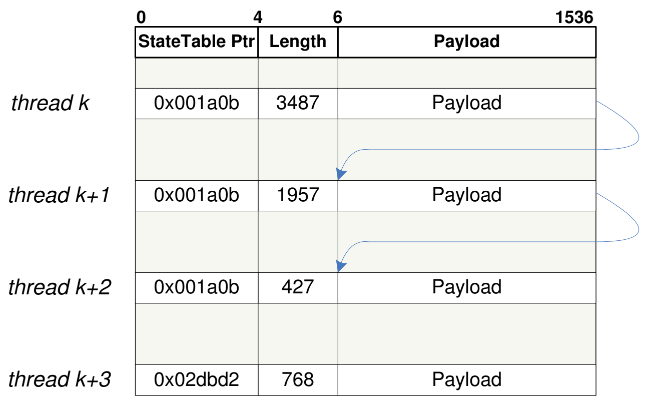


Fig. 4. Matching packets that exceed the MTU size.

and periodically "flushes" the stream by reassembling all contents and emitting a large pseudo-packet containing the reassembled data.

Consequently, the size of a pseudo-packet that is created by the Stream5 preprocessor may be up to 65,535 bytes in length, which is the maximum IP packet length. However, assigning the maximum IP packet length as the size of each row of the buffer would result in a huge, sparsely populated array. Copying the whole array to the device would result in high communication costs, limiting overall performance.

A different approach for storing reassembled packets that exceed the Maximum Transmission Unit (MTU) size, without altering the dimensions of the array, is to split them down into several smaller ones. The size of each portion of the split packet will be less or equal to the MTU size and thus can be copied in consecutive rows in the array.

Each portion of the split packet is processed by different threads. To avoid missing matches that span multiple packets, whenever a thread searches a split portion of a packet, it continues the search up to the following row (which contains the consecutive bytes of the packet), until a *final* or a *fail* state is reached, as illustrated in Figure 4. While matching a pattern that spans packet boundaries, the state machine will perform regular transitions. However, if the state machine reaches a final or a fail state, then it is obvious that there is no need to process the packet any further, since any consecutive patterns will be matched by the thread that was assigned to search the current portion.

4.2 Compiling PCRE Regular Expressions to DFA state tables

Many existing tools that use regular expressions have support for converting regular expressions into DFAs [5, 1]. The most common approach is to first compile them into NFAs, and then convert them into DFAs. We follow the same approach, and first convert each regular expression into an NFA using the Thompson algorithm [29]. The generated NFA is then converted to an equivalent DFA incrementally, using the *Subset Construction* algorithm. The basic idea of subset construction is to define a DFA in which each state is a set of states of the corresponding NFA. Each state in the DFA represents a set of active states in which the corresponding NFA can be in after some transition. The resulting DFA achieves $O(1)$ computational cost for each incoming character during the matching phase.

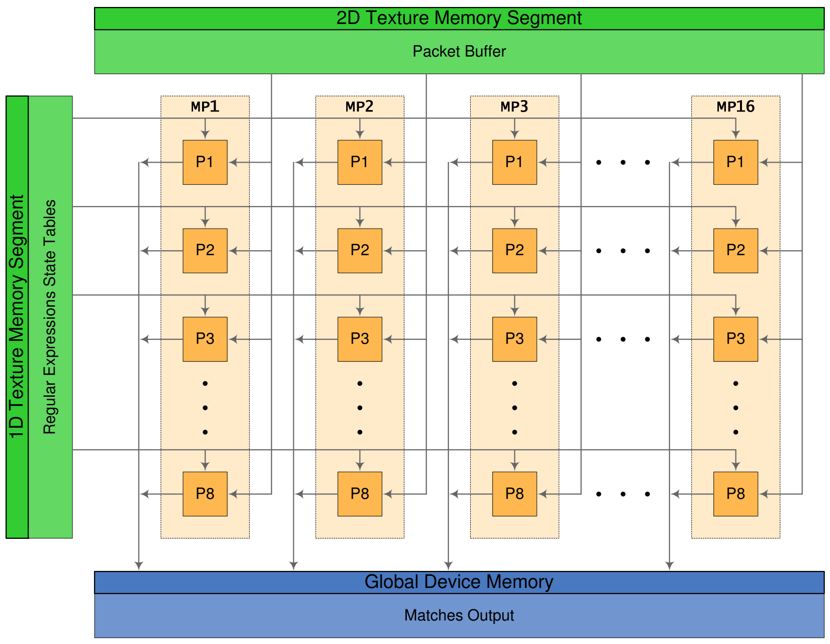
A major concern when converting regular expressions into DFAs is the *state-space explosion* that may occur during compilation [6]. To distinguish among the states, a different DFA state may be required for all possible NFA states. It is obvious that this may cause exponential growth to the total memory required. This is primarily caused by wildcards, e.g. `(.*)`, and repetition expressions, e.g. `(a(x,y))`. A theoretical worst case study shows that a single regular expression of length n can be expressed as a DFA of up to $O(\Sigma^n)$ states, where Σ is the size of the alphabet, i.e. 2^8 symbols for the extended ASCII character set [12]. Due to state explosion, it is possible that certain regular expressions may consume large amounts of memory when compiled to DFAs.

To prevent greedy memory consumption caused by some regular expressions, we use a hybrid approach and convert only the regular expressions that do not exceed a certain threshold of states; the remaining regular expressions will be matched on the CPU using NFAs. We track of the total number of states during the incremental conversion from the NFA to the DFA and stop when a certain threshold is reached. As shown in Section 5.2, setting an upper bound of 5000 states per expression, more than 97% of the total regular expressions can be converted to DFAs. The remaining expressions will be processed by the CPU using an NFA schema, just like the default implementation of Snort.

Each constructed DFA is a two-dimensional state table array that is mapped linearly on the memory space of the GPU. The dimensions of the array are equal to the number of states and the size of the alphabet (256 in our case), respectively. Each cell contains the next state to move to, as well as an indication of whether the state is a final state or not. Since transition numbers may be positive integers only, we represent final states as negative numbers. Whenever the state machine reaches into a state that is represented by a negative number, it considers it as a final state and reports a match at the current input offset. The state table array is mapped on the memory space of the GPU, as we describe in the following section.

4.3 Regular Expression Matching

We have investigated storing the DFA state table both as textures in the texture memory space, as well as on the linear global memory of the graphics card. A straightforward way to store the DFA of each regular expression would be to dynamically allocate global device memory every time. However, texture memory can be accessed in a random fashion for reading, in contrast to global memory, in which the access patterns must be coalesced [19]. This feature can be very



2D Texture Memory Segment
Packet Buffer
1D Texture Memory Segment
Regular Expressions State Tables
MP1
MP2
MP3
MP16
P1
P2
P3
P8
Global Device Memory
Matches Output

Fig. 5. Regular expression matching on the GeForce 9800 with 128 stream processors. Each processor is assigned a different packet to process using the appropriate DFA.

useful for algorithms like DFA matching, which exhibit irregular access patterns across large datasets. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. As we will see in Section 5.3, the texture memory is 2 to 2.5 times faster than global device memory for input data reads.

However, CUDA does not support dynamic binding of memory to texture references. Therefore, it is not feasible to dynamically allocate memory for each state table individually and later bind it to a texture reference. To overcome this limitation, we pre-allocate a large amount of linear memory that is statically bound to a texture reference. All constructed state tables are stored sequentially in this texture memory segment.

During the searching phase, each thread searches a different network packet in isolation, as shown in Figure 5. Whenever a thread matches a regular expression on an incoming packet, it reports it by writing the event to a single-dimension array allocated in the global device memory. The size of the array is equal to the number of packets that are processed by the GPU at once, while each cell of the array contains the position within the packet where the match occurred.

5 Evaluation

5.1 Experimental Environment

For our experiments, we used an NVIDIA GeForce 9800 GX2 card, which consists of two PCBs (Printed Circuit Board), each of which is an underclocked Geforce 8800 GTS 512(G92) video card in SLI Mode. Each PCB contains 128 stream processors organized in 16 multiprocessors, operating at 1.5GHz with 512 MB of memory. Our base system is equipped with two AMD Opteron$^{\text{TM}}$ 246 processors at 2GHz with 1024KB of L2-cache.

For our experiments, we use the following full payload network traces:

U-Web: A trace of real HTTP traffic captured in our University. The trace totals 194MB, 280,088 packets, and 4,711 flows.

SCH-Web: A trace of real HTTP traffic captured at the access link that connects an educational network of high schools with thousands of hosts to the Internet. The trace contains 365,538 packets in 14,585 different flows, resulting to about 164MB of data.

LLI: A trace from the 1998-1999 DARPA intrusion detection evaluation set of MIT Lincoln Lab [2]. The trace is a simulation of a large military network and generated specifically for IDS testing. It contains a collection of ordinary-looking traffic mixed with attacks that were known at the time. The whole trace is about 382MB and consists of 1,753,464 packets and 86,954 flows.

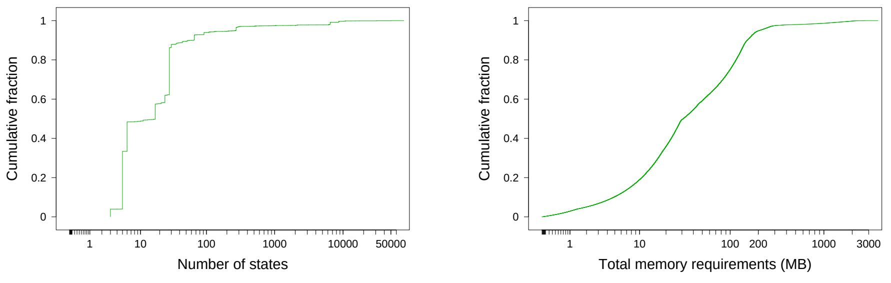
In all experiments, Snort reads the network traces from the local machine. We deliberately chose traces of small size so that they can fit in main memory—after the first access, the whole trace is cached in memory. After that point, no accesses ever go to disk, and we have verified the absence of I/O latencies using the `iostat(1)` tool.

We used the default rule set released with Snort 2.6 for all experiments. The set consists of 7179 rules that contain a total of 11,775 `pcre` regular expressions. All preprocessors were enabled, except the HTTP inspect preprocessor, in order to force all web traffic to be matched against corresponding rules regardless of protocol semantics.

5.2 Memory Requirements

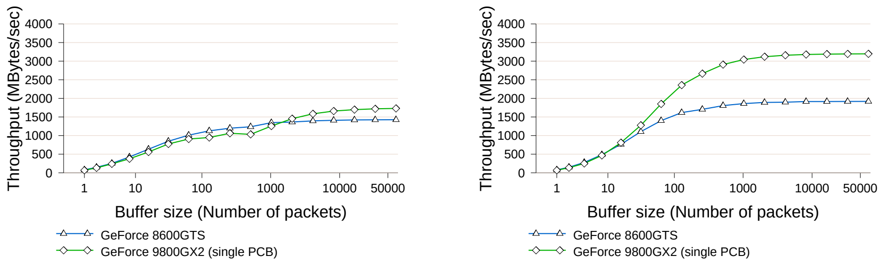
In our first experiment, we measured the memory requirements of our system. Modern graphics cards are equipped with enough and fast memory, ranging from 512MB DDR up to 1.5GB GDDR3 SDRAM. However, the compilation of several regular expression to DFAs may lead to state explosion and consume large amounts of memory.

Figure 6(a) shows the cumulative fraction of the DFA states for the regular expressions of the Snort rule set. It appears that only a few expressions are prone to the state-space explosion effect. By setting an upper bound of 5000 states per regular expression, it is feasible to convert more than 97% of the regular expressions to DFAs, consuming less than 200MB of memory, as shown in Figure 6(b).



(a) (b)

Fig. 6. States (a) and memory requirements (b) for the 11,775 regular expressions contained in the default Snort ruleset when compiled to DFAs.



(a) Virtual memory (b) Page-locked memory

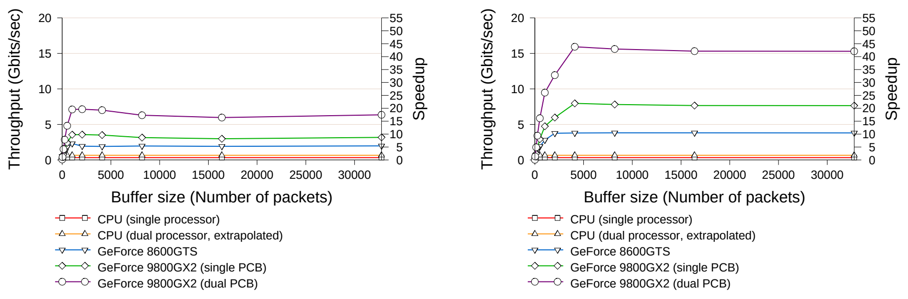
Fig. 7. Sustained throughput for transferring packets to the graphics card using virtual (a) and paged-locked (b) memory.

5.3 Microbenchmarks

In this section, we analyze the communication overheads and the computational throughput achieved when using the GPU for regular expression matching.

Packet transfer performance In this experiment we evaluated the time spent in copying the network packets from the memory space of the CPU to the memory space of the GPU. The throughput for transferring packets to the GPU varies depending on the data size and whether page-locked memory is used or not. For this experiment we used two different video cards: a GeForce 8600 operating on PCIe 16x v1.1, and a GeForce 9800 operating on PCIe 16x v2.0.

As expected, copying data from page-locked memory, despite the fact that can be performed asynchronously via DMA, is substantially faster than non page-locked memory, as shown in Figure 7. Compared to the theoretical 4 GB/s peak throughput of the PCIe 16x v1.1 bus, for large buffer sizes we obtain about 2 GB/s with page pinning and 1.5 GB/s without pinning. When using PCIe 16x



(a) Global device memory (b) Texture memory

Fig. 8. Computational throughput for regular expression matching.

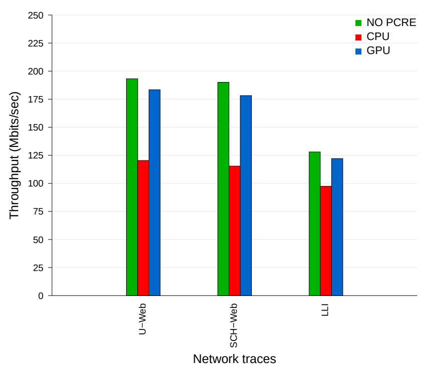
v2.0, the maximum throughput sustained reached 3.2 GB/s, despite the maximum theoretical being 8 GB/s. We speculate that the reason of these divergences from the theoretical maximum data rates is the use of 8b/10b encoding in the physical layer.

Regular expression matching raw throughput In this experiment, we evaluated the raw processing throughput that our regular expression matching implementation can achieve on the GPU. Thus, the cost for delivering the packets to the memory space of the GPU is not included.

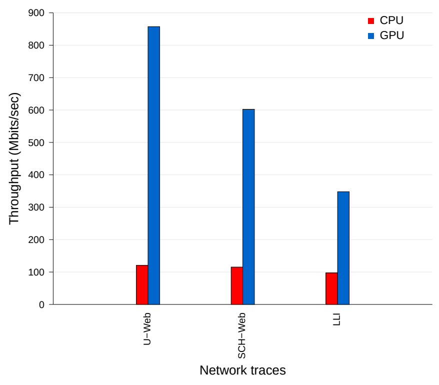
Figure 8 shows the raw computational throughput, measured as the mean size of data processed per second, for both CPU and GPU implementations. We also explore the performance that different types of memory can provide, using both global and texture memory to store the state machine tables. The horizontal axis represents the number of packets that are processed at once by the GPU.

When using global device memory, our GPU implementation operates about 18 times faster than the speed of the CPU implementation for large buffer sizes. The use of texture memory though appears to maximize significantly the utilization of the texture cache. Using texture memory and a 4096 byte packet buffer, the GeForce 9800 achieved an improvement of 48.2 times compared to the CPU implementation, reaching a raw processing throughput of 16 Gbit/s. However, increasing the packet buffer size from 4096 to 32768 packets gave only a slight improvement.

We have also repeated the experiment using the older GeForce 8600GT card which contains only 32 stream processors operating at 1.2GHz. We can see that the achieved performance doubles when going from the previous model to the newest one, which demonstrates that our implementation scales to newer graphics cards.



(a)



(b)

Fig. 9. Sustained processing throughput for Snort using different network traces. In (a) `content` matching is performed on the CPU for both approaches. In (b), both `content` and `pcre` matching is performed on the GPU.

5.4 Overall Snort Throughput

In our next experiment we evaluated the overall performance of the Snort IDS using our GPU-assisted regular expression matching implementation. Unfortunately, the single-threaded design of Snort forces us to use only one of the two PCBs contained in the GeForce 9800 GX2. Due to the design of the CUDA SDK, multiple host threads are required to execute device code on multiple devices [19]. Thus, Snort's single thread of execution is able to execute device code on a single device. It is possible to run multiple instances of Snort dividing the work amongst them, or modify Snort to make it multi-threaded. We are currently in the processes of extending Snort accordingly but this work is beyond the scope of this paper.

We ran Snort using the network traces described in Section 5.1. Figure 9(a) shows the achieved throughput for each network trace, when regular expressions are executed in CPU and GPU, respectively. In both cases, all `content` rules are executed by the CPU. We can see that even when `pcre` matching is disabled, the overall throughput is still limited. This is because `content` rules are executed on the CPU, which limits the overall throughput.

We further offload content rules matching on the GPU using the implementation of GPU string matching from our previous work [30], so that both `content` and `pcre` patterns are matched on the GPU. As we can see in Figure 9(b), the overall throughput exceeds 800 Mbit/s, which is an 8 times speed increase over the default Snort implementation. The performance for the *LLI* trace is still limited, primarily due to the extra overhead spent for reassembling the large amount of different flows that are contained in the trace.

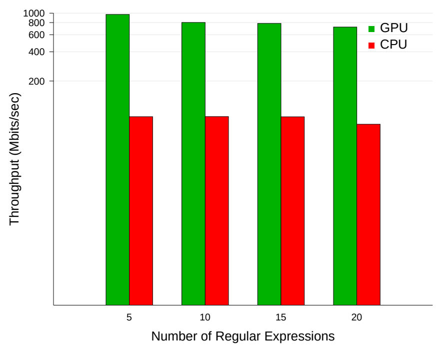


Fig. 10. Sustained throughput for Snort when using only regular expressions.

5.5 Worst-case Performance

In this section, we evaluate the performance of Snort for the worst-case scenario in which each captured packet has to be matched against several regular expressions independently. By sending crafted traffic, an attacker may trigger worst-case backtracking behavior that forces a packet to be matched against more than one regular expressions [25].

We synthetically create worst-case conditions, in which each and every packet has to be matched against a number of regular expressions, by removing all `content` and `uricontent` keywords from all Snort rules. Therefore, Snort's pre-filtering pattern matching engine is rendered completely ineffective, forcing all captured packets to be evaluated against each `pcre` pattern individually.

Figure 10 shows how the CPU and the GPU implementations scale as the number of regular expressions increases. We vary the number of `pcre` web rules from 5 to 20, while Snort was operating on the *U-Web* trace. In each run, each packet of the network trace is matched against all regular expressions. Even if the attacker succeeds in causing every packet to be matched against 20 different regular expressions, the overall throughput of Snort remains over 700 Mbit/s when regular expression matching is performed on the GPU. Furthermore, in all cases the sustained throughput of the GPU implementation was 9 to 10 times faster than the throughput on the CPU implementation.

6 Discussion

An alternative approach for regular expression matching, not studied in this paper, is to combine many regular expressions into a single large one. The combination can be performed by concatenating all individual expressions using the logical union operator [28]. However, the compilation of the resulting single expression may exponentially increase the total number of states of the resulting deterministic automaton [16, 26]. The exponential increase, mainly referred as *state-space explosion* in the literature, occurs primarily due to the inability of the DFA to follow multiple partial matches with a single state of execution [15].

To prevent state-space explosion, the set of regular expressions can be partitioned into multiple groups, which can dramatically reduce the required memory space [31, 16]. However, multiple DFAs require the traversal of input data multiple times, which reduces the overall throughput. Recent approaches attempt to reduce the space requirements of the automaton by reducing the number of transitions [16] or using extra scratch memory per state [26, 15]. The resulting automaton is compacted into a structure that consists of a reasonable number of states that are feasible to store in low-memory systems.

Although most of these approaches have succeed in combining all regular expressions contained in current network intrusion detection systems into a small number of automata, it is not straightforward how current intrusion detection systems (like Snort) can adopt these techniques. This is because most of the regular expressions used in attack signatures have been designed such that each one is scanned in isolation for each packet. For example, many expressions in Snort are of the form `/^.{27}/` or `/.{1024}/`, where `.` is the wild card for *any* character followed by the number of repetitions. Such expressions are used for matching the presence of fixed size segments in packets that seem suspicious. Therefore, even one regular expression of the form `/.{N}/` will cause the relevant automaton to generate a huge number of matches in the input stream that need to be checked against in isolation.

Moreover, the combination of regular expressions into a single one prohibits the use of specific modifiers for each regular expression. For example, a regular expression in a Snort rule may use internal information, like the matching position of the previous pattern in the same rule. In contrast, our proposed approach has been implemented directly in the current Snort architecture and boost its overall performance in a straightforward way. In our future work we plan to explore how a single-automaton approach could be implemented on the GPU.

Finally, an important issue in network intrusion detection systems is traffic normalization. However, this is not a problem for our proposed architecture since traffic normalization is performed by the Snort preprocessors. For example, the URI preprocessor normalizes all URL instances in web traffic, so that URLs like “`GET /%43md.exe HTTP/1.1`” become `GET /cmd.exe HTTP/1.1`. Furthermore, traffic normalization can be expressed as a regular expression matching process [22], which can also take advantage of GPU regular expression matching.

7 Related Work

The expressive power of regular expressions enables security researchers and system administrators to improve the effectiveness of attack signatures and at the same time reduce the number of false positives. Popular NIDSes like Snort [21] and Bro [20] take advantage of regular expression matching and come preloaded with hundreds of regexp-based signatures for a wide variety of attacks.

Several researchers have shown interest in reducing the memory use of the compiled regular expressions. Yu et al. [31] propose an efficient algorithm for partitioning a large set of regular expressions into multiple groups, reducing significantly the overall space needed for storing the automata. Becchi et al. [4] propose a hybrid design that addresses the same issue by combining the benefits of DFAs and NFAs. In the same context, recent approaches attempt to reduce the space requirements of an automaton by reducing the number of transitions [16] or using extra scratch memory per state [26, 15].

A significant amount of work focuses on the parallelization of regular expression matching using specialized hardware implementations [9, 24, 7, 18]. Sidhu and Prasanna [24] implemented a regular expression matching architecture for FPGAs achieving very good space efficiency. Moscola et al. [18] were the first that used DFAs instead of NFAs and demonstrated a significant improvement in throughput.

Besides specialized hardware solutions, commodity multi-core processors have begun gaining popularity, primarily due to their increased computing power and low cost. For highly parallel algorithms, packing multiple cores is far more efficient than increasing the processing performance of a single data stream. For instance, it has been shown that fixed-string pattern matching implementations on SPMD processors, such as the IBM Cell processor, can achieve a computational throughput of up to 2.2 Gbit/s [23].

Similarly, the computational power and the massive parallel processing capabilities of modern graphics cards can be used for non graphics applications. Many attempts have been made to use graphics processors for security applications, including cryptography [11, 8], data carving [17], and intrusion detection [14, 30, 10, 27, 13]. In our previous work [30], we extended Snort to offload the string matching operations of the Snort IDS to the GPU, offering a three times speedup to the processing throughput compared to a CPU-only implementation. In this work, we build on our previous work to enable both string and regular expression matching to be performed on the GPU.

8 Conclusion

In this paper, we have presented the design, implementation, and evaluation of a regular expression matching engine running on graphics processors, tailored to speed up the performance of network intrusion detection systems. Our prototype implementation was able to achieve a maximum raw processing throughput of 16 Gbit/s, outperforming traditional CPU implementations by a factor of 48. Moreover, we demonstrated the benefits of GPU regular expression matching by implementing it in the popular Snort intrusion detection system, achieving a 60% increase in overall packet processing throughput.

As part of our future work, we plan to run multiple Snort instances in parallel utilizing multiple GPUs instead of a single one. Modern motherboards contain many PCI Express slots that can be equipped with multiple graphics cards. Using a load-balancing algorithm, it may be feasible to distribute different flows to different Snort instances transparently, and allow each instance to execute device code on a different graphics processor. We believe that building such *"clusters"* of GPUs will enable intrusion detection systems to inspect multi-Gigabit network traffic using commodity hardware.

Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS. Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos and Sotiris Ioannidis are also with the University of Crete.

References

1. Pcre: Perl compatible regular expressions. `http://www.pcre.org`.
2. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4):262–294, 2000.
3. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.
4. M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12, New York, NY, USA, 2007. ACM.
5. E. Berk and C. Ananian. Jlex: A lexical analyzer generator for java. Online: www.cs.princeton.edu/ appel/modern/java/JLex/.
6. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126, 1986.
7. C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. pages 956–959, 2003.
8. D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. Cryptographics: Secret key cryptography using graphics cards. In *Proceedings of RSA Conference, Cryptographer's Track (CT-RSA)*, pages 334–350, 2005.
9. R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29(3):603–622, 1982.
10. N. Goyal, J. Ormont, R. Smith, K. Sankaralingam, and C. Estan. Signature matching in network processing using SIMD/GPU architectures. Technical Report TR1628, 2008.
11. O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th USENIX Security Symposium*, pages 195–209, Berkeley, CA, USA, July 2008. USENIX Association.
12. J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
13. N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai. A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 62–67.

14. N. Jacob and C. Brodley. Offloading IDS computation to the GPU. In *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference (ACSAC '06)*, pages 371–380, Washington, DC, USA, 2006. IEEE Computer Society.
15. S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 155–164, New York, NY, USA, 2007. ACM.
16. S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, New York, NY, USA, 2006. ACM.
17. G. G. R. I. Lodovico Marziale and V. Roussev. Massive threading: Using GPUs to increase the performance of digital forensics tools. *Digital Investigation*, 1:73–81, September 2007.
18. J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *in FCCM*, pages 31–38, 2003.
19. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 1.1. `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.
20. V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium (SSYM '98)*, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.
21. M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999.
22. S. Rubin, S. Jha, and B. Miller. Protomatching network traffic for high throughput network intrusion detection. In *Proceedings of the 13th ACM conference on Computer and Communications Security (CCS)*, pages 47–58.
23. D. P. Scarpazza, O. Villa, and F. Petrini. Exact multi-pattern string matching on the cell/b.e. processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 33–42, New York, NY, USA, 2008. ACM.
24. R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, 2001.
25. R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 89–98, Washington, DC, USA, 2006. IEEE Computer Society.
26. R. Smith, C. Estan, and S. Jha. Xfa: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, pages 187–201. IEEE Computer Society, 2008.
27. R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.
28. R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 262–271, New York, NY, USA, 2003. ACM.
29. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

30. G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 116–134, 2008.
31. F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, New York, NY, USA, 2006. ACM.