2015 IEEE 34th Symposium on Reliable Distributed Systems Workshops

On the Use of Mobile GPU for Accelerating Malware Detection Using Trace Analysis

Manel Abdellatif, Chamseddine Talhi
Depart. of Software Engineering and IT
École de Technologie Supérieure
Montréal, Québec, Canada
manel.abdellatif.1@ens.etsmtl.ca
Chamseddine.Talhi@etsmtl.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, Québec, Canada
abdelw@ece.concordia.ca

Michel Dagenais
Department of Computer Engineering
École Polytechnique de Montréal
Montréal, Québec, Canada
michel.dagenais@polymtl.ca

Abstract—Malware detection on mobile phones involves analysing and matching large amount of data streams against a set of known malware signatures. Unfortunately, as the number of threats grows continuously, the number of malware signatures grows proportionally. This is time consuming and leads to expensive computation costs, especially for mobile devices where memory, power and computation capabilities are limited. As the security threat level is getting worse, parallel computation capabilities for mobile phones is getting better with the evolution of mobile graphical processing units (GPUs). In this paper, we discuss how we can benefit from the evolving parallel processing capabilities of mobile devices in order to accelerate malware detection on Android mobile phones. We have designed and implemented a parallel host-based anti-malware for mobile devices that exploits the computation capabilities of mobile GPUs. A series of computation and memory optimization techniques are proposed to increase the detection throughput. The results suggest that mobile graphic cards can be used effectively to accelerate malware detection for mobile phones.

Index Terms—*Malware detection, Parallel processing, Multi-pattern matching, Trace analysis of mobile phones*

I. Introduction

Smartphones or mobile phones are more and more used in personal and business life. Their popularity has made them an attractive target for malicious attacks. In fact, the number of malicious software and threats is growing significantly for mobile devices. Recently, Kaspersky Lap reported around 10 million malwares were detected between 2012 and 2013, where 98.10% of all detected malwares in 2013 targeted Android systems [1].

To ensure a high security level, a typical anti-malware matches streams of data, generated from mobile devices, against a large set of known malware signatures, using multi-matching algorithms. Most of these algorithms use a Deterministic Finite Automata (DFA) structure. The main advantage of using a DFA structure is that we can check the presence of malicious signatures in a single pass of the input data stream. As the number of malwares grows, the number of signatures also increases, hindering scalability of mobile anti-malware systems due to reduced memory and computing capabilities of mobile devices. A common solution is to offload malware detection processing to an external server. The reliance on an external server, however, exposes malware detection systems to connectivity problems.

In this paper, we explore how mobile GPUs (Graphics Processing Units), combined with parallelization techniques, can be used to design an anti-malware system that resides on the mobile device itself. Both computational capabilities and memory specifications of mobile GPUs are rapidly evolving, which makes more intensive applications and GPGPU (General-Purpose GPU) processing possible. For instance, the performance of ARM's mobile graphic cards has been improved five times in the last two years [2]. As the System-on-Chip (SoC) industry gained momentum from growing cell phone sales, ARM introduced more sophisticated GPUs like Mali T-658, which is one of the most performant ARM GPUs for mobile devices. It is scalable up to eight cores and gained up to 10x graphics performance compared to mainstream Mali-400 MP implementations [2].

In this paper, we discuss how we can effectively use the power of parallel processing on mobile GPUs to enhance the security level of the whole system. To our knowledge, this is the first time the use of mobile GPUs for security is attempted. More precisely, we show how mobile GPUs can be used to accelerate malware detection using multi-pattern matching techniques. We also investigate techniques for compacting DFA structures in order to scale over the reduced memory of mobile GPUs.

In summary, the main contributions of the paper are as follows: We have designed and implemented a high performance parallel host-based anti-malware on mobile GPU using behavioral detection techniques. We implemented a series of optimizations to deal with low memory of mobile devices and the ever-increasing computing and memory requirements of malware detection. In order to improve the performance of our prototype, we focus on the efficient use and placement of the different data in the hierarchical mobile GPU memory and reduce the average latencies of memory access.

II. Background

Multi-pattern matching algorithms have extensively been used by intrusion detection systems (IDS) and malware scanners. GPUs are used in accelerating multi-pattern matching due to their high efficiency level and the evolving computation capabilities compared to CPUs. In this section we introduce the Aho-Corasick (AC for short) [4] multi-pattern matching algorithm which is the core of our detection framework. Then, we introduce the OpenCL programming model that we used as well as some mobile malware detection paradigms.

A. Aho-Corasick Pattern Matching Algorithm

The Aho-Corasick [4] algorithm has been widely used for multiple pattern matching. This algorithm is simple and capable to locate a finite set of key words within a given input stream in a single pass. The algorithm consists of two steps: pre-processing and processing. During pre-processing, a

978-1-5090-0092-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SRDSW.2015.18

deterministic finite automaton structure is built from given patterns. Then, in the processing step, the engine detects the presence of patterns from an input string in a single pass.

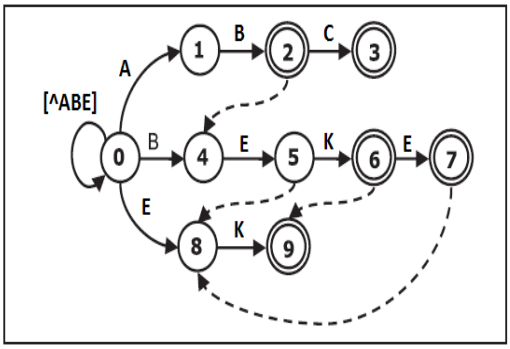

Fig. 1. AC state machine relative to patterns "AB", "ABC", "BEKE", and "EK".

In order to reduce outgoing transitions of each state, AC DFA integrates another kind of transitions, defined as failure transitions. Failure transitions are represented by dotted lines in Figure 1. They are used to backtrack any pattern that starts from any position of the input string. The DFA structure can be described by three tables: the state transition table (where valid transitions are stored), the failure transition table containing the reference of the different failure transitions in the DFA, and finally the output states table where final states are stored. Given the input character from an input stream and the current state of the finite state machine, a valid transition is checked by the machine to determine the next state. If there is no valid transition, a redirection to the state pointed by the failure transition is done. So the engine reads the same input character until it a valid transition is found.

B. *OpenCL programming model*

OpenCL [3] is an open industry standard and a framework for parallel programming of heterogeneous systems composed of devices with different capabilities such as CPUs and GPUs. The OpenCL platform model consists of a host and one or more OpenCL devices. Each device contains one or more compute unit. Each compute unit contains several processing elements. The host offloads parallel tasks to a device within special functions called kernels that are compiled at runtime by an OpenCL driver. Parallel jobs are executed by threads called work items. A hierarchical memory model is defined by OpenCL containing several types of memories like global, local and private memories. In order to achieve higher performance and maximum use of the limited computation resources on a mobile platform, partitioning the tasks between CPU and GPU, exploring efficient algorithmic parallelism, and optimizing the memory access are needed to be carefully considered.

III. ARCHITECTURE

Our framework, which is illustrated by Figure 2, is divided into three blocks: pre-processing, trace collection, and processing block.

The pre-processing block is located outside the mobile phone. It uses as input malware signatures. In our architecture, we use a combination of patterns of raw system calls reflecting malicious behaviour as signatures [5]. We can also use for example bytecode signatures or permissions' patterns. We convert malware signatures into DFA structure that will be used later by a multi-pattern matching algorithm in order to detect the presence of a malware on the device. With a DFA structure, we can check the presence of malicious signatures in a single pass of the input data stream.

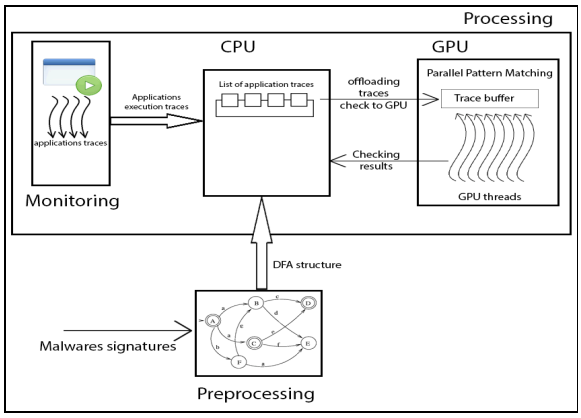

Fig. 2. The framework architecture

Figure 3 shows the transition states matrix TSM that illustrates the AC automaton structure. The row index corresponds to the automaton states and the column index denotes system calls index. Thus, for a given state *i* and an input system call *j* TSM[i][j] indicates the next state to reach. As the number of malwares grows, the number of signatures also increases. As a result, the DFA structure becomes memory consuming and needs to be compacted in order to scale with the small device memory. Many DFA compacting techniques can be exploited such as eliminating failure transitions or using special state encoding scheme to allow memory efficient use of DFA. This issue will be detailed in the next section.

The second block of our architecture is the trace collection block. It is responsible for recording application events and their behaviour by tracking the different system calls made by the processes on the embedded devices at runtime. We use for this goal the Strace tool [6], which is a useful diagnostic and debugging tool that records and intercepts system calls with their arguments. Only raw system calls are kept in our trace files. These traces will be scanned in parallel by the processing block in order to detect a malicious behaviour.

The core block of our architecture is the processing block. It is responsible for analysing application traces and communicating data to the GPU for further processing. The CPU offloads the stream of an application's traces to the GPU. The input stream is divided into several segments. Every work-group is responsible for scanning a given segment. Each byte of the input stream segment is allocated to a thread that checks the presence of malicious pattern from its starting position. If there is no valid transition, the thread terminates and starts the check from another position which is equal to the current starting position plus the total thread number of the processing block. As a result, there is no need to keep failure transitions on the DFA since GPU threads do not need to backtrack the state machine.

In this section, we discuss several optimization techniques in order to increase the performance of mobile malware detection using GPUs. These optimizations are structured into two parts: host code optimizations and kernel code optimizations.

A. *Host code optimizations*

Total memory requirements optimization: With the ever increasing number of malware signatures and the small

memory size of mobile GPUs, we need to optimize memory usage in order to achieve a high security level on a mobile phone. In our framework, three DFA structures compacting techniques are applied to deal with more malware signatures. The first one consists of eliminating failure transitions since we allocate a thread for each input string byte that checks the presence of a pattern from its start position. Then we eliminate the final states table by reordering the numbering of such states and allocating for each one a number greater than the total states number [7]. Finally, we apply P3FSM [8] algorithm on the DFA structure to have a more compacted dataset. This technique offers an effective coding of state machine and deals with the excessive memory requirement of a DFA structure.

System calls

States	open	close	read	...	write	mssget	...	brk
0	1	2	5	0	0	3	0	4
1	6	0	0	0	7	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	8	0	0
4	0	0	9	0	0	0	0	0
5	0	10	0	0	12	0	0	11
...	...	...	...	...	...	...	...	...

Fig. 3. State Transition Matrix

IV. Optimization Techniques

Load balancing configuration: It is an important aspect of OpenCL device processing elements to be performed on both host and device levels. At the host level, we have to choose carefully how to send trace files to the device in order to perform parallel scan. Three scenarios are considered and detailed in the next section. For the device side, every kernel is executed with the specification of the work-items distribution size. This is performed by specifying tow parameters, which are the global work size and the local work size. The first one is the number of work-items to be processed for each dimension and the second one is the number of work-items in a work-group for each dimension. To achieve high performance, it is recommended to maximize the global work size in order to fully utilize the GPU. For the local work size, the OpenCL developer's guide recommends to put it to Null if no data is shared between work-items, and let the driver device determine the most suitable work group size value. In our case, work-items are sharing data. So we have to determine explicitly local work size in order to enhance the performance of our architecture.

B. Kernel code optimizations : effective use of memory types:

Typical GPUs have hierarchical memory model architecture. There are four types of memories: global, constant, local and private. Global memory is visible to all the compute units on a device. All transfers between the host and the device are transfers to and from the device's global memory. Constant memory is also visible to all of the compute units on the device. In addition, any element of constant memory is accessible by all work-items. Local memory is a memory that belongs to a compute-unit and shared by all the work-items within a work-group. The private memory belongs to a work-item and is not accessible for other work-items. The access to constant memory is much faster than global memory access. That's why we have to choose carefully where to locate our data in order to maximize the framework performance.

V. Evaluation

In this section, we evaluate the effectiveness of our architecture to detect malwares on a mobile device using GPU and parallel processing. For our experiments, we used Qualcomm Snapdragon 801 [9], quad-core CPU at 2.5 GHz and a Qualcomm Adreno 330 GPU integrated on a Sony Xperia mobile phone. We use, as benchmark, a dataset of malicious system calls patterns that we generated from different malware families. To evaluate the performance of our framework, we measure the throughput and acceleration rate over the sequential approach. The adjustment of threads' number is studied in order to guarantee the maximum use of GPU and achieve higher throughput. To assess memory usage and throughput, several configurations are measured in our experiments. In our experiments, the system throughput is defined as:

$$Throughput = Input\ Stream\ Size / (T_{host/device}+T_{GPU}+T_{device/host}) \quad (1)$$

where $T_{host/device}$ is the transfer data time from host to the GPU, T_{GPU} is the processing time of input stream data on GPU, and $T_{device/host}$ is the transfer data time from GPU to the host.

The number of threads: In this experiment, we focus on the regulation of local work-group size. It must be specified before queuing any kernel that is executed on the device. For each configuration, we calculate the relative throughput as described by Equation (1). Figure 4 shows that the best throughput is obtained with 16 threads per work-group. Then we notice that the more we increase the number of thread, the worse the throughput.

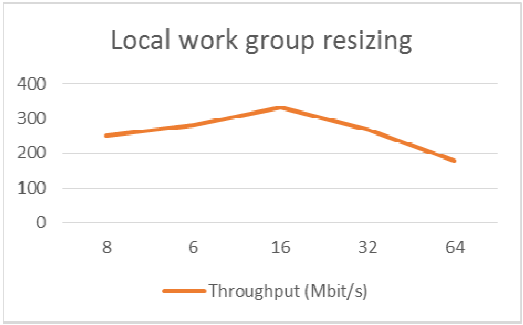

Fig. 4. Local work-group resizing

Memory types: We compared several memory configurations that are listed in Table 1. The data used are: *transition_table* which contains information about the different transitions of the DFA structure, the *input_buffer* which contains application traces that will be analyzed and the *result_buffer* that contains processing results. As results must be sent to the CPU and written to the *result_buffer,* it always must be placed on the global memory. Both constant and local memory size is much lower than the GPU global memory. As the size of *transition_table* is important, we cannot place it as a full block in the constant memory neither in local memory. Only its first part *transition_tableP1* is placed on these memory types for some configurations listed on Table 1. The second part of the table *transition_tableP2* is located in global memory for the same configurations.

Table 1. Different memory allocation configurations for the framework data structures

Memory Config.	Global Memory	Constant Memory	Local Memory
Conf1	*transition_table input_buffer result_buffer*	NA	NA
Conf2	*transition_table result_buffer*	NA	*input_buffer*
Conf3	*transition_table result_buffer*	*input_buffer*	NA
Conf4	*transition_tableP2 result_buffer*	*transition_tableP1*	*input_buffer*
Conf5	*transition_tableP2 input_buffer result_buffer*	*transition_tableP1*	NA
Conf6	*transition_tableP2 result_buffer*	*input_buffer*	*transition_table P1*

Figure 5 shows the relative throughput of the different configurations. As we can see, the best throughput is obtained with Config4. The constant memory access time is faster than the access time of global memory. Comparing to the second configuration, with the use of constant memory, a gain of around 19% in performance is obtained. We keep this type of configuration in the following experiments.

Acceleration: We compare the performance of our framework with one that does not parallel processing. As we can see in Figure 5, all configurations of the parallel processing perform better than the serial processing. An acceleration of around three times is obtained with the parallel processing on the mobile GPU using Conf4.

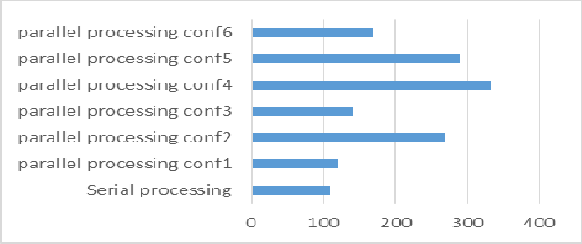

Fig. 5. Serial processing throughput vs parallel processing

Applications trace files scanning management: In this section, we experiment with different scenarios of scanning trace files for every application. The first scenario is to consider a fixed length GPU input buffer to which we send sequentially trace files. The input buffer is dedicated to only one trace file per application at the same time. Every application traces will be profiled in parallel in order to detect a malicious behavior.

The second scenario is almost the same as the first one. The only difference consists in the use of the input buffer. We send sequentially trace files to the input buffer. If, at a given time, there is a space left in the input buffer we send immediately the following application trace file. As a result the input buffer will be always fully used except at the end of scanning the final application trace file.

The third scenario is to consider also a fixed length GPU input buffer, but that has multiple entries. In other words, we send to the input buffer multiple application trace files simultaneously and profile them in parallel. In order to estimate the optimal number of applications that can be scanned in parallel we collected system calls traces from 100.

Android applications were executed normally during 3 minutes each on a Sony Xperia Z smartphone operating on Android 4.4.4. For the third scenario, we considered input stream buffers divided by 5, 10 and 15 segments, and for every buffer each segment is dedicated to an input application trace file. GPU profiling threads are equally distributed for each segment. More threads will be allocated per input buffer segment in order to have a balanced distribution of the parallel scanning processing. We found that the third scenario performs well when the input buffer is divided to 10 segments. As a result, the optimal number of applications that we can scan in parallel with this configuration is 10.

In order to study the effectiveness of the three scenarios we considered different sizes of input stream buffers. As we can see in Figure 6, the third scenario has always the best throughput. In fact, for the first scenario, there is waste of allocated resources because the input buffer is not fully exploited. Moreover for the tow first configurations more data transfers are required to process all the application traces. As a result, the processing overhead becomes more important with such scenarios. On the other hand, the parallel processing of applications trace files simultaneously gives much less processing overhead due to its effectiveness in exploiting the GPU input buffer allocation and processing. We notice that the bigger the input buffer is, the faster the total execution time become. In fact, several data large transfers between the CPU and GPU are much better than many small ones. Finally, we notice that the framework throughput is dominated by data transfers between the host and the device which consist of 70% of the total processing time.

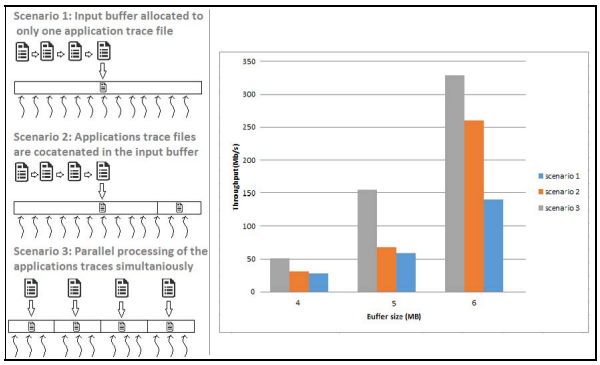

Fig. 6. Performance of the trace files scanning scenarios

Table 2. Memory requirement of PFAC and P3FSM

Number of patterns	PFAC (KB)	P3FSM (KB)
2000	67677	8922
2200	74398	9234
10000	678937	50765
16000	806554	60432
17600	809321	74380

Memory requirement: Storing a DFA structure on the GPU is memory consuming especially that mobile GPU memory is small. Table 2 lists the difference in memory requirement between PFAC DFA and P3FSM. The best result is obtained with P3FSM that compacts the DFA structure by around 10 times comparing to standard PFAC DFA. With PFAC, the limit number of patterns that we can work with on our mobile GPU is 2200 patterns. Applying P3FSM compacting technique allows as to work with 17600 patterns witch is much better.

VI. Related Work

Parallel processing techniques have been widely used over the years in order to improve the performance of multi-pattern

matching algorithms and this is due to the rapid advent of GPGPUs. Both hardware and software techniques are used to accelerate multi-pattern searching. On one hand, common hardware supported techniques use GPU [7][10][8], FPGA [11] as well as Cell/B.E processors [12]. Parallel multi-pattern matching software based approaches [13][14] exploits multicore processors to accelerate the overall processing.

Perhaps, the first implementation of parallel regular expression searching and multi-pattern matching was in Gnort [15][16]. For performance issues, Gnort utilizes a multi-pattern matching technique that uses a high memory DFA structure. A maximum throughput of 2.3 GB/s is achieved. Many studdies have been conducted to optimize the Aho-Corasick [4] string matching algorithm. An example is the Parallel Failureless-AC Algorithm (PFAC) [8]. There, all failure transitions of the DFA were removed so that there is no need to backtrack the state machine used, reducing the complexity of the algorithm. Arudchutha et al. [13] proposed an adaptation of the Aho-Corasick algorithm for multicore CPUs using POSIX thread utility. The pattern-set is divided into smaller ones and allocated for each CPU thread. This approach performs better than Herath et al. [14] implementation where they applied the same techniques but considered smaller deterministic state machines with failure transitions. Tran et al. [17] proposed a parallel implementation of the Aho-Corasick algorithm on an Nvidia desktop GPU that focuses on efficient scheduling of the off-ship global memory loads and the storage in the desktop GPU global memory. A speed-up of 222x was achieved compared to a sequential version of the same algorithm running on a 2.2Ghz Core2Duo Intel processor. Huang et al. [10] implemented a Wu-Manber-like multi-pattern matching algorithm on GPUs and achieved a maximum speed twice as fast as the modified Wu-Manber algorithm used in Snort [18]. Vasiliadis et al. [19] worked on a massively parallel antivirus engine based on ClamAV [20], called Gravity. A parallel pre-scanning of patterns is done through the Geforce GTX GPU, reaching a throughput of 40GB/s.

Many DFA compacting techniques have been proposed in the literature. Compression and bitmap [21] were proposed to improve the memory efficiency of the Aho-Corasick algorithm. Tan and Sherwood [22] introduced a memory-efficient, multi-pattern matching engine based on the bit-split techniques and string partitioning. Vespa et al [8] proposed a memory-efficient, portable and scalable string matching engine called P3FSM. One entry in memory is required for each state. The code for each state contains all the information about the possible next state.

VII. Conclusion

In this paper, we have designed and implemented a parallel host-based anti-malware for Android mobile devices based on the use of GPUs. Our framework capitalizes on the use of the highly threaded architecture of mobile GPUs, as well as the parallel nature of malware scanning to achieve end-to-end throughput in the order of 333 Mbits/s. This result is three times faster than the serial version running on a mobile CPU. To accelerate mobile malware detection on GPU, we used different types of GPU memories and explored the optimal buffer size for input streams data as well as the best scanning process scenario. In order to overcome the problem of the low memory of mobile GPUs and to deal with more malware signatures, series of memory compacting techniques were applied.

Acknowledgment

This research is partly supported by a grant from NSERC, DRDC Valcartier (QC), and Ericsson Canada.

References

[1] V. Chebyshev, and R. Unuchek, "Mobile Malware Evolution: 2013" White paper available on http://securelist.com/analysis/kaspersky-security-bulletin/58335/ mobile-malware-evolution-2013/

[2] "ARM," at http://www.arm.com/products/multimedia/mali-cost-efficient-graphics/index.php

[3] "The OpenCL 1.2 specification," at http://www.khronos.org/opencl, 2012.

[4] A. Aho and M. Corasick, 'Efficient string matching: an aid to bibliographic search', Commun. ACM, vol. 18, no. 6, pp. 333-340, 1975.

[5] Y-D Lin, Y-C Lai, C-H Chen, H-C Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," Elsevier Computers & Security, vol. 39, p. 340-350, 2013.

[6] STrace manual, Retrieved February 20, 2015, http://man7.org/linux/man-pages/man1/strace.1.html

[7] Cheng-Hung Lin; Sheng-Yu Tsai; Chen-Hsiung Liu; Shih-Chieh Chang; Shyu, J.-M., "Accelerating String Matching Using Multi-Threaded Algorithm on GPU," Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE , vol., no., pp.1,5, 6-10 Dec. 2010.

[8] Vespa, L.; Mathew, M.; Ning Weng, "P3FSM: Portable Predictive Pattern Matching Finite State Machine," Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, vol., no., pp.219, 222, 7-9 July 2009.

[9] "Qualcomm," at https://www.qualcomm.com/products/snapdragon/processors/801

[10] N. Huang, H. Hung, S. Lai, Y. Chu, and W. Tsai, "A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems," In Proc. of the International Conference on Advanced Information Networking and Applications, pp. 62-67, 2008

[11] B. W. Watson and G. Zwaan, "A taxonomy of keyword pattern matching algorithms," Computing Science Note 92/27, Eindhoven University of Technology, The Netherlands, 1992.

[12] D. P. Scarpazza, O. Villa, F. Petrini, "Exact multi-pattern string matching on the cell/be processor," In Proc. of the 5th Conference on Computing Frontiers, pp. 33-42, 2008.

[13] S. Arudchutha, T. Nishanthy, R. G. Ragel, "String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm," In Proc. of the 8th IEEE International Conference on Industrial and Information Systems, pp. 231-236, 2013.

[14] D. Herath, C. Lakmali, R. Ragel, "Accelerating string matching for bio-computing applications on multi-core CPUs," In Proc. of the 7th IEEE International Conference on Industrial and Information Systems, pp. 1-6, 2012.

[15] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," In Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection, 2008.

[16] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," In Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection, 2009.

[17] N. P. Tran, M. Lee, S. Hong, J. Choi, "High throughput parallel implementation of Aho-Corasick algorithm on a GPU," In Proc. of the IEEE 27th International Symposium on Parallel and Distributed systems (Workshops & PhD Forum), pp. 1807-1816, 2013.

[18] M. Roesch, "Snort—Lightweight Intrusion Detection for Networks," In Proc. of 15th Systems Administration Conference, 1999.

[19] G. Vasiliadis, S. Ioannidis, "Gravity: a massively parallel antivirus engine," In International Symposium on Recent Advances in Intrusion Detection, pp. 79-96, 2010.

[20] "CLAMAV," at http://www.clamav.net

[21] N. Tuck et al., "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," INFOCOM, pp. 333–40, 2004.

[22] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," In Proc. of the 32nd Annual International Symposium on Computer Architecture, pp. 112–22, 2005.