GPUs and Malware: The Good and The Bad

Kostas Papagiannopoulos
Kerckhoff's Institute
kostaspap88@gmail.com

September 27, 2012

Abstract

Malware and Antivirus software are constantly engaging in a war of evasion and detection. Code armoring techniques are used to obfuscate the viral code in order to evade detection and both new and old detection techniques require additional computational horsepower to address these new threats. In this paper, we discuss how the *Graphics Processing Unit* (GPU) can be utilized to serve both purposes: *evasion* (the Bad), by improving obfuscation techniques and *detection* (the Good), by employing their parallel architecture to tackle computationally intensive antivirus tasks such as pattern matching.

1 Introduction

Malware, or malicious software, is software used or created to disrupt computer operation, gather sensitive information, or gain access to private computer systems. It can appear in the form of code, scripts, active content, and other software components [2]. Traditionally, malware attack and defense revolved around the CPU with attackers putting effort on obfuscating their malware code and the defenders focusing on detecting threats using signatures that revealed malicious software. However, recent advances in malware obfuscation techniques have rendered several commercial antivirus software incapable of detection [3]. In this direction, malware writers have been able to employ the graphics processing unit (GPU) in order to further obfuscate their attacks. On the other side, dynamic analysis techniques are the defender's promising new technologies in malware detection and they could definitely tap the GPU's computational power to serve their needs. In this paper we aim to provide a small overview of the GPU architecture (Section *2*) and then continue by demonstrating how the GPU can be either the Bad or the Good factor in the modern virus - antivirus landscape (Sections *3*,*4*).

2 Graphic Processing Units - CUDA Architecture

General purpose computing on graphics processing units (GPGPU) has transformed modern GPUs into highly parallel, multithreaded, manycore processors with impressive computational power and very high memory bandwidth. The Compute Unified Device Architecture (CUDA) [1] was introduced by NVIDIA and consists of a framework that provides functions to control the GPU from the host. Every CUDA application consists of a serial program running on the CPU (with the common x86 instruction set) and a parallel *kernel*, running on the GPU, on GPU-specific assembly language. The GPU architecture intends to create a device capable of high stream or throughput computing. As a result, deep cache hierarchy is not the focal point at GPUs and the device memory supports very high data bandwidth using a wide data path.
A typical kernel execution involves partitioning the data into *threads*, organized in *thread blocks*, transferring the data to the GPU memory, executing in parallel using all the GPU cores and then copying the results back to host memory [1, 5]. In section *3* we will demonstrate how the nature of GPU assembly helps malware writers and in section *4* we will discuss how NVIDIA graphics cards, CUDA and their efficiency in parallel processing can help antivirus detection.

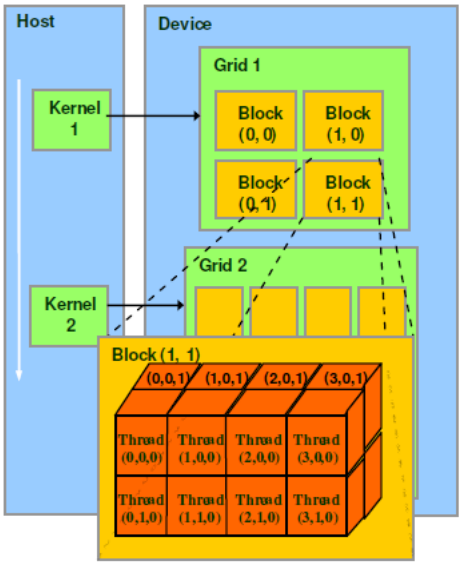

Figure 1: The CUDA thread and block mode, host and device memory [1].

3 GPU Powered Malware

Two of the most prevalent forms of code obfuscation use by malware are *unpacking* and *run-time polymorphism*. In this section, we will explore the ability of graphic processing units (GPUs) to implement these two code armoring techniques, estimate the threats that emerge and assess the capability of antivirus software to detect such threats.

3.1 Malware unpacking and polymorphism

Malware unpacking. This technique consists of unpacking, i.e. decompressing or decrypting malware data into malware code at runtime. Live and randomized forms of decryption allow malware both to evade several static detection techniques and produce new variants of the malware software component [4]. In a GPU context, the unpacking is performed as follows: the malware execution begins on the CPU and the malicious data packet is stored on host memory (i.e.RAM) in encrypted form. Subsequently, the encrypted data is transfered to device memory (GPU) where decryption is performed in parallel fashion and the malicious code is unpacked. Finally, control is transfered back to the CPU where the malicious code gets executed [5]. Existing AES implementations on GPUs manage to perform this task substantially faster than CPUs of equal cost [6].

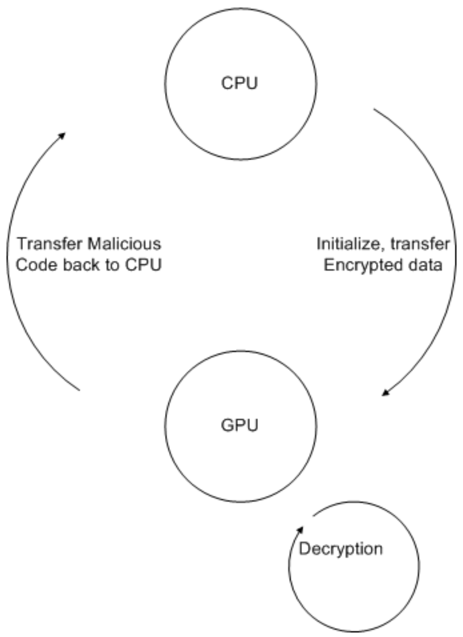
CPU
Transfer Malicious Code back to CPU
Initialize, transfer Encrypted data
GPU
Decryption

Figure 2: Malware unpacking: encrypted data is transfered to GPU, is decrypted and returned to CPU for execution.

Malware polymorphism. Taking the unpacking idea one step further, we may employ the GPU encryption/decryption scheme to implement runtime polymorphism. After the unpacking is complete, the malicious code is stored and is ready to be executed in host memory. Thus, it is rendered vulnerable to detection techniques and a memory dumper can snapshot the running process and detect the unpacked malicious code [7]. To further obfuscate the code, the GPU malware may decrypt the required parts of code (a small portion, e.g. one function at a time), execute them and re-encrypt them in order to minimize the code exposure and offering the polymorphism required for evasion [5].

3.2 GPU Malware against the current detection environment

The factors that influence the potency and lifetime of malware is no other than their ability to evade the current state of affairs [5]. So, how do GPU powered malware fare against modern antivirus technology? Old-fashioned *signature-only* antiviral software will encounter severe problems in detecting an unpacking and polymorphic virus and we maintain that GPU malware is able to evade it.

When it comes to *dynamic analysis* techniques, there are still issues. A core characteristic of GPU malware is that their assembly instructions do not follow the usual IA-32 instruction set architecture and their instruction set depends on the graphics unit, e.g. PTX assembly for the NVIDIA CUDA devices. Many dynamic detection techniques use virtualization/emulation techniques to a) monitor the system system calls [8],b) to execute a malicious code in advance [10] or c) to unpack executables [9]. The usage of PTX assembly creates a failure point for these dynamic detection techniques, since the executable (written in PTX assembly) cannot be emulated. This feature can be deployed in conjunction with other anti-emulating methods to detect/hinder virtualized environments, such as software interrupts or time locks [7]. Notably, we mention that Anubis [11] does not recognize the file format of an executable that uses GPU assembly code. Still, this failure point does not cause problems to virtualization/emulation techniques in principle. The later can adapt by adding new instruction sets to their static or dynamic analysis procedures. Heuristic methods proposed by M. Polychronakis, K. Anagnostou and E. Markatos [12] may be of help when dealing with this type of malware, since they do not focus on the malware code but on the malware effect on the system stack, calls etc. and do not employ a VM like the system call monitoring analysis of [8].

Last, we note that GPU assisted malware can only affect devices that use a GPU and specifically one that employees NVIDIA CUDA or ATI OpenCL. Still, we maintain that a sufficient amount of end users regularly access one of the later GPUs, specifically NVIDIA and ATI combined market share reached 99.6% (2011) in end-user discrete desktop graphics systems [25] and the General Purpose GPU computing capability is supported by the majority of modern GPUs. The extra effort required to produce a malware on both platforms (CUDA & OpenCL) is deemed justifiable, taking into consideration the existence of malware targeting even specific type of programming logic controllers (PCL) [13].

4 GPU Powered Antivirus

In the previous section, the discussion revolved around the various techniques and methods employed by GPU powered Malware, as well as the potential impact and limitations of these technologies. In the current section, we will focus on the capabilities and untapped potential of GPU Powered Antivirus systems. Most antiviral software components can be characterized as ‘computationally intensive’, since they employ usage of pattern matching algorithms on huge dataflows and under specific time constraints, commonly ‘live’ detection, protection and mitigation of threats. In this context, devices such as the GPU can be tapped to improve computational performance.

4.1 The potential of GPU Powered Detection

To achieve high performance (throughput in the range of 40Gbps to 4Gbps) and low latency, it is a common practice to use dedicated hardware in the form of application specific intergrated circuits. Moving to the 100Mbps-4Gbps scale, a combination of CPUs, network processing units and FPGAs are employed and only in the 100Mbps and lower scale, do we consider only an x86 CPU. The relatively low number of of dedicated hardware unit volumes and the additional R&D costs tend to increase the hardware price and thus, we are encouraged to search for alternative architectures [14].
We point out that network processing units and graphical processing units share a set of common characteristics: Both are designed to handle highly data-parallel algorithms (we refer to them as *extremely parallelizable* procedures), since both the network load/streams and graphics procedures can be divided to smaller sub-procedures that are handled by a large number of of multithreaded processors (another common feature of network and graphic processing units) [14, 1]. Last, as we mentioned, both are designed to handle large data flows and thus, to implement a fast interconnection bus between the processing units and the data [1].

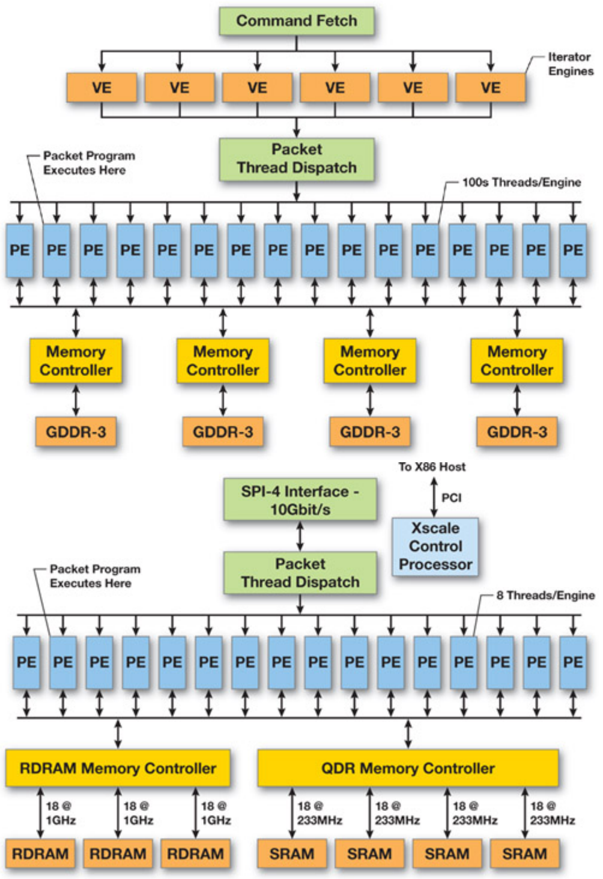

Figure 3: Top: A modern NVIDIA GPU. Bottom: Intel Network Processor IXP [14]

The applicability of GPUs to network scanning and malware detection has led to the development of antivirus detection engines and pattern matching software that have sustained throughput in the order of 40Gbps in the malware signature-matching version of the Aho-Corasick algorithm [15], around 24x performance increase of the Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search algorithms, compared to a CPU-only execution [16] and increased performance in intrusion detection systems [22].

4.2 Detection Algorithms and Performance

Several algorithms/methods have been proposed for signature matching in antivirus systems, including Bayer-Moore [19], Aho-Corasick [18], Rabin-Karp [26] and Bloom filters. We will focus on Bayer-Moore and Aho-Corasick implementations on GPUs and their limitations [15, 21], due to the fact that they are commonly used in antivirus software. Another reason is that the open-source Clam Antivirus employs both algorithms, using Bayer-Moore for simple signature-based matching (non-polymorphic viruses) and Aho-Corasick for regular expression pattern matching (polymorphic viruses) [15, 23], thus proving their applicability in our context.

Boyer-Moore. The Boyer-Moore string searching algorithm [19], performs preprocessing on the string being searched for (called pattern or 'needle') and not on the string being searched in (called text or 'haystack'). This makes it well suited for applications where the text does not persist across multiple searches, e.g. network data. The pattern preprocessing takes the form of the Bad Character rule and the Good Suffix rule which shift the pattern according to mismatched characters and suffixes. When implemented, the rules take the form of three lookup tables that are accessed in *O(1)* time and store the optimal jumps (i.e. pattern shifts) in the case of a mismatch. Overall, the algorithm has linear complexity of *O(n+m)*, n being the length of the pattern and m being the length of the text.

In a CUDA implementation, the lookup tables translate to a large number of memory lookups, so if the GPU *Global* memory is used to store the tables, the performance is decreased since GPUs have an high-bandwidth approach to tackle latency in their architecture instead of the multilevel cache hierarchy that we encounter in the traditional GPUs [21]. A suggested alternative is to store the lookup tables in the *Shared* memory of the graphics device (the equivalent of a CPU cache) in order to decrease memory-transfer latency [16] and increase performance but this may not always be feasible due to the large size of malware signatures.

Aho-Corasick. The Aho-Corasick string matching algorithm is a linear complexity algorithm that builds up a dictionary (i.e. all possible patterns and pattern prefixes) and a finite state machine in which all nodes consist of dictionary prefixes and the links between the state machine nodes represent successful and failed transitions in pattern matching [18, 20]. When implementing the Aho-Corasick algorithm in a GPU context, the preferred option is to construct the finite state machine traversal graph in the CPU and *then* transfer it to the GPU *Global* memory (instead of attempting to construct it inside the unsuitable GPU kernel) [21]. Since the DFA construction is a relatively rare operation (it occurs only when new entries are added to the signature database), performance is not impacted.

Researchers developed an engine to detect polymorphic malware signatures using the Aho-Corasick algorithm in GPUs [15]. The DFA is constructed on the CPU, using only a small prefix of each virus signature (further analysis on section *4.3*). The pattern matching is performed byte-wise (8 bits at a time), so the dictionary's alphabet has a size of *256* and thus, each state/node contains 256 pointers to other states. To utilize the parallel processing capabilities of the GPU, the input stream is split in to chunks and assigned to threads. To address the issue of pattern overlaps (a pattern spanning over multiple chunks and is being processed by independent threads) each thread continues to search up to the following chunk until either a pattern is found or the thread fails to detect any pattern at all. The heuristic is complete, since matching will continue in the undecided case, i.e. when the the results are ambiguous and there is even trivial chance of a malware being part of several network packets.

As in the Boyer-Moore case, the pattern matching is inhibited by the need to lookup the DFA (stored in global memory) multiple times. Memory can be optimized via usage of page-locked memory (non swappable) and by employing the *texture* memory of the graphics device, which can also act as GPU 'cache' memory, offer random and not coalesced access (which favors the DFA structure and access) and speed up the execution [15].

Direct application of graphic operations. Apart from GPGPU techniques to improve antivirus efficiency, it has been suggested that signature checking and memory analysis can be computed by using directly graphic textures and masks. Specifically, it was suggested that the data is bound on raw memory, that the signatures are bound to a masking texture atlas and then use a shader to sweep the mask over the data using texture operations [17].

4.3 The Live Operation Mode

Perhaps one of the most crucial system requirements (apart from detection success rate) in antivirus software is speed. Host-based antivirus systems need to react quickly and not inhibit the end user and network-based solutions need to handle huge dataflows in a small time frame. Static and dynamic detection techniques address this requirement and often they calculate the extra latency induced by execution and virtualization [10]. In this section we will address two important obstacles in the direction of high-speed traffic processing, in conjunction with the GPU architecture.

The Bottleneck issue. We encountered two categories of performance bottlenecks. The first type of bottleneck is a native performance issue of GPUs: their global memory is limited and the data they are able to process is upper-bounded by the global memory offered [1]. The common programming practice is to make subsequent kernel calls in the device, trying to allocate as much memory as possible in every call, due to the high penalty of data transfers between CPU host memory and GPU device memory.

The second case of bottleneck is more generic. Researchers that employed GPUs to perform 'live' operations such as live encryption/decryption for ssl/https purposes [27] have observed that network packet queuing and processing may be delayed even by the Linux kernel handling procedures. To tackle these issue and to demonstrate the construction of a large-scale GPU-assisted intrusion detection system, researchers have created specific architectures, manipulating the Linux kernel, its networking interfaces/queues and ensuring traffic load balancing [24].

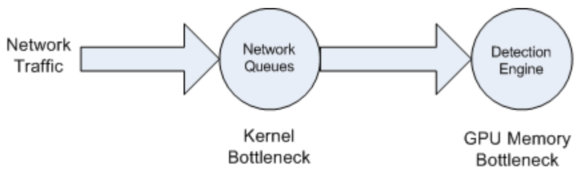

Figure 4: The two bottleneck cases, interconnected.

The False Positive - Performance tradeoff. Antivirus signatures are relatively large needles to search for. Constructing a DFA capable of *full*, 100% correct signature detection will require a big amount of memory and even more memory lookups, thus, rendering it hard to implement. The solution proposed by G. Vasiliadis and S.Ioannidis [15] is to use the GPU as *first filter* in detection. The GPU will try to rule out a large portion of the traffic on its own (exploiting its large parallel processing potential) and then transfer the remaining potential threats to the CPU for postprocessing. As mentioned in Section *4.2*, only a small prefix of the signature is used to create the DFA. The downside of this is an increased false positive rate since partial pattern matching contains errors. This is tackled by employing the CPU *after* the first filter to verify the alerts (which may either be real or false positives raised by the GPU filter) fast, since of its multi-level cache hierarchy that enables fast response time for a small set of alerts.

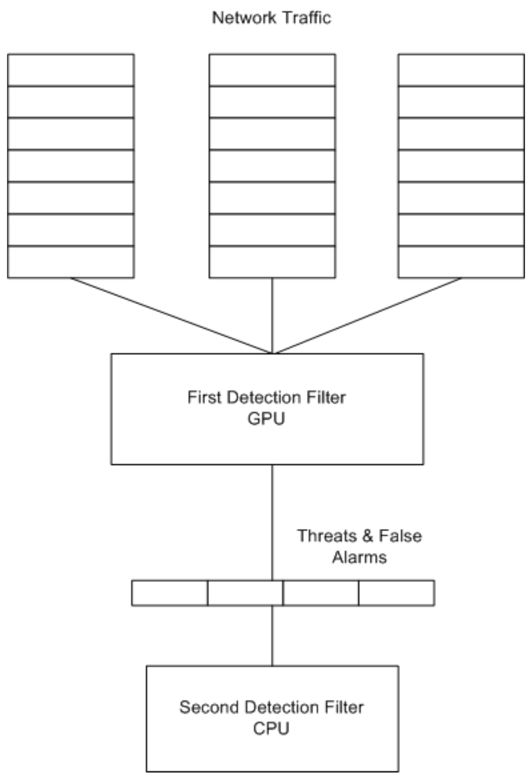

Figure 5: The two engine filters: The GPU operates on the huge dataflow and the CPU on the possible candidates of the first filter.

5 Conclusions

The constant evolution of obfuscation and detection techniques will be the prevailing situation in future issues and we believe that the GPUs are likely to influence both. The antivirus companies will need to quickly address the existing issues that have damaged the security services offered and will probably move to more dynamic, instead of traditional defense mechanisms. We do not expect that the old, signature-based defense model to eclipse, we rather believe that it will be integrated with dynamic elements and result in hybrid antivirus systems.

References

[1] NVIDIA CUDA C Programming Guide. Version 4.2. Date: 4/2012.

[2] T.Nash. *An Undirected Attack Against Critical Infrastructure. A Case Study for Improving Your Control System Security.* US-Cert Control Systems Security Center. 2005.

[3] Why Antivirus Companies Like Mine Failed to Catch Flame and Stuxnet. M.Hypponen. Wired. Date: 6/2012. URL: `http://www.wired.com/threatlevel/2012/06/internet-security-fail/`. Date Retrieved: 8/2012.

[4] P.Royal, M.Halpin, D.Dagon, R.Edmonds, W.Lee. *Polyunpack: Automating the hidden-code extraction of unpack-executing malware.* In Proceedings of the 22nd Annual Computer Security Applications Conference. 2006.

[5] G.Vasiliadis, M.Polychronakis, S.Ioannidis. *GPU-Assisted Malware.* PRACSE 2010.

[6] O.Harrison, J.Waldron. *Practical Symmetric Key Cryptography on Modern Graphics Hardware.* Computer Architecture Group, Trinity College Dublin. SS'08 Proceedings of the 17th conference on Security symposium, Pages 195-209. 2008.

[7] P.Ferrie. *Anti-unpacker Tricks.* Microsoft.

[8] C.Kolbitsch, P.M.Comparretti, C.Kruegel, E.Kirda, X.Zhou, X.Wang. *Effective and Efficient Malware Detection at the End Host.* SSYM'09 Proceedings of the 18th conference on USENIX security symposium, Pages 351-366. 2009.

[9] M.G.Kang, P.Poonsakam, H.Yin. *Renovo: A Hidden Code Extractor for Packed Executables.* WORM '07 Proceedings of the 2007 ACM workshop on Recurring malcode, Pages 46-53. 2007.

[10] D.Bolzoni, C.Schade, S.Etalle. *A Cuckoo's Egg in the Malware Nest. On-the-fly Signature Malware Analysis, Detection and Containment for Large Networks.* University of Twente, Eindhoven Technical University. BlackHat 2010.

[11] Anubis Malware Analyzing Engine. URL: `http://anubis.iseclab.org/` Date Retrieved: 8/2012.

[12] M.Polychronakis, K.G.Anagnostou, E.P.Markatos. *Comprehensive Shellcode Detection using Runtime Heuristics.* ACSAC '10 Proceedings of the 26th Annual Computer Security Applications Conference, Pages 287-296. 2010.

[13] Siemens: Stuxnet worm hit industrial systems. URL: `http://www.computerworld.com/s/article/print/9185419/Siemens_Stuxnet_worm_hit_industrial_systems?taxonomyName=Network+Security&taxonomyId=142`. Computer World. Date Retrieved: 8/2012.

[14] E.Seamans, T.Alexander. NVIDIA. *GPU Gems 3, Chapter 35, Fast Virus Signature Matching on the GPU.* URL: `http://http.developer.nvidia.com/GPUGems3/gpugems3_part01.html`. Date Retrieved: 8/2012

[15] G.Vasiliadis, S.Ioannidis. Intistitute of Computer Science, Foundation for Research and Technology - Hellas. *GrAVity: A Massively Parallel Antivirus Engine.* RAID'10 Proceedings of the 13th international conference on Recent advances in intrusion detection, Pages 79-96. 2010.

[16] C.S.Kouzinopoulos, K.G.Margaritis. Parallel and Distributed Processing Laboratory, Dep. of Applied Informatics, Univ. of Macedonia. *String Matching on a multicore GPU using CUDA.* PCI '09 Proceedings of the 2009 13th Panhellenic Conference on Informatics, Pages 14-18. 2009.

[17] J.Carlson. *GPUs Assisted Malware Detection for Mobile Devices.* Recon 2012.

[18] A.V.Aho, M.J.Corasick. *Efficient String Matching: an Aid to Bibliographic Search.* Communications of the ACM, 18(6):333-340. 1975.

[19] R.Boyer, S.Moore. *A Fast String Searching Algorithm.* Communications of the ACM, 20 (10): 762772. 1977.

[20] GNU Documentation. *Chapter 13. The DFA pattern matcher.* URL: `http://www.delorie.com/gnu/docs/gnugo/gnugo_160.html`. Date Retrieved: 8/2012.

[21] A.Gee. *Research into GPU accelerated pattern matching for applications in computer security.* Dep. of Computer Science and Software Engineering, Univ. of Canterbury NZ. 2009.

[22] G.Vasiliadis, S.Antonatos, M.Polychronakis, E.Markatos, S.Ioannidis. *Gnort: High Performance Network Intrusion Detection Using Graphics Processors.* Institute of Computer Science, Foundation for Research and Technology Hellas. Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection. Springer, Lecture Notes in Computer Science, 2008, Volume 5230, Recent Advances in Intrusion Detection, Pages 116-134.

[23] Clam Open-Source Antivirus. URL: `http://www.clamav.net/lang/en/`. Date Retrieved: 8/2012.

[24] G.Vasiliadis, M.Polychronakis, S.Ioannidis. *MIDea: A Multi-Parallel Intrusion Detection Architecture.* Proceedings of the 18th ACM conference on Computer and communications security Pages 297-308.

[25] D.Perry. *Nvidia Graphics Chip Market Share Nosedives.* Tom's Hardware US. URL: `http://www.tomshardware.com/news/grpahics-chip-gpu-geforce,12698.html`. Date Retrieved: 8/2012.

[26] R.Karp, M.Rabin. *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development - Mathematics and computing, Volume 31 Issue 2, Pages 249 - 260. 1987

[27] K.Jang, S.Han, S.Han, S.Moon, K.Park. *Accelerating SSL with GPUs.* Dep. of Electrical Engineering, KAIST. Proceedings of the ACM SIGCOMM 2010 conference, Pages 437-438.