CPU/GPU Hybrid Detection for Malware Signatures

Radu Velea
Department of Computer Engineering
Technical Military Academy of Bucharest, Romania
Bitdefender, Romania
radu.velea@mta.ro, rvelea@bitdefender.com

Ștefan Drăgan
Bitdefender, Romania
sdragan@bitdefender.com

Abstract—**Malware detection is an important aspect of cyber security. The process of identifying malicious code in files or network traffic is very complex and requires a lot of computational resources. Most security solutions that deal with malware detection implement advanced string matching algorithms or look for certain behavioral patterns during program execution. These methods of detection can cause significant performance penalties for real-time applications, can limit the scan surface or degrade user experience. In this paper we discuss a hybrid approach that leverages CPU and GPU compute capabilities in order to accelerate pattern matching for malware signatures. The solution presented focuses on improving performance and reducing power consumption of string matching algorithms on devices such as ultrabooks and laptops.**

Keywords—String matching, Parallel algorithm, Malware, GPGPU, Virus, OpenCL

I. INTRODUCTION

Malware is the designated term for any malicious software that disrupts the normal workflow of a computer application or system. Malware can take the form of executable code that passes itself as a legitimate in order to compromise a system. A compromised system may become vulnerable to additional cyber-attacks and this in turn can lead to information loss or theft, denial of service and other undesired consequences. Malware scope can range from single users to organizations or public infrastructure.

The definition of the term is broad and it can be used to refer to any kind of software that has a negative impact on user experience or damages assets. Common types of malware include ransomware, viruses, spyware or Trojan horses. Versions of these entities can be found in just about every known form factor: from embedded devices to super computers. A system is usually contaminated through an infected file. The infected file can take advantage of a vulnerability inside a legitimate application and execute malicious code. Infections can come via hard drives, USB sticks, and optical storage devices or from the network. In the age of the Internet, where most devices are interconnected, unchecked infections can spread rapidly. To mitigate this risk, multi-layered defense systems have been implemented to protect endpoints and networks. Typical applications that combat malware consist of antivirus software, firewalls, network intrusion detection and prevention systems (NIDPS).

In the current environment, the attack surface for malicious applications is very large. Lack of transparency and reaction from vendors can result in security incidents that go unnoticed for weeks of even months. Detection and prevention tools usually rely on matching suspected files against a database of known threats. The security of a system can greatly depend on how often its signature database is updated with the latest discovered threats. While organizations tend to have a process in place for protection against attacks, the average user has to rely on his or hers provider. Lack of technical knowledge causes users to fall prey to viruses that have been known to exist for years or decades. Even if their service provider supplies regular security updates, users may be unwilling to adopt them due to performance considerations (fear it may slow down their device, take too much disk space, bandwidth, etc.) or sheer ignorance [1].

The context described above drives security companies to develop performance-efficient solutions to cope with the ever-growing amount of malicious content their customers are exposed to. In the following sections we will describe what makes malware detection troublesome from a performance point of view and underline the hotspots anti-malware tools have. We will then propose a new method to speed up detection by combining the compute capabilities found in consumer desktop systems and laptops.

Detection of malicious code can be done statically or at runtime. Static analysis requires a set of pattern matching operations that have to determine if a blob of data resembles any known malware. Researchers try to populate databases with malware signatures that will then be used to scan files. To counter this, malware writers go to extreme lengths to obfuscate their code and bypass any known filters. A practical way to detect a malware instance is to generate a footprint by performing semantic rather than syntactic analysis of the code [2]. Another frequent challenge is dealing with zero-day vulnerabilities and self-mutating malware. These kind of attacks can be mitigated through runtime analysis (executing the code in a contained environment and observing its behavior) or the implementation of machine learning algorithms. Applying such techniques in real-time can take up a significant amount of resources and generate false positives. An option would be to perform these investigations in a controlled environment and then generate static signatures that can detect the new threat and its derivatives in the wild.

Once the signature has been generated, it can be used by compatible tools to detect that type of malware. Detection could happen on a variety of devices: embedded systems, mobile phones, desktops, cloud infrastructure, etc. For example a vulnerability in a web browser could affect all systems that access the Internet through it. Protection systems are thus deployed in different forms according to the device's available

978-1-5386-2752-5/17/$31.00 ©2017 IEEE

resources and its security needs. A network sensor that runs a NID&PS would be responsible for deep packet inspection. Studies have shown that a common performance bottleneck during this process is the necessity to perform string matching [3] [4]. The overhead of pattern matching can cause degradation of network performance, while relaxing the rules could allow threats to go undetected. A host-oriented software such as an antivirus is required to perform regular scans in order to ensure the integrity of the system. If these regular scans take too long or strain the machine's resources, the user might opt to perform them at longer intervals or skip them altogether.

It is therefore important that the patterns-matching process required for malware detection be performed in an efficient manner and that it takes advantage of all the computing resources available on the host device.

II. Related Work

A. Detection through String Matching

Static malware detection comes down to string searches - finding known blocks of malicious code inside data on the system or network. String matching algorithms based on Aho-Corasick [5] and Boyer-Moore [6] have been adapted for this task. They are used by commercial software as well as open source projects like Snort, Suricata or ClamAV. The logic behind these implementations is to perform the minimum number of byte comparisons on the smallest set of data possible without compromising the accuracy of the search. The theoretical details behind these algorithms are out of the scope of this work.

Hashing techniques can accelerate pattern matching algorithms and can potentially detect viruses encrypted with simple functions such as ADD and XOR [7].

B. Parallel Implementations

A survey estimated that as much as 75% of CPU time is spent performing pattern matching in NIDPSs [8]. High traffic throughput has motivated researchers to look for alternative ways to offload scan tasks that would normally run on the CPU. The GPU is an ideal candidate for this assignment because of its SIMD architecture and high level of parallelism. Experiments with Snort [9] have concluded that GPU string matching is efficient, but that performance can deteriorate if memory transfers are not handled accordingly. This can make real-time detection problematic if the GPU is used to scan packets that are few and far between or small, individual files on the disk.

Changes to the algorithms that run on the GPU focus on optimizing memory accesses [10] and exploiting the large number of available compute units [11]. Favored approaches include the compression [12] of the state machine for automatons and removing the failed transactions [13] (the current thread will exit after a character mismatch rather than try to continue from another valid state). For algorithms based on lookup tables and optimization would be to use hashed prefixes [14] to skip as many characters as possible from the benign input.

Some works have proposed hybrid implementations that use OpenMP together with CUDA to perform string matching [15] [16]. Memory limitations negatively impacted the number of signatures that could be searched in the string, but the solutions provided significant speedups over the serial versions. To solve some of the drawbacks caused by transferring data back and forth between the GPU and CPU developers have looked for alternative solutions that can perform opportunistic load balancing [17] or shallow searches. GPU hardware vendors have advertised new designs that promise to solve this problem by providing a unified memory model [18]. Some of the devices available on the market that share memory between CPU and GPU are mobile phones and other small form factors (ultrabooks, laptops, chromebooks) with integrated graphics.

Software frameworks used for GPU programming are Compute Unified Device Architecture (CUDA) and OpenCL. CUDA is the older and more popular technology. It is designed to run on NVidia hardware and besides graphics, it is used for high performance computing in physics, medical imaging, distributed computing and other GPGPU related work. OpenCL is an open standard maintained by a consortium of hardware and software vendors. It is designed to run on a greater variety of hardware and has both proprietary and open source implementations. OpenCL is a flexible API and can run on multicore CPUs and better map itself to low-end platforms. CUDA and OpenCL have similar memory and programming models. The solution described in this paper has been implemented in OpenCL. The choice of OpenCL over CUDA is motivated by the fact that OpenCL's role is to enable parallel programs to run across heterogeneous hardware and, as a result, is more widely available among users (NVidia graphics can run OpenCL applications).

III. Implementation

A. Efficient Pattern Matching

The current section describes our proof of concept for parallelizing malware detection across heterogeneous hardware. Our string matching implementation is based on a variation of Boyer–Moore–Horspool [19] algorithm. The algorithm was designed to search for malware signatures inside files located on the drive. We make the assumption that for the most part our searches will not result in any detection, regardless of the signature database size or file system. This detail will be used to provide an additional speedup during the scanning phase.

The static fingerprint of a piece of malware is defined as a set of instruction blocks that make it stand out from other conventional pieces of software. Malware signatures are available online and are updated regularly when new infections are discovered. These instruction blocks are the patterns we have to identify amongst the scanned content. Before building the lookup table we load all the signatures into memory and sort them using the first 32 bytes of their binary code as key. The resulting sorted structure will be used later to search for exact matches.

In the preprocessing stage the lookup table is built with integer (4-byte) values rather than individual bytes. The integer

values represent a *hash* of the last key bytes. Offsets or skip distances between input bytes are computed via hash equality and not byte equality. The hash function is not injective and occasional collisions will occur between signatures. This means that once a match is detected there is a chance it could be a false positive (a byte sequence that just happens to have the same hash as a malicious pattern). In order to mitigate this, the algorithm performs a binary search into the sorted key structure and checks for an exact match. This process is compute intensive but it is only expected to be executed in exceptional cases. If any malware signatures are detected, they will be reported to the upper levels of the application and dealt with accordingly.

B. OpenCL Parallelization

The scan process involves parsing a large amount of input data, one byte at a time and identifying possible matches, as described above. Our OpenCL parallelization efforts focused on offloading this part of the code to the GPU. After the lookup-search is done in parallel, the results are transferred to the CPU for the binary search to complete the match and take necessary actions. This process is done one file at a time. Some of the related work presented [14] in the previous section suggests concatenating a significant amount of data before sending it to the GPU in order to minimize the penalty incurred from frequent memory transfers. While this approach may seem sound, it is not practical for most real-life scenarios. A user may decide to incrementally scan his hard drive, a few files at a time, or may simply not possess the available resources to load large amounts of data (GBs) into memory. Files most susceptible to infection are generally small in size [20]. For this reason, the focus of our implementation is to achieve equal or better speedups of the smallest amount of data.

The most straightforward approach is to split the amount of data evenly across all available GPU threads. Each thread would receive a chunk of bytes and will have to report which of them are valid offsets for future analysis. To reduce the amount of computation on the CPU we first attempted to compute the *candidate* key for the next-stage binary search. Experimental results showed the penalty for repeatedly accessing GPU global memory to be significant. To reduce the number of memory accesses we also reduced the amount of computation and only outputted a corresponding bit value for each processed byte. Input data would be padded to 64 bytes and each GPU work item would be responsible to compute the output for a fixed chunk of 64 bytes. The output bits would be added to a 64-bit *ulong* mask and copied back to CPU memory. Each thread would only have to access global memory that contained input bytes and lookup tables (which would only be transferred to the GPU once – after the signature preprocessing stage). The CPU would then iterate through the bitmasks received from the GPU and process any non-zero values. Boyer-Moore table lookups would ensure that each GPU thread will actually process less than 64 bytes, as most of them are expected to be skipped. Further parallelization can be done on the CPU-side by using multithread libraries such as Pthread. Each pthread would have a corresponding OpenCL context and handle its own content. This would allow for multiple files to be scanned in parallel, but would duplicate the amount of memory required on the GPU-side, as lookup tables would not be shareable across contexts. The current work presents a solution that scans one file at a time and uses on a single CPU thread with a single OpenCL context in which multiple work items are executed.

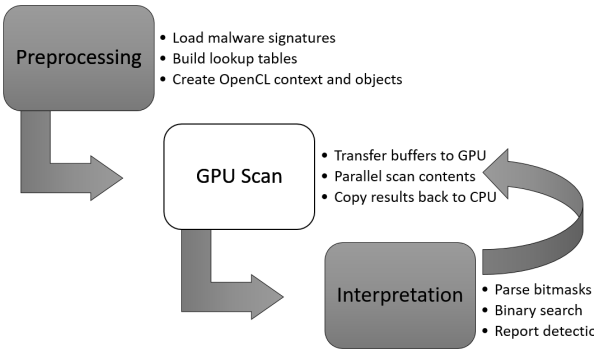

Figure 1. Malware scan workflow.

IV. RESULTS

A. Experimental Setup

The implementation was tested on a 64-bit ultrabook with Intel® CoreTMi7-6600U with integrated HD Graphics 520. This form factor can run Windows operating systems as well as Linux-based distributions such as Ubuntu, ChromeOS or Android. The integrated GPU has 24 compute units clocked at 1050 MHz. It does not have dedicated graphics memory, but instead use a part of the main memory. Level 3 cache is shared between CPU and GPU. This setup is ideal for testing the performance of our hybrid detection framework. Similar devices are available on the market and used for business or leisure.

The malware database used for the tests contained approximately 20000 signatures. The test files consisted of Windows system files, Linux root file system, and randomly generated input, along with some selected malware samples. In case a file was too large a fixed-sized buffer was created in order to scan only X amount of data at a time.

B. Performance

Performance tests where categorized into two groups. The first group involved repeated scans using variable buffer sizes. This test will determine the speedup between the hybrid implementation and the CPU-only one. The sizes of the scanned files will range from a couple of bytes to several GBs in size. This method of evaluation will help us find the ideal buffer size that provides the best performance on our device.

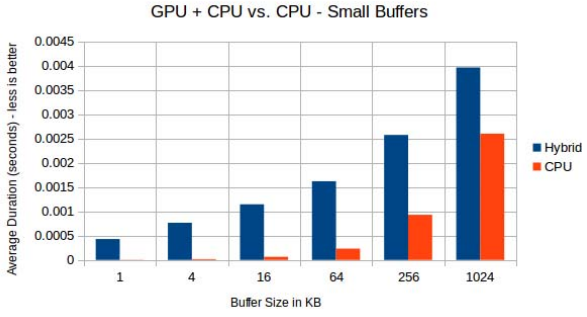

Figure 2. Hybrid pattern matching performance on small buffers.

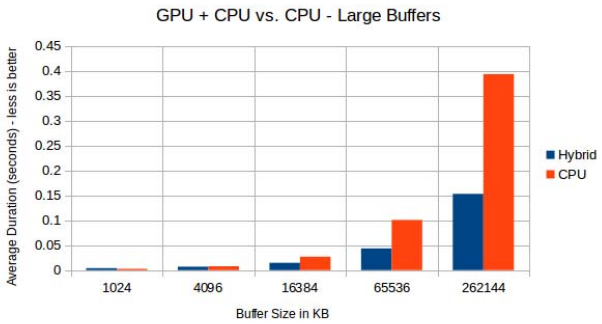

Figure 3. Hybrid pattern matching performance on large buffers.

Fig. 2 shows OpenCL is not very efficient in scanning small buffers. Upon closer examination it was found that for a 1 KB buffer only 2% of computation time was spent on the GPU (either executing code or performing memory transfers). The rest of the time (about 0.3 milliseconds) was spent in OpenCL library calls: sending commands to the execution queue, scheduling, waiting for other events, etc. To reduce part of this penalty memory transfers were performed by mapping GPU buffers into host address space and performing read and write (memcpy) operations on the CPU. As the size of the scanned buffer grows, the hybrid performance improves compared with the CPU-only. The point where hybrid performance surpasses the CPU is around a 4 MB buffer (the test machine has a 4 MB L3 SmartCache [21]). In fig. 3 (as the buffer grows), GPU-time reaches around 96% of the total hybrid computation and speedup values increase from 1.12x to 2.57x in favor of the GPU+CPU solution.

The second group of tests focused on power consumption and overall efficiency. Tools like GPU-Z[1] and Intel® Power Gadget[2] were used to measure the power consumption and other metrics while scanning.

We set the buffer size to 16 MB and measured the power consumption of the CPU and GPU:

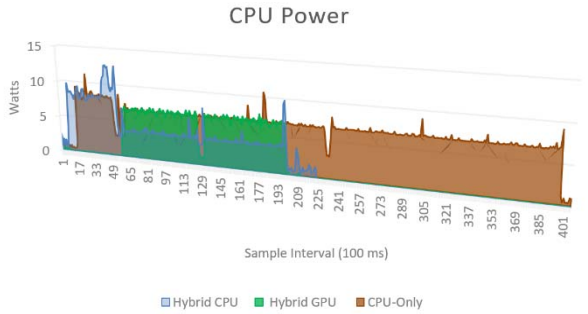

Figure 4. Comparison of power consumption between hybrid and CPU-only

[1] https://www.techpowerup.com/gpuz/
[2] https://software.intel.com/en-us/articles/intel-power-gadget-20

The hybrid implementation is almost twice as fast and consumes 25% less power while running the benchmark:

TABLE I. TEST SUMMARY

	Hybrid	**CPU-Only**
Duration (s)	22.4	40.7
Avg. CPU Power (W)	4.68	6.5
Avg. GPU Power (W)	4.23	0.1
Total Energy (W * s)	199.58	268.62

The test was performed while the device's power plan was set to high performance: CPU frequency was 3200 MHz for the duration of the test.

V. CONCLUSION

The results presented in fig. 4 suggest the hybrid solution offers a significant advantage in power and performance over a CPU-only implementation. However, these advantages disappear if the application has to scan small files (less than 1 MB in size). Based on these experimental results we could introduce a logic inside the application to take different code paths according to the amount of data available. For the selected test platform, the impact on battery power seems to favor this hybrid approach. If the application has to handle large amounts of data a bigger internal GPU scan buffer would provide incremental benefits.

Our conclusion is that efficient usage of an integrated GPU can speed-up string matching operations for antimalware software on battery based devices. The unified memory model provided by the test hardware reduced the overhead incurred by repeated memory transfers between CPU and GPU, but also created a penalty for multiple accesses of GPU global memory inside the kernel code and made usage of local memory impractical. OpenCL provides a versatile framework for offloading intensive computation performed in network and host intrusion detection systems in environments that have, otherwise, limited resources. Future work will include deployment and measurement on platforms that have dedicated GPUs and further comparisons with CUDA-based implementations and other string matching algorithms.

REFERENCES

[1] L. Zhang-Kennedy, S. Chiasson, and R. Biddle, "Stop clicking on "update later": Persuading users they need up-to-date antivirus protection," in International Conference on Persuasive Technology, pp. 302–322, Springer, 2014.

[2] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in Security and Privacy, 2005 IEEE Symposium on, pp. 32–46, IEEE, 2005.

[3] K. Salah and A. Kahtani, "Performance evaluation comparison of Snort NIDS under Linux and Windows Server," Journal of Network and Computer Applications, vol. 33, no. 1, pp. 6–15, 2010.

[4] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, and T.-H. Lee, "Using string matching for deep packet inspection," Computer, vol. 41, no. 4, 2008.

[5] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," Communications of the ACM, vol. 18, no. 6, pp. 333–340, 1975.

[6] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no. 10, pp. 762–772, 1977.

[7] M. Ciubotariu, “Virus Cryptoanalysis,” Virus Bulletin, 2003.

[8] S. Potluri and C. Diedrich, “High Performance Intrusion Detection and Prevention Systems: A Survey,” in ECCWS2016-Proceedings fo the 15th European Conference on Cyber Warfare and Security, p. 260, Academic Conferences and publishing limited, 2016.

[9] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, “Gnort: High performance network intrusion detection using graphics processors,” in International Workshop on Recent Advances in Intrusion Detection, pp. 116–134, Springer, 2008.

[10] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, “Accelerating pattern matching using a novel parallel algorithm on GPUs,” IEEE Transactions on Computers, vol. 62, no. 10, pp. 1906–1916, 2013.

[11] A. Tumeo, O. Villa, and D. Sciuto, “Efficient pattern matching on GPUs for intrusion detection systems,” in Proceedings of the 7th ACM international conference on Computing frontiers, pp. 87–88, ACM, 2010.

[12] C. Pungila and V. Negru, “A highly-efficient memorycompression approach for GPU-accelerated virus signature matching,” in International Conference on Information Security, pp. 354–369, Springer, 2012.

[13] D. Thambawita, R. Ragel, and D. Elkaduwe, “To use or not to use: Graphics processing units (GPUs) for pattern matching algorithms,” in Information and Automation for Sustainability (ICIAfS), 2014 7th International Conference on, pp. 1–4, IEEE, 2014.

[14] G. Vasiliadis and S. Ioannidis, “Gravity: a massively parallel antivirus engine,” in International Workshop on Recent Advances in Intrusion Detection, pp. 79–96, Springer, 2010.

[15] S. Ashkiani, N. Amenta, and J. D. Owens, “Parallel Approaches to the String Matching Problem on the GPU,” in Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 275–285, ACM, 2016.

[16] H. A. Kadhim and N. A. Rashid, “Parallel GPU-Based Hybrid String Matching Algorithm,” in Advanced Computer and Communication Engineering Technology, pp. 1199–1208, Springer, 2016.

[17] Y.-S. Lin, C.-L. Lee, and Y.-C. Chen, “A Capability-Based Hybrid CPU/GPU Pattern Matching Algorithm for Deep Packet Inspection,” International Journal of Computer and Communication Engineering, vol. 5, no. 5, p. 321, 2016.

[18] P. Rogers and A. Fellow, “Heterogeneous system architecture overview,” in Hot Chips, vol. 25, 2013.

[19] R. N. Horspool, “Practical fast searching in strings,” Software: Practice and Experience, vol. 10, no. 6, pp. 501–506, 1980.

[20] R. Poston, “How large is a piece of Malware?.” https://nakedsecurity.sophos.com/2010/07/27/large-piecemalware/. Accessed: 2017-02-20.

[21] T. Tian and C.-P. Shih, “Software techniques for shared-cache multi-core systems,” Intel Software Network, 2007.