CPU/GPU Hybrid Detection for Malware Signatures for Battery-Powered Devices Using OpenCL

Radu Velea, Ştefan Drăgan, and Florina Gurzău

1 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 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 other types of computer viruses. Versions of these entities can be found in just about every known form

---

This is an extended version of a preliminary conference paper that was accepted and presented in ICCA 2017 [1].

R. Velea (✉)
Department of Computer Engineering, Technical Military Academy of Bucharest, Bucharest, Romania

Bitdefender, Bucharest, Romania
e-mail: rvelea@bitdefender.com; radu.velea@mta.ro

Ş. Drăgan
Bitdefender, Bucharest, Romania
e-mail: sdragan@bitdefender.com

F. Gurzău
Department of Computer Engineering, Technical Military Academy of Bucharest, Bucharest, Romania
e-mail: florina.gurzau@mta.ro

© Springer International Publishing AG, part of Springer Nature 2018

J. M. Alja'am et al. (eds.), *Recent Trends in Computer Applications*,
https://doi.org/10.1007/978-3-319-89914-5_9

factor: from embedded devices to supercomputers. 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 (NIDPSs).

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 or 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 her 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 [2].

Customer and enterprise security solutions are required to process ever-increasing amounts of data. Normal workloads include scanning files on the disk, examining process memory, validating user input or filtering network traffic. The growing complexity of new cyber-threats means that detection and protection tasks need to consume more resources in order to be efficient.

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 antimalware 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 [3]. 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 through 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 resources and its security needs. A network sensor that runs an NIDPS 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 [4, 5]. 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.

2 Related Work

2.1 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 [6] and Boyer-Moore [7] 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 [8].

Efficient hash functions that provide few collisions over a set of characters could be used instead of conventional string matching operations. This approximate solution to detect threats can be useful in scenarios where the set of input characters is reduced—for example, when scanning scripts, markup language or human-readable text.

2.2 Parallel Implementations

A survey performed by [9] estimated that as much as 75% of CPU time is spent performing pattern matching in NIDPSs. 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 [10] 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 [11] and exploiting the large number of available compute units [12]. Favored approaches include the compression [13] of the state machine for automatons and removing the failed transactions [14] (the current thread will exit after a character mismatch rather than try to continue from another valid state). For algorithms based on lookup tables, an optimization would be to use hashed prefixes [15] 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 [16, 17]. 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 [18] or shallow searches. GPU hardware vendors have advertised new designs that promise to solve this problem by providing a unified memory model [19]. 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).

The current paper explores the opportunity of using the GPU to offload compute-intensive tasks that usually take a lot of the CPU time. The case studies described in

this work target battery-powered devices, such as laptops or ultrabooks, that implement some form of security software. The goal is to use the extra computing power of the GPU to improve execution times and reduce overall power consumption for system scans or other antimalware processes.

3 Implementation

3.1 Exact 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 [20] 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 among 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.

3.2 Approximate Pattern Matching

The method described above has some limitations: it requires the scan engine to load the signatures in their fullest form in order to perform an exact match. For simple, 32-bit hashes used for malware signature lookup we can expect to have

regular collisions that have to be solved down the line by performing byte-to-byte comparisons. Given the nature of the algorithm we can expect to have a match for one out of one million bytes, regardless of the nature of the input. This means that we will have to perform an extra check for roughly 1 MB of scanned content. While this does not influence the scan speed significantly, it does impact memory consumption, as the large signature database has to be carried around for these extra checks. This makes deployment problematic for low-end devices because it ties down an amount of memory proportional to the size of the signature database.

To solve this problem, the scan engine would have to fully rely on hashes for matching incoming traffic against the malware database. This procedure is not 100% accurate, but can provide a solution that can satisfy the current needs within a reasonable degree of certainty. In this case, the scan engine would no longer perform byte-level comparisons to determine if a match is a false positive or not, instead relying on a combination of hashes and to determine the final result. Given the a token size of 256 bytes for a malware signature, we can look for hash functions that have a good spread and are easy to compute.

A solution would be to use the same hash function at different offsets. The malware signature would be preprocessed and the hash values would be stored in the process's memory instead of the actual contents. This method would allow our algorithm to reuse the latter computation at different stages of the detection process and would significantly reduce the memory footprint.

$$\begin{aligned} f(0) &= hash\,(offset\;0, 255) \\ f(1) &= hash\,(offset\;1, 256) \end{aligned} \quad (1)$$

If the computed values for $f(0)$ and $f(1)$ match the ones loaded for a particular signature, we can safely claim that we have match. To further reduce the possibility of having a false positive, multiple levels of the function f can be used. The mathematical chance that a signature matches benign content based on this mechanism still remains, but a careful analysis of the hash function and the type of input it is used on (e.g., human readable text, scripts and binary code) can be used to reduce it to almost 0. A further advantage in speed for serial implementations can be achieved if the $f(n+1)$ depends on $f(n)$. This dependency can help the algorithm's speed, but in turn adds a constraint that can hinder the parallelization process.

An interesting candidate for a fast hash function has been identified in CLHASH [21]. CLHASH uses the carry-less multiplication instruction CLMUL, available on x86 architectures to compute a non-cryptographic hash. The structure of the algorithm is based on performing multiple carry-less multiplications on chunks of bytes obtained by XOR-ing the input data with a randomly generated key, and then accumulating the result. The mechanism is simple enough to allow modifications—we will use some of the elements present in CLHASH to create a candidate-function that could be used on both the CPU and GPU. The original code of CLHASH was based on the *pclmulqdq* instruction from the expanded instruction set of x86-64 architectures. This instruction performs a carry-less multiplication on two 64-bit

integer values and stores the result in a 128-bit register. This instruction provides significant speedup for the function on the CPU side.

3.3 *OpenCL Parallelization for Exact Pattern Matching*

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 [15] 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 [22]. For this reason, the focus of our implementation is to achieve equal or better speedups when using small amounts 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 unsigned long mask and copied back to CPU memory. Each thread would only have to access the 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 nonzero 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 (Fig. 1).

The function responsible for preprocessing or filtering the input buffer should be chosen carefully. Some types of operations are known to cause significant performance penalties when used on the GPU (e.g., branching instructions).

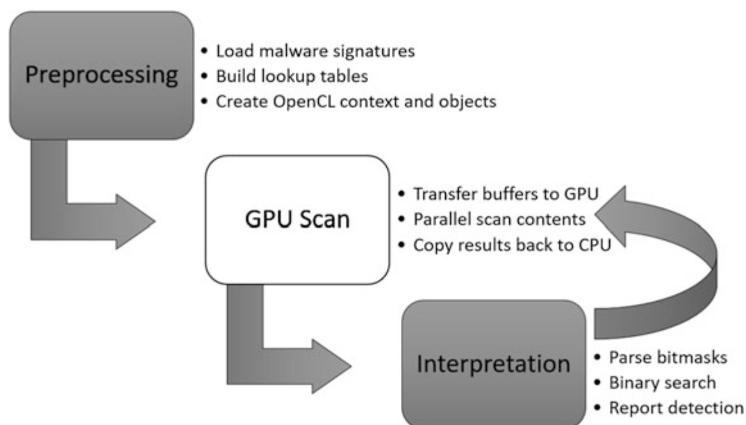

Fig. 1 Malware scan workflow

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.

4 Results

4.1 Test 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 and ChromeOS, or Android. The integrated GPU has 24 compute units clocked at 1050 MHz. The machine does not have dedicated graphics memory, but instead uses 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 20,000 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.

4.2 Performance

Performance tests were 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 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 ms) 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 [23]). 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.

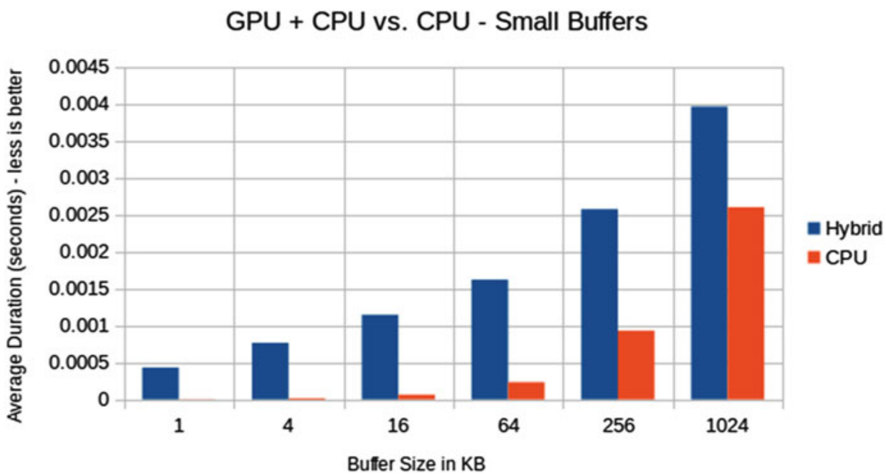

Fig. 2 Hybrid pattern matching performance on small buffers

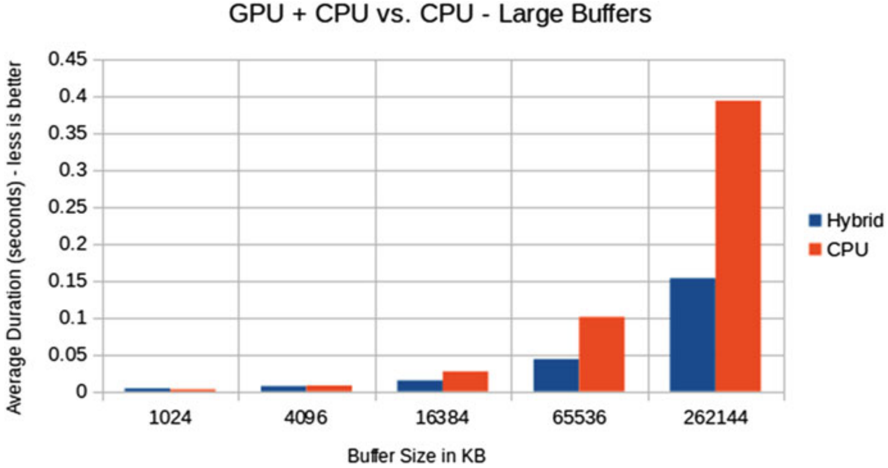

Fig. 3 Hybrid pattern matching performance on large buffers

Table 1 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 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:

The hybrid implementation is almost twice as fast and consumes 25% less power while running the benchmark (Table 1):

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.

4.3 Experiments with Other Hash Functions

To create a more complex hash function, suitable for both the GPU kernel and the CPU, we experimented with some elements from CLHASH: we selected a scenario in which the scanned input consists of a stream of human-readable text and we generated hashes from the malware database to be used instead of byte-per-byte comparisons. For each 256-byte signature we would create two 64-bit hashes at

[1]https://www.techpowerup.com/gpuz/

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

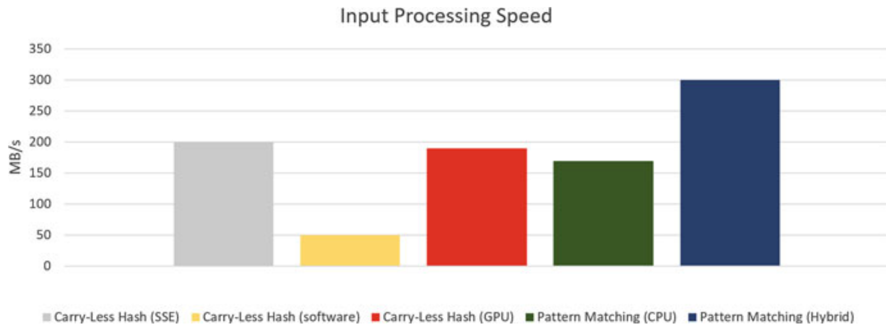

Fig. 4 Comparison of power consumption between hybrid and CPU-only

64-byte intervals and store them sorted in the memory. We would then process the input text and perform 64-bit lookups in order to determine if we have a match or not. CLMUL instructions are not available from OpenCL on the GPU. To mitigate this disadvantage, we created software versions of the hash functions and compared the results with the previous detection mechanism and the SSE[3] version of the hash function.

Figure 4 shows that in spite of the parallelization efforts, the carry-less hash function performs poorly without dedicated hardware support. Nevertheless, the performance is still a good 10% better than the best CPU-only pattern matching scheme. With future dedicated instructions that can perform carry-less multiplication on the GPU, there is the potential of achieving better results in terms of speed.

From a memory point of view, this scheme brings a significant reduction in the size of the memory used by the scan engine. With the original pattern matching framework, the process would have to store 256 bytes plus the size of the lookup table key for each of the malware signatures. Using an approach based exclusively on hashes can bring down memory by almost 75%.

5 Conclusion

The results presented in Fig. 5 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.

[3]Streaming SIMD Extensions.

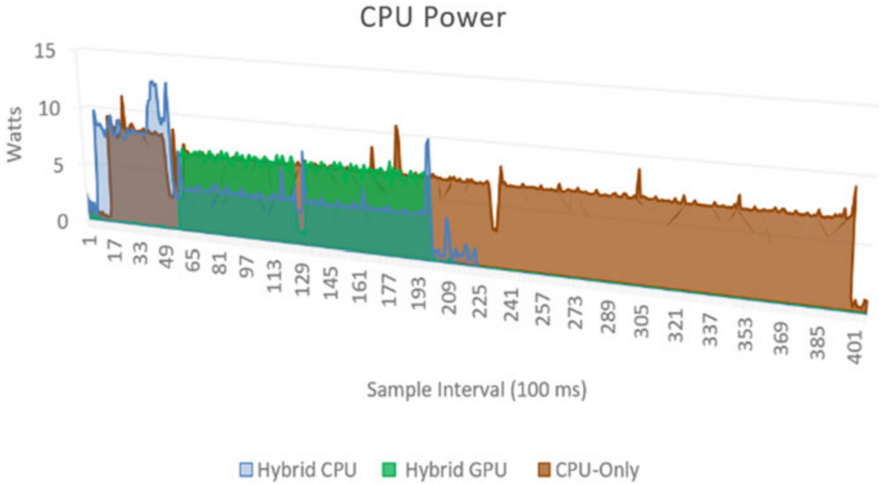

Fig. 5 Speed processing speed for hash-based implementations

Contextual scenarios can benefit more from frameworks that use a combination of hash values rather than classical pattern matching. The parallelization of such frameworks might not yield significant performance improvements compared to an optimized CPU-only version, but there is a promise for future improvement as hardware and software evolve.

The current proof of concept represents a step closer toward the creation of new security solutions that harness all the existing computing resources available on a machine. The concept of "security through total computing" looks to promote the implementation of heterogeneous software products that enhance the level of security of the average user, without compromising user experience.

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. R. Velea and Ş. Drăgan, "CPU/GPU Hybrid Detection for Malware Signatures," in *Computer and Applications (ICCA), 2017 International Conference on*, Dubai, 2017.
2. Zhang-Kennedy, S. C. Leah and R. Biddle, "Stop clicking on "update later": Persuading users they need up-to-date antivirus protection," in *International Conference on Persuasive Technology*, 2014.
3. 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*, 2005.
4. 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.
5. 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.
6. 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.
7. R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM,* vol. 20, no. 10, pp. 762–772, 1977.
8. M. Ciubotariu, "Virus Cryptoanalysis," *Virus Bulletin,* 2003.
9. 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*, 2016.
10. G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*, Berlin/Heidelberg, Springer, 2008, pp. 116–134.
11. C.-H. Lin, C.-H. Liu, L.-S. Chien and a. 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.
12. 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*, 2010.
13. Pungila and V. Negru, "A highly-efficient memory compression approach for GPU-accelerated virus signature matching," in *International Conference on Information Security*, 2012.
14. D. R. V. L. B. 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*, Colombo, Sri Lanka, 2014.
15. G. Vasiliadis and S. Ioannidis, "Gravity: a massively parallel antivirus engine," *International Workshop on Recent Advances in Intrusion Detection,* vol. 63, no. 7, pp. 79–96, 2010.
16. S. Ashkiani, N. Amenta and J. D. Owens, "Parallel Approaches to the String Matching Problem on the GPU," *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures,* pp. 275–285, 2016.
17. H. A. Kadhim and N. A. Rashid, "Parallel GPU-Based Hybrid String Matching Algorithm," *Advanced Computer and Communication Engineering Technology,* pp. 1199–1208, 2016.
18. 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, pp. 321–330, 2016.
19. P. Rogers, "Heterogeneous system architecture overview," in *Hot Chips 25 Symposium (HCS), 2013 IEEE*, Stanford, CA, USA, 2013.
20. R. N. Horspool, "Practical fast searching in strings," *Software: Practice and Experience,* vol. 10, no. 6, pp. 501–506, 1980.
21. D. Lemire and O. Kaser, "Faster 64-bit universal hashing using carry-less multiplications," *Journal of Cryptographic Engineering,* vol. 6, no. 3, pp. 171–185, 2016.

22. R. Poston, "How large is a piece of Malware?," 27 July 2010. [Online]. Available: https://nakedsecurity.sophos.com/2010/07/27/large-piece-malware/. [Accessed 20 December 2017].
23. T. Tian and C.-P. Shih, *Software techniques for shared-cache multi-core systems,* Intel Software Network, 2007.