GPU Encrypt: AES Encryption on Mobile Devices

James Gleeson, Sreekumar Rajan, Vandana Saini
Department of Computer Science
University of Toronto
Toronto, ON
{james.gleeson,sreekumar.rajan,vsaini}@cs.toronto.edu

I. Introduction

Cell phones have evolved into a ubiquitous and essential method of communication across the world. Smartphones and mobile devices have significantly increased the productivity of users through accessibility to important services and have thus become critically integrated into both their work and personal lives [1]. A side effect of this integration is that phones contain a large volume of probative information linked to an individual. The scope of information sensitivity ranges anywhere from low severity (call history, contact information, and text message data) to more valuable information (e-mail, browser history, chat logs, and even passwords) [2]. The ever increasing number of phones available and in use by the public will continue to enhance the convenience of day-to-day life, but simultaneously brings along an increased potential for criminals to misuse this technology.

Traditional desktops are assumed physically secure and need only be protected against software attack vectors (e.g. a firewall for guarding against malicious network attacks), but mobile devices are vulnerable to physical tampering, leading to a variety of previously unconsidered data risks [1]. Most of the sensitive and private information hoarded on cell phones can be legally or illegally breached by someone with the technical expertise and the required equipment. However, the tools needed to perform these attacks are often publicly available in the form of source development tools [3]. Currently, mobile phone systems try to offer security and privacy to users through user authentication, signal encryption, and user anonymity. Nevertheless, these techniques cannot completely guarantee user privacy in domains such mobile commerce, particularly when they are in conflict with resources present on the device (e.g. power, cycles, memory, and bandwidth) [1]. A single event of device theft could lead to financial and information loss, a privacy breach, loss of intellectual property, or even more severe damage. Information security is thus becoming increasingly important, given the ever increasing number of new applications that make use of that data.

Existing disk encryption mechanisms typically deployed in laptops have been shown to be vulnerable to methods that instead focus on obtaining sensitive data directly from memory. One such method is a cold boot attack [4], whereby an attacker gains physical access to a computer and is able to obtain sensitive data either in plain text or indirectly by first obtaining encryption keys from the memory of a running operating system. In particular, if the computer is forcefully powered off during operation (e.g. unplugging the power cord), the operating system will not have a chance to scrub memory of sensitive information. Subsequently, the DRAM chips will retain the data in a readable form for a period of time even after the computer is completely shut down. This period can even be increased by lowering the temperature (temperatures as low as $-50\,^{\circ}\mathrm{C}$ achieved using off-the-shelf air dusters have been shown to preserve up to 99.9% of memory for up to 1 minute [4]). The contents are then read on the same or separate machine by booting the machine with the compromised RAM into a custom kernel that dumps the contents into a separate storage medium. This process thereby grants unauthorized access of a computer's encryption keys when the computer is left physically unattended. Therefore, the disk encryption systems have virtually no safe place to store the cryptographic keys. An attacker can recover encryption keys using key harvesting algorithms even in the presence of 15% bit decay for 128-bit AES keys [4].

Operating systems are still in the early stages of offering a completely secure environment to protect against breaches to secure information in the absence of physical security, as is typical for phones. With attacks like cold boot making foreign memory access easily achievable, some safeguards for storing the encryption keys have been explored. TRESOR [5] is a Linux kernel patch for the x86 architecture that runs the AES algorithm entirely on the microprocessor, storing the secret key in CPU debug registers as a step to protect against key harvesting. This provides the advantage of preserving binary compatibility, preventing user-space applications from accessing them (by patching system calls), and being large enough to store cryptographic keys. This serves as good motivation for the refinement of memory encryption techniques, and motivates the work herein.

In this technical report, we propose the use of GPUs as a hardware accelerator for implementing the Advanced Encryption Standard (AES) algorithm over a traditional CPU based implementation. Modern GPUs are attractive for parallel processing because these architectures have hundreds of processing cores and high bandwidth, delivering up to a teraflop (1 trillion calculations per second) of computing power from the same silicon area as a comparable microprocessor and using only a fraction of the power per calculation [6]. These high-performance and low-energy advantages are a result of the GPU's support for hiding latency in memory transactions through massive multi-threading with low context switch overhead [7]. Modern mobile devices have a system-on-chip (SoC) architecture where the CPU, GPU, and other computing nodes share the same system bus for accessing main memory. Furthermore, there is active work in providing frameworks for performing general-purpose parallel programming on GPUs (GPGPU). One such framework is OpenCL, whereby a host processor (in this case, a CPU) can delegate work to be

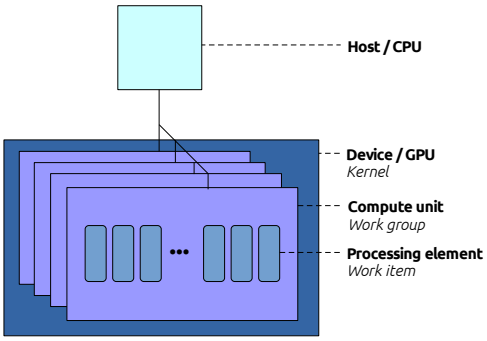

Fig. 1. The OpenCL platform and execution model

performed in parallel by computing nodes (GPUs). We chose the OpenCL framework for implementing the AES algorithm due to its availability on Android phones.

AES is a variant of the Rijndael cipher which acts on input blocks of 128 bits and uses keys of size 128, 192 or 256 bits. The algorithm can work in parallel on each individual block of the input plain text (each block being 128 bits). At a high level, the algorithm does the following. It first goes through a key expansion stage, where the cipher key is used to derive the round keys using a key scheduling algorithm. The plain text is XORed with one of the derived keys (AddRoundKey) before being subjected to a series of rounds, the number of rounds being decided by the size of the cipher key (10 in case of 128-bit keys). The input and output of each of the rounds is modeled as 4 by 4 matrix, called a state, with each cell being a byte in size. In each of the rounds the input goes through the following steps. First, each byte is replaced in a non-linear manner with another using a lookup table known as the T-box (SubBytes). Then each of the rows of the input is cyclically shifted to the left a certain number of steps (ShiftRows). This is followed by a mixing operation which operates on the columns of the state, combining the four bytes in each column (MixColumns). The last round of the encryption algorithm omits this step. The final step in the round is again AddRoundKey where the state is XORed with a derived key from the key expansion step. The output of these rounds is the encrypted ciphertext [8], [9].

In the OpenCL platform and execution model (figure 1), the user creates a host (in our case, the CPU) program that compiles and dispatches a GPGPU program to run on a particular device (the GPU). Upon executing the kernel, the user must specify the number of work groups to execute as well as the size of each work group. The execution model defines work items to be an executing instance of an OpenCL kernel which are run on processing elements, and work groups to be equally sized groups of work items. The OpenCL runtime ensures that work items within a work group will execute concurrently on the same compute unit, but provides no guarantees whether work items across work groups will execute concurrently or serially. All work items within a group are assigned a local id that is unique within that group as well as a global id that is unique across all work items in all work groups of the executing kernel. The work items then make use of their local and global ids to appropriately divide up the work, and can make use of synchronization constructs like a work group barriers to synchronize their execution (but only for work items within the same group) [10].

II. Evaluation

An existing implementation of an OpenCL kernel implementing the AES encryption algorithm was obtained from a public source code repository [11]. The original implementation of the OpenCL encrypt kernel is implemented using a data parallel programming model, whereby a program is parallelized by performing the same computation on different pieces of data [10]. In particular, it only makes use of the global id of a work group to index itself into the input array at a particular position (e.g. index 0 of the input array), with one work group instance being responsible for encrypting an entry of the input array. The kernel also makes use of the vector instruction set of the GPU to increase the number of bytes that a single "entry" in the input array corresponds to. In particular, in addition to traditional analogs to C data types like `uint` (a 32-bit integer for a total of 4 bytes), there are vector data types like `uint4` (4 32-bit integers, for total of 16 bytes). Using these vector types allows one to execute native vector operations for the target platform [10]. In particular, a single work item can operate on 16 bytes at once, whereas a traditional CPU would operate on 4 bytes at once. Thus, given an input array of N bytes, we only need $N/16$ work groups of 1 work item each to encrypt an input array. Hence, in this implementation, the local id is ignored, since a work group only consists of a single work item.

The target device used for conducting experiments was a MotoX phone [12] (table I). We initially intended to use the Nexus 4 phone, but this was abandoned due to apparent instabilities triggered by the use of OpenCL on the device.

Preliminary experiments determined that the full parallelism offered by the GPU was not being exploited using the original implementation. Hence, modifications were made to the original implementation in order to explore the different levels of parallelism offered by varying the input parameters to the kernel such as the number of work groups and the work group size (which were previously strictly $N/16$ and 1 respectively). The following sections describe the modifications that were performed to the original implementation and investigates their effect on the throughput of encryption.

1) Version 1 - even partitioning by work items: We modified the OpenCL AES encrypt kernel such that the input array is split up evenly amongst the work items across all work groups (whereas previously they only encrypted 16 bytes). In particular, all OpenCL instances get some multiple of 16 bytes on which to operate, all of which are equally sized except one OpenCL instance which will be allocated whatever remains of the input array (e.g. given $N = 128$ bytes, and 3 work items, 2 instances will operate on 48 bytes and 1 instance will operate on 32 bytes).

Timing experiments were performed on a 128 MB input array, with the total number of work groups being varied from 1 up to 16, and the work group size being kept at 1 (refer to figure 2). Note that all timing measurements throughout this report are restricted only to the encryption operation and do not not include any input initialization operations.

Going from of 1 up to 4 work groups, we see a linear speedup in encryption throughput (going from work groups of

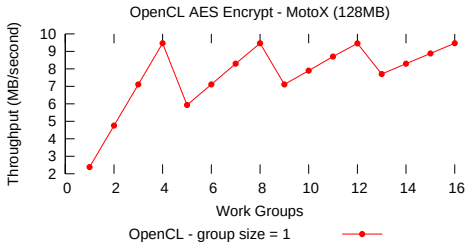

Fig. 2. OpenCL performance over a varying number of work groups

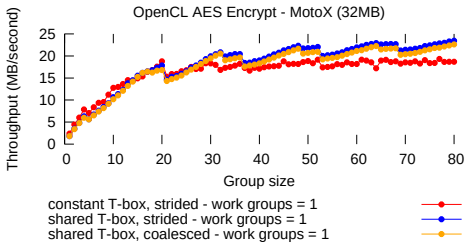

Fig. 3. OpenCL performance over varying work group size

1 to 2 we see a 2.37 MB/second speedup, 2 to 3 we see 2.36 MB/second, and 3 to 4 we see 2.36 MB/second). These linear speedups going from 1 up to 4 work groups are consistent with the device specifications in table I which state that the GPU has 4 computing units (hence, it is using an additional computing unit for each additional work group).

When we reach 5 work groups, we see the worst decrease in encryption throughput (3.54 MB/second). This is most likely due to the OpenCL runtime scheduling 4 simultaneous OpenCL kernel instances (4 work groups each consisting of a single work item) that operate on $1/5$ of the input array, with a single OpenCL instance running on the remaining $1/5$ only after the first 4 complete (thereby underutilizing the parallelism of the GPU).

The throughput degrades the most for numbers of work groups that that are 1 modulo 4 since given that instances will tend to complete in roughly the same time (since they have the same data size and instructions executed), there will tend to be a point at which only one GPU core will be utilized, thus reducing parallelism (similarly for 2 modulo 4, and 3 modulo 4). As we increase the number of work groups, the degradation is less since the point at which we aren't maximally using all 4 GPU cores operates over less of the input array. This phenomenon has been described in [13] as the tail effect.

Hence, from figure 2, the main conclusion to be drawn is that work groups are being scheduled on compute units, but only one work group at a time is being scheduled (otherwise the peaks would reach above 10 MB/second as the number of work groups increases).

Next, we investigated the affect of increasing the number of work items used on a single compute unit. In particular, we began by querying the OpenCL runtime for the maximum available work group size, which was found to be 256. However, this is only a theoretical maximum, and the actual maximum enforced by OpenCL is dependant on the kernel being executed and its resource requirements [10]; in our case, the maximum available was 80. Timing experiments were performed on a 32 MB input array, with the total number of work items being varied from 1 up to 80, and the number of work groups being kept at 1 (refer to "constant T-box, strided" in figure 3).

From figure 3, we can see the encryption throughput gradually increases from 0 up to 20 work items, doubling the maximum throughput observed when only considering a single work item (as in figure 2). However, after 20 work items, we observe some degradation, followed by the throughput leveling off at around 18 MB/second. This decrease in throughput for an increase in parallelism is an unexpected result and warranted further investigation. Previous work has shown that memory access patterns are important to consider when optimizing the performance of a GPU program [14]. One important factor to consider is whether accesses to global memory within a warp [14] or team [10] (i.e. a hardware schedulable division of the work group size) are coalesced. Memory accesses are said to be coalesced within a team if each work item in the team accesses global memory in an aligned, contiguous sequence of 32, 64, or 128 bytes [15]. Hence, in the modification of the implementation in section II-2, we investigate coalesced memory accesses.

Another factor to consider is redundant access to global memory. In particular, access to global memory is the slowest memory operation, whereas access to local memory (shared within a work group) is more efficient [10]. The current implementation would not benefit from the use of local memory for the input array since each work item reads global memory addresses once and encrypts it independently. However, previous work has shown that loading the T-box into the local memory of a compute unit is beneficial, since it is subject to random access [16]. Thus, in section II-3, we explore adding this optimization to the current algorithm.

2) Version 2 - coalesced accesses within work groups: We further modified the kernel (from section II-1) such that in addition to work items operating on equal sized input chunks, the work items within a work group access contiguous sequences.

That is, the kernel in section II-1 performed strided memory accesses [14] within a work group. For example, given a single work group of size 2 where each work item is responsible for encrypting `E` entries, accesses to the input array `A` were as follows:

Work item 0	**Work item 1**
`A[i]`	`A[i + E]`
`A[i + 1]`	`A[i + E + 1]`
...	...
`A[i + E - 1]`	`A[i + 2 * E - 1]`

In the following implementation, the memory accesses within such a work group have been modified such that instead of concurrent memory accesses being `E` entries apart, they are contiguous. That is, if we have a total of `G` work groups, the accesses now look like:

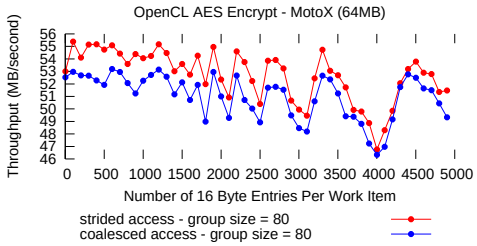

Fig. 4. OpenCL performance for strided vs coalesced accesses within a work group

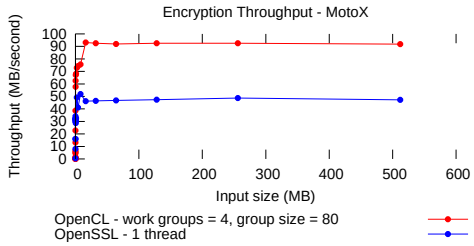

Fig. 5. OpenCL GPU vs OpenSSL performance in AES encryption

Work item 0	**Work item 1**
$\mathtt{A[i]}$	$\mathtt{A[i+1]}$
$\mathtt{A[i+E]}$	$\mathtt{A[i+1+E]}$
...	...
$\mathtt{A[i+E*(G-1)]}$	$\mathtt{A[i+1+E*(G-1)]}$

Timing experiments were performed on a 64 MB input array, with the total number of entries for a work item to encrypt being varied from 1 (16 bytes) up to 5000 (78K). In order to fully utilize all potential processing power, the work group size was set to 80 and the number of work groups was set such that all 4 compute units were used (refer to figure 4).

From figure 4, we generally achieve the best encryption throughput for a small number of entries (< 500), and it tends to get worse at the peaks for increasingly larger sizes per work item. These drops in the peaks could be caused by increasingly large stride distances within a work group.

The fluctuations observed in the graph are due to the GPU being underutilized near the end of the computation. In particular, in order to use a work group size of 80, the OpenCL runtime forces the user to make each work group that size. If the number of work items needed doesn't evenly divide the input, this will cause the final scheduled work groups to underutilize the GPU (and it becomes increasingly bad if a work item is responsible for a larger encryption size).

Comparing the coalesced and strided implementations, we observe nearly identical throughput behaviour. However, we get slightly less throughput for the coalesced implementation, which is likely due to the increased operations being performed for entry index calculations [10]. However, if memory coalescing is in fact the reason for poor performance in the strided implementation, then the coalesced graph indicates that memory accesses are still not coalesced. Alternatively, the memory access are already coalesced in the original implementation (i.e. it may be the case that reading 16 bytes from global memory at a time is the maximum), in which case increasing the length of contiguous accesses will result in no increase in throughput.

To rule out the possibility that any benefit from coalesced memory access was not being realized due to interference from not using the shared T-box optimization of section II-3, we implemented both optimizations (see "shared T-box, coalesced" in figure 3). However, we again observed a constant decrease in performance.

Hence, subsequent modifications did not build on the coalesced optimization, and instead use the original implementation as described in section II-1.

3) Version 3 - storing the T-box in local memory: The original implementation of the kernel stored the 4 KB T-box in the constant memory region [11]. However, the constant memory region is simply a read-only section of global memory. The MotoX compute units were found to have a local memory size of 8 KB, making this optimization possible. The kernel in section II-1 was modified to have work items in a work group evenly divide the task of copying the T-box into local memory (synchronized via a local memory barrier) prior to performing encryption. To measure the affects of this optimization, we reran the timing experiments for investigating the affect of increasing the number of work items used on a single compute unit (section II-1, refer to "shared T-box, strided" in figure 3).

From figure 3, we see an increased maximum throughput (23.14 MB/second) over the previous implementation. Further, we observe a less pronounced leveling off in throughput, with the maximum throughput achieved at the maximum work group size (80). Hence, this implementation was used in the comparison against CPU performance.

4) GPU vs CPU Performance: Next, we evaluate the performance of encryption of the GPU over the CPU using the implementation in section II-3. Figure 2 was used to guide the selection of an optimal number of work groups (4) and figure 3 was used in choosing an optimal work group size (80). The CPU implementation of AES encryption uses the OpenSSL crypto library [17], using a 256-bit AES key and the CBC encryption mode. As was done the GPU timing measurements, the CPU timing measurements are restricted only to the encryption operation. Timing experiments were performed on input sizes varying from 16 bytes up to 512 MB incrementing by powers of 2 (refer to figure 5).

From figure 5, we are able achieve a maximum throughput of 93.07 MB/second at an input size of 16 MB, whereas the CPU is able to achieve 51.87 MB/second at an input size of 8 MB (though it quickly degrades to 46.25 MB/second for an input of 16 MB). Hence, the GPU's increase in maximum throughput speed is 46.82 MB/second over the CPU, with a speedup of 1.79. While we have nearly doubled the CPU's performance, this margin is still small based on previously published experiments investigating GPU encryption acceleration on the desktop using CUDA, which have found up to a 28.39 fold speedup in encryption [18]. However, this may be partly explained by the abundance of cores available on such GPUs in comparison to mobile GPUs.

TABLE I. MOTOX DEVICE SPECIFICATIONS

Name	Value
Manufacturer	Qualcomm
Chipset	Snapdragon S4 Pro MSM8960DT
CPU	2x Qualcomm Krait 300 Harvard Superscalar processor
GPU	400 MHz quad-core Adreno 320 MP4 GPU
GPU device memory	898 MB
Compute units	4
Max work group size	256
Constant buffer size	4096 bytes
Cache line size	16 bytes
Profiling timer resolution	1000 nanoseconds
Local memory size	8192 bytes
Max clock frequency	325 MHz

III. RELATED WORK

CleanOS is a modification of Android that introduces an abstraction called Sensitive Data Objects (SDOs) to track sensitive data in RAM, created and hoarded by apps that run on top of the Dalvik VM [19]. The system works by defining SDOs (either by the apps or by the default OS), which are a logical collection of Java Objects that contain sensitive data. It introduces a modified garbage collector known as EvictIdleGarbageCollector (eiGC) which walks through an idle SDO and encrypts their data-bearing fields using a key which is then evicted to the cloud whereas the data remains encrypted in memory. This system, like ours, prevents the download delay of the encrypted data from a remote host as the data is always present in the device itself. Moreover, the key is inaccessible to a malicious user stealing the phone.

However, CleanOS incurs the latency of a network request when retrieving the key from cloud. In particular, CleanOS will incur unpredictable pauses in the application during an otherwise inexpensive memory access on an encrypted value. Due to the implementation of encryption at the Davlik VM level, the application doesn't have a means for knowing when the pauses are going to happen, and hence cannot deal with them by computing something else in the meantime, or avoiding the request altogether. In fact, evaluations show that many operations (like loading of an email onto the screen) incurred overheads of more than 100% compared to a non-CleanOS implementation, especially over 3G networks [19].

In contrast, our system intends to store the key on the device in a special register accessed by the operating system and not user-level applications. We can use techniques like TRESOR [5] for this, which ensures that the key is only stored in CPU registers which are not accessible to user applications (like debug registers) and that the key is only brought into RAM for a very short period on reboot even before all other applications come to life. The key is cleared from RAM soon after this. Another major shortcoming of CleanOS is that it only encrypts data (memory) which is managed by the Dalvik VM, and does not account for data that is unencrypted in other parts of the operating system [19]. This includes OS data buffers (e.g. for writing to a file, inside device drivers from reading from sensors such as a camera).

XOM is a system to prevent software piracy, wherein the executing code of the software which is in RAM is encrypted by marking it as "execute-only" and so is prevented from tampering [20]. Cryptkeeper is a software-encrypted virtual memory manager that segments RAM into two segments, one for running the decrypted program instructions (small in size) and the other for keeping the encrypted pages. The pages are swapped automatically between the segments and decrypted on demand [21]. Encrypted Swap aims at encrypting pages before swapping them to swap memory so that sensitive data related to swapped out processes are not available in backing storages [22]. These systems all have the same disadvantage that, even though the data is encrypted, the decryption key is still left in RAM and so can be recovered through memory-harvesting techniques.

IV. CONCLUSION

In this report, we have taken the first steps in investigating the feasibility of using the GPU as a cryptographic accelerator for the AES algorithm on mobile devices. In particular, our focus was on exploring the use of OpenCL as a framework for implementing the algorithm. Using modifications of an existing implementation [11], we first showed that OpenCL schedules work groups to execute exclusively on the compute units of the GPU, and that optimal throughput must be achieved through more than just a simple data parallel implementation. Next, we investigated how performance varied with the implementation as the number of active work items in a single compute unit was varied, discovering a surprising level off in encryption throughput when using only $1/4$ of the available work items. Next we investigated two potential reasons for lower than expected performance: strided access to input data within work groups, and redundant random accesses to the T-box lookup table that could be avoided using local memory. We discovered the T-box lookup table optimization provided the best increase in throughput, whereas the coalesced access implementation only added to the runtime. Finally, we determined that using optimal parameters for the number of work groups and work group size, we were able to achieve a 1.79 fold speedup in maximum throughput over a single CPU using the OpenSSL implementation.

V. FUTURE WORK

Our next endeavour would be to evaluate other characteristics of the GPU implementation of AES relevant to mobile devices, such as energy consumption during encryption. Finally, we would investigate methods for securely providing a privileged user-level process the permissions and capabilities needed to encrypt memory on behalf of the Linux kernel.

REFERENCES

[1] A. K. Ghosh and T. M. Swaminatha, "Software security and privacy risks in mobile e-commerce," *Communications of the ACM*, vol. 44, no. 2, pp. 51–57, 2001.

[2] "Ponemon Institute - The lost smartphone problem," http://www.mcafee.com/us/resources/reports/rp-ponemon-lost-smartphone-problem.pdf, accessed: 2013-12-04.

[3] J. Lessard and G. Kessler, "Android forensics: Simplifying cell phone examinations." 2010.

[4] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[5] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *USENIX Security Symposium*, 2011.

[6] K. Kothapalli, D. S. Banerjee, P. Narayanan, S. Sood, A. K. Bahl, S. Sharma, S. Lad, K. K. Singh, K. Matam, S. Bharadwaj *et al.*, "CPU and/or GPU: revisiting the GPU vs. CPU myth," *arXiv preprint arXiv:1303.2171*, 2013.

[7] C.-L. Duta, G. Michiu, S. Stoica, and L. Gheorghe, "Accelerating encryption algorithms using parallelism," in *Control Systems and Computer Science (CSCS), 2013 19th International Conference on*. IEEE, 2013, pp. 549–554.

[8] P. FIPS, "197, Advanced encryption standard (AES)," *National Institute of Standards and Technology*, 2001.

[9] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[10] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.

[11] "GitHub - softboysxp/OpenCL-AES," https://github.com/softboysxp/OpenCL-AES, accessed: 2013-12-04.

[12] "PDAdb.net - Qualcomm Snapdragon S4 Pro MSM8960DT multi-core application processor with Modem Datasheet," http://www.pdadb.net/index.php?m=cpu&id=a8960dt&c=qualcomm_snapdragon_s4_pro_msm8960dt, accessed: 2013-12-04.

[13] "NVIDIA GPU performance analysis and optimization," http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf, accessed: 2013-12-04.

[14] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 86–97.

[15] "NVIDIA - OpenCL programming guide for the CUDA architecture," http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, accessed: 2013-12-04.

[16] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, "Implementation and analysis of aes encryption on gpu," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012, pp. 843–848.

[17] "OpenSSL - documents, crypto(3)," http://www.openssl.org/docs/crypto/crypto.html, accessed: 2013-12-04.

[18] K. Iwai, T. Kurokawa, and N. Nisikawa, "AES encryption implementation on CUDA GPU and its analysis," in *Networking and Computing (ICNC), 2010 First International Conference on*. IEEE, 2010, pp. 209–214.

[19] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: limiting mobile data exposure with idle eviction," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA*, 2012.

[20] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[21] P. A. Peterson, "Cryptkeeper: improving security with encrypted ram," in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*. IEEE, 2010, pp. 120–126.

[22] N. Provos, "Encrypting virtual memory," in *Proceedings of the Ninth USENIX Security Symposium*, 2000, pp. 35–44.