Executing Parallelized Dictionary Attacks on CPUs and GPUs

Hassan Alnoon
ENSIMAG
Grenoble, France
hassan.alnoon@ensimag.imag.fr

Shaima Al Awadi
ENSIMAG
Grenoble, France
shayma.al-awadi@ensimag.imag.fr

ABSTRACT

This paper presents dictionary based attacks in addition to their corresponding MD5 and SHA1 implementation on GPU and CPU. All computational time testing was programmed in CUDA-C language. Kaapi Adaptive Standard Template Library (KASTL) was utilized to aid in performing parallel CPU operations complimenting today's widely spread systems with multi-processors or multi-core processors. Comprehensive analysis of performed dictionary attacks are detailed throughout this paper with performance results highlighted towards the conclusion.

Keywords

Dictionary Attack, Cryptographic Hash Function, GPU, CPU, MD5, SHA1, KASTL, CUDA, Parallel Execution, Password Cracking.

1. INTRODUCTION

In computer security, a dictionary attack is a technique used by cryptanalysts to break the security of the system by attempting to retrieve its decryption passphrase by searching all likely possibilities. Operating systems store passwords in their message digest form, computed from one way functions. In cryptography, one-way functions are procedures that are easy to compute for a given input, yet complex to inverse in polynomial time. A hash function takes an input and produces a string of fixed size, often called message digest. Operating systems store hash values of passwords and compare them with message digests of user-entered keys.

Cryptographic hash functions and their respective implementation in dictionary-based attacks on graphic processors are receiving considerable attention worldwide. Realtime modeling of parallelized cryptographic hash functions has expedite performance expectations detected on theoretical modeling. It was found that cryptographic hash functions perform much faster and more efficiently on GPUs than they do on CPUs. Because GPUs allow for parallel thread execution, parallelized hash functions are the natural choice for fastest dictionary attack results; especially when dealing with extremely long messages.

This paper takes a deeper look at dictionary based attack on MD5 and SHA1 based passwords, and discusses their implementation on CPU and GPU, analyzing executions and comparing results.

Objectives:

- Detect any factors that influence the performance of dictionary based attacks on cryptographic hash functions on multicore processors and multi-processor systems.
- Determine the feasibility of performing dictionary based attacks on hash values on GPU compared to CPU.
- Explore the acceptability of the new role that GPUs take in today's computerized systems

2. DICTIONARY ATTACKS BASED ON MD5 AND SHA1

2.1 DICTIONARY ATTACKS

Dictionary based attacked are basically using brute force technique to systematically go through a list of words from a specific wordlist of choice. These words list can be based on words from the dictionary or commonly known used password list that are available all over the internet. The success in a dictionary based attacks depends on the wordlist used; that is the amount of possibilities to go through. Today, most webpages and software's store the user passwords hashed using one of the many available one-way hash functions.

In our report we focused on MD5 and SHA1 hashed passwords, as they are from the top commonly used hashed functions. We tackled this project by using a wordlist that was available online with the most commonly used words. The idea is to hash each word in the wordlist and then to compare to the hash of the word that e want to crack.

2.2 Parallelization of Dictionary Attacks

Recovery of hash digests of passwords using dictionary words is ideal for parallel computing. In essence the dictionary based attacked are parallel in nature due to ability to perform each operation independently without any dependencies between them., and the fact that modern iterated hash algorithms require sufficiently long time to compute on single core processor especially for long messages, extension of dictionary based attack using hash algorithms to support parallel computing will significantly increase performance of a cracking the hashed password.
The parallelization on the CPU is a bit more complex then parallelization on GPU. CPU are made more suited for parallel task situations where processes run parallel but require communication and possibly have dependencies between them, where as GPU are developed to handle parallel data processing, due to the nature of it usage, that is handle graphics, where units of graphics operations are done in parallel with hardly any dependencies.

2.3 MD5

Based on the earlier hash function MD4, MD5 was designed by Ron Rivest; last in succession of cryptographic hash functions. It became an internet standard and has been integrated in a variety of security applications such as SSL/TLS and IPSec.

MD5 uses Merkle-Damgard paradigm such that the security of the hash function reduces to the security of its relevant compression function.

MD5's algorithm is designed to take an arbitrary input and produce a fixed 128-bit output. The input is padded such that its length is 448 modulo 512, and a 64-bit representing the length of the message is appended before padding. A four-word of 128-bit buffer (A, B, C, D) is initialized as follows:

A =	01234567
B =	89ABCDEF
C =	FEDCBA98
D =	76543210

Then, the message is processed in 16-word (512-bit) chunks, using 4 rounds of 16-bit operations each.

The compression function of MD5 is composed of rounds, where each round has 16 steps of the following form:

$$\boldsymbol{a} = \boldsymbol{b} + ((\boldsymbol{a} + \boldsymbol{g}(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}) + \boldsymbol{X}[\boldsymbol{k}] + \boldsymbol{T}[\boldsymbol{i}] <<< s)$$

Where:

- a, b, c, d are the four words of the buffer, but are used in varied permutations.
- g(b, c, d) is a different nonlinear function in each round (F,G,H,I). for example, in round 1, $\boldsymbol{g}(\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}) = (\boldsymbol{b} \wedge \boldsymbol{c}) \vee (\boldsymbol{neg}(\boldsymbol{b}) \wedge \boldsymbol{d})$
- T[i] is a constant value derived from the sin function and X[k] is derived from a 512-block of the message.

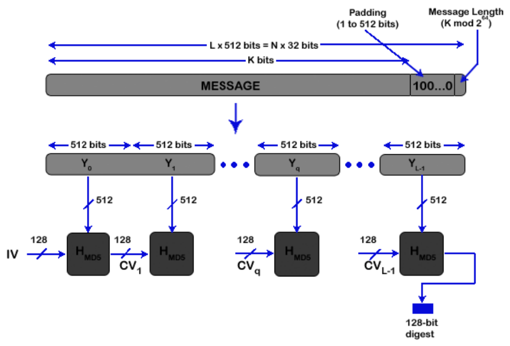

Figure 1 Message Digest 5 Hashing Algorithm: takes message of arbitrary size and after a sequence of procedures and rounds, produces an output of 128-bits.

2.4 SHA 1

Unlike MD5, SHA1 produces a 160-bit output digest from a message with a maximum length of $2^{64\ -1}$ bits. Although SHA1 is based on the same principles that Ron Rivest used for MD5, SHA1 has a more conservative design than that of MD5.

In SHA1, the message we have is padded in order to become a multiple of 512 bits, and it is split into 16 32-bit words, 512 blocks. A 5-word of 160-bit buffer is initialized as follows:

A =	67452301
B =	EFCDAB89
C =	98BADCFE
D =	10325476
E =	C3D2E1F0

The message is then processed in 16-word (512-bit) chunks as follows:

- Expand 16 words into eighty words by mixing and shifting.
- Use 4 rounds of 20 operations on message block and buffer.
- The compression function has rounds of 20 steps each, and updates the buffer as follows: $(A, B, C, D, E) \leftarrow (E + f(t, B, C, D) + (A \ll 5)\ W\_t + K\_t\ ), A, (B \ll 30), C, D)$
    - t is the step number,
    - f(t,B,C,D) is a non-linear function for the round,
    - W_t is derived from the message block
    - K_t is a constant value derived from the sin function
    - S_k is a circular left shift by k bits

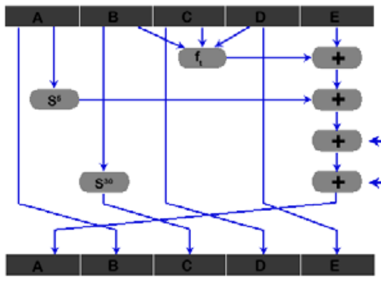

Figure 2 SHA1 Hash Algorithm

3. EXPERMENTATION PROCEDURE

To do our experimentation used a wide range of words in each wordlist file. The number of words that we tested for are around: 0.5million, 1million, 2million, 4million, 6million and 8million. We also tested on files smaller then 1million, just to see the performance between the GPU and CPU in computation and GPU memory copy procedure time. But due the speed that the hashes were computed it was best test on file with large amount of data. The wordlist were retrieved from different sources on the net, but mainly hashkiller.com, which has a list of combination of dictionary word and also commonly used password by users. The words in the file were increased by adding arbitrary words. There is a possibility of the repetitiveness of the word in these huge files, but these files are used for performance testing and repetitiveness of the word doesn't affect the result.

Our main aim was to try and test the time it takes to compares hash with the hashes of all the words in the list, so worst cases scenario was taken when the hash wasn't in the list.

The systems for the test are the following:

Table 1. Systems used for the experimentation

ID.	Processor	GPU Card
System 1	INTEL Core 2 Duo 2.4 GHZ	NVIDIA Quadro NVS 320M
System 2	INTEL Core 2 Duo, 2.5 GHZ	NVIDIA GeForce 8600 GS
Idkoiff	8 * AMD Opteron 875 , total 16 cores, 2.2 GHZ	2 x NVIDIA GeForce GTX 280

Throughout this paper wherever required we would refer to the systems illustrated in Table 1 as System 1, System 2 and idkoiff.

4. DICTIONARY ATTACKS ON CPUs

4.1 KASTL

KASTL (*Kaapi Adaptive Standard Template Library*) is a tool that was developed by the INRIA lab that runs on top of KAAPI. It allows for the parallelization of STL algorithms based on work stealing implementation of KAAPI. In general it allows parallelizing certain tasks, especially loops that are processed on CPU with multi-cores. There are other tools in the market that allow us to parallelize tasks such as Cilk++ and Intel Thread Building Blocks (TBB). We had several issues trying to get Cilk++ to work on our systems, and due to time restrictions, we decided to perform our parallelization with KASTL as we are also been able to get in touch with the developer if any issues were raised .

4.2 Implementation

4.2.1 MD5

To enable us to parallelize the MD5 implementation we had to first modify the code we found on the web that was developed by Mario Juric[4]. The code has two mode of operations a search and just a general overall hashing function. We were interested in the search, but it was only implemented for GPU. Our first task was to create a function that would allow us to do the search using the CPU processing in one run of the program, and also keeping in mind that we would like to be able to parallelize it too with KASTL. The implementation of MD5 CPU processing was straightforward but we had to a lot of issues with getting KASTL implementation done properly. We took the assistance of MR. Traore whose PHD thesis is based on KASTL.

The STL algorithm that we were trying to parallelize was `std::transform`, and so the loop for the hash was modified to make use of the transform function. Once that was achieved, we converted the `std::transform` function to the `kastl::transform` which takes in the same concept but with slight modification. We're basically giving it extra parameters indicating where the output result should be stored.

4.2.2 SHA1

For the SHA1 hash implementation we used the code that was developed by Mr. Vilkeliskis[5]. The code had the SHA1 implementation for the CPU straightforward, so, we plugged it in the program we used for MD5, where for each loop through the wordlist, instead of hashing it with the MD5, we hashed it with SHA1, and the same we did for the word that we wanted to crack.

The kastl::transform function stayed the same as for MD5, so that helped speed up the implementation for SHA1.

4.3 Experiments

The experiments were done on the three systems mentioned in table1 as per the experimentation procedures mentioned in section 3 of this report. Each systems total core was exhausted in each test performed, to make sure that we parallelize the operation as much as possible. Figure 3 and Figure 4 show the performance of the systems upon executing MD5 and SHA1 algorithms on each.

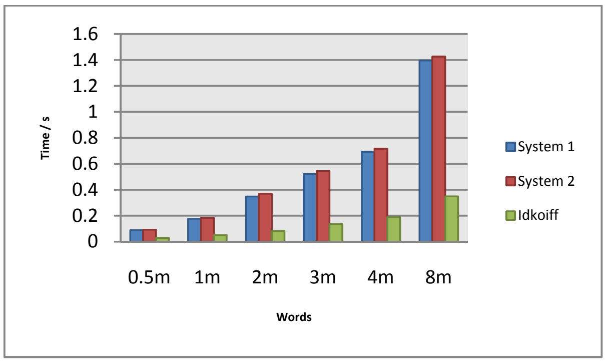

Figure 3 MD5 Systems Performance

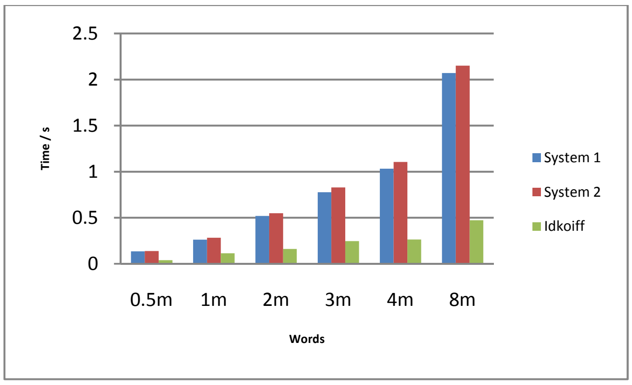

Figure 4 SHA1 System Performance

We can clearly see that performance on idkoiff was much higher then the performance of the other two systems, as idkoiff has a total of 16 cores available for usage. We also noticed as was expected from reports read that dictionary attacks on SHA1 was slower than MD5. We can definitely say that the increase in core of the process almost cut the processing time by half.

5. PARALLEL DICTIONARY ATTACKS ON GPU

5.1 CUDA

Graphic Processors are difficult to program for general-purpose uses. Programmers can either learn graphics APIs or convert their applications to use graphics pipeline operations, or they can use stream programming abstractions on GPUs. NVIDIA released a software development kit named Compute Unified Device Architecture (CUDA) for its graphics hardware in February 2007. CUDA allows programmers to access the computing power of GPU directly. Programmers use C for CUDA to develop programs for execution on GPUs. CUDA's most utilized benefits is its use of shared memory, a fast region that can be shared amongst threads.

5.2 Implementation

5.2.1 MD5

The code used for the GPU was mainly taken also from the same developer who we used the MD5 CPU implementation from Mr. Juric[7]. The GPU implementation he had at a first glance seem to be working just as we wanted. There was slight modification done on the code, mainly separating the CPU and GPU operations so that they can be run independently, and also simplify the running procedure without the extra parameters that the developer included for benchmarking and extra search feature. In the initial stages, a lot of work was done trying to tweak the shared memory, and the numbers of threads per blocks that is assigned for each process to see if it would increase the performance of the GPU operation.

5.2.2 SHA1

The code used for the GPU implementation of SHA1 was developed by Mr. Vilkeliskis. The implementation was done for single hashing through GPU, without taking full effect of GPU capabilities. He did have a benchmarking technique where he goes through a list of arbitrary values to check the performance. These values are not passed from __global__ function but rather, are looped from within the device. This technique wouldn't have been useful for our implementation as it would mean for each word we have to copy it individually to the device and then run it, and also doesn't take full effect GPU parallelism capabilities.

Our implementation for SHA1 on GPU was then consistent of two main tasks, trying to use the MD5 implementation of GPU that is copying the words as a batch to the device, and to try and use the shared memory that was used in the MD5 implementation. This was basically a straightforward implementation from the GPU MD5 function with a slight tweaking to enable us to produce SHA1 hashes rather than MD5 hashes. The SHA1 implementation interface was also adopted to use the same interface as the MD5 implementation.

5.3 Experiments

The experiments in this section were also done on the 3 systems mentioned in table1 as per the experimentation procedures mentioned in section 3 of this report. We have provided figure for both the SHA1 and MD5 dictionary attack times, with and without memory consideration. We found out that memory operation takes a lot of overhead of the overall GPU processing time. It was unexpected to find that even though the idkoiff processing time was extremely fast compared to the other systems, the memory operation threw off the results once we considered the memory operation of copying from and to the device.

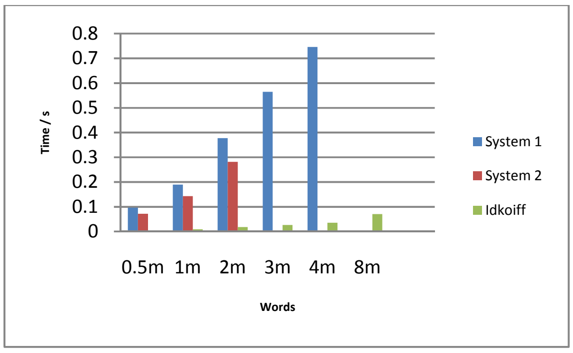

Figure 5 MD5 GPU without memory operation consideration

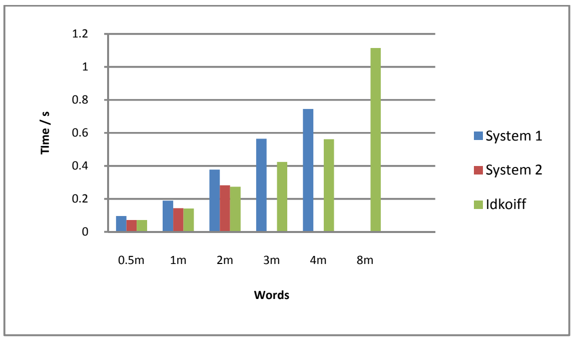

Figure 6 MD5 GPU with memory operation consideration

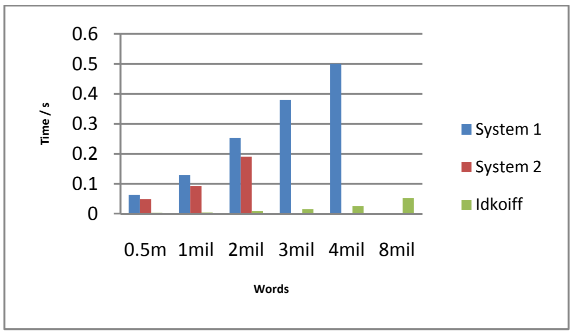

Figure 7 SHA1 GPU without memory consideration

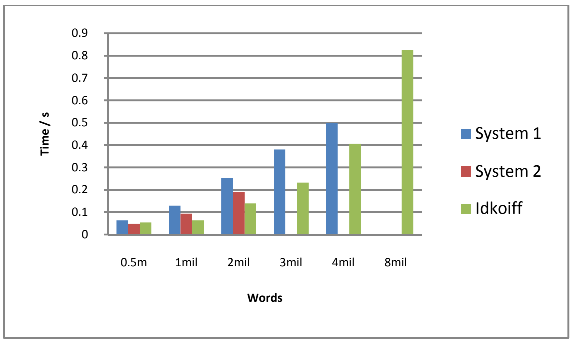

Figure 8 SHA1 GPU with memory consideration

6. COMPARATIVE ANALYSIS

We noticed overall the performance on the GPU were much higher than that of CPU. In exception is the idkoiff system where even the processing time of each thread was processed extremely fast almost 5x faster than the CPU, but with the memory consideration it drops it down to around 1.5x faster. Though there was a problem with the copy to the GPU memory, as it is very slow and time consuming which dropped the GPU performance on that system much slower than that of the CPU 16-multicore performance. To sum up we added two extra figure showing the GPU and CPU multi-core performance on the idkoiff server with and without the memory consideration.

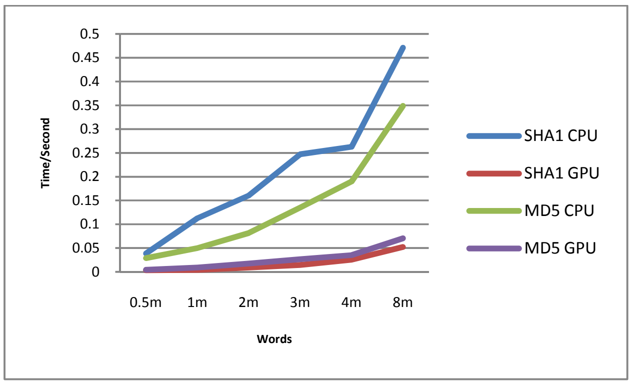

Figure 9 Idkoiff system CPU and GPU performance without GPU memory consideration

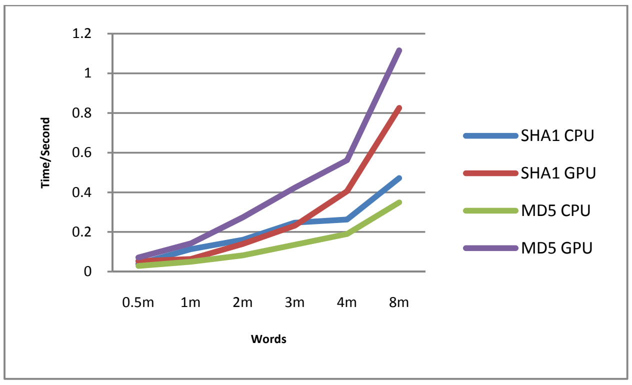

Figure 10 Idkoiff system CPU and GPU performance with GPU memory consideration

7. CONCLUSION AND PERSPECTIVE

Finally, we were able to see the difference in performing dictionary based attacks on one-way cryptographic hashing functions in particular MD5 and SHA1. Overall in both GPU and CPU tests cracking MD5 hashed passwords are faster than that of SHA1. Also using GPU to crack the hashed password did give a kick to the speed, except maybe in Idkoiff system where the overhead in memory exchange was a lot.

It would have been more interesting to been able to include a wider range of one-way hashing functions such as SHA256 and SHA512 or any of the new hashing functions that have been submitted to the NIST hash function competition such as MD6 or Skein in the test that were performed.

8. REFERENCES

[1] Zonenberg, A. 2009. Distributed Hash Cracker: A Cross-Platform GPU-Accelerated Password Recovery System. Rensselaer Polytechnic Institute.

[2] Ruane, J. 2006. General Purpose Computing with a Graphics Processing Unit. Dublin Institute of Technology. Vol 1.

[3] Bernaschi, M. Bisson, M. Gabrielli, E. & Tacconi, S. 2009. An Architecture for Distributed Dictionary Attacks on Cryptosystems. Journal of Computers. Vol 4. No. 5.

[4] Vilkeliskis, T. 2008. Computing SHA message digest on GPU (PowerPoint). Retrieved from http://andernetas.lt/vptr/wp/?page_id=246

[5] Vilkeliskis, T. 2008. Computing SHA message digest on GPU [Software]. Available from http://andernetas.lt/vptr/wp/?page_id=246

[6] Collange, S. Daumas, M. Dandass, Y. S. & Defour, D. Using Graphics Processors for Parallelizing Hash-based Data Carving. 2009. Proceedings of the 42nd Hawaii International Conference on Computer Sciences. Hawaii, United States.

[7] Juric, M. 2008. CUDA MD5 Hashing Experiments [Software]. Available from http://majuric.org/software/cudamd5/

[8] Radev, R. 2008. GPU Computing CUDA (PowerPoint). Retrieved from http://cluster.phys.uni-sofia.bg/ross/CUDA-pres.pdf

[9] Farber, R. 2008. CUDA, Supercomputing for the masses. Retrieved from http://www.ddj.com/hpc-high-performance-computing/207402986