GPU-Based Parallel Signature Scanning and Hash Generation

Bernhard Fechner, FernUniversität in Hagen, 58084 Hagen, Germany

Abstract

Today, nearly every user of electronic devices is affected by threats. Computer viruses infect harmless programs and change the function of that program. One means against these threats is a virus scanner, searching for signatures of known viruses within code and/or data. In this work, we present a novel approach to on-line virus scanning and hash calculation with the help of GPUs (graphics processing units). The main idea is to speed up the search by not searching for signatures on the hard-drive but to scan parts of the main memory and concurrently generate hash values on the code to detect changes. This is appealing and obvious, as the pattern matching has to look at most characters in the text. The first experiments showed that the CPU needs much more time for the execution of shift-like operations and testing for equivalence than the GPU, demonstrating that pattern matching algorithms and hash functions can be efficiently computed on GPUs. As a basis, we use the SHA-1 (160 bit) algorithm. The algorithm uses text properties for the initialization of the hash value and partial sums and/or path properties for the constants in each round. A fundamental part of future research will therefore be the testing if such an initialization can help to detect concatenated extensions like h(m)=h(m·x), where m and x are messages of arbitrary length, furthermore, if the independent calculation of hash results in each round will increase or reduce collision resistance.

1 Introduction

The bandwidth needs for Internet applications grow constantly. Data must be processed at low latencies to achieve high throughput. Ever since the development of comfortably programmable GPUs, there has been the idea to speed up calculations. For an overview of general purpose computations on graphics hardware, please see [5] and [11]. The market pressure urges companies to develop applications for the everyday user, not for distinguished engineers. One of these applications is a virus scanner [7]. At the moment several companies seem to work on this idea but none of them has published, except the basic concept [8] and implementation hints. The main work of a virus scanner is pattern matching, which is also the main process for intrusion detection systems (IDS), already implemented for GPUs [9]. The data must be transferred from the network to the GPU, today over the PCIe bus. The basic bandwidths (the time unit is seconds) of PCIe 2.0 (2.5 GHz bus frequency) interfaces, as well as for Ethernet [10] and SATA standards are depicted in Table 1. We show these values, since they represent raw bandwidth values per second for all devices mentioned in this work. One bottleneck of virus detection which is not described nor considered in [8] is the hard-drive. The bandwidth of hard-drives (b_{hd}) is much lower than the bandwidth of the graphics card bus (b_{bus}). A parallel implementation does not seem to make sense if the achieved bandwidth is $b_{scan} < b_{bus} \vee b_{scan} < b_{hd} \vee b_{bus} \ll b_{scan} \vee b_{hd} \ll b_{scan}$.

		Bandwidth		
PCIe-Slot	**Lanes/ Direction**	**PCIe GByte**	**Ethernet Mbit/MByte**	**SATA Gbit/GByte**
x1	1	0.5	1/0.125	N/A
x4	4	2	10/1.25	N/A
x8	8	4	100/12.5	1.5/0.1875
x16	16	8	1/0.125	3.0/0.375
x32	32	16	10/1.25	6.0/0.75

Table 1 Raw bandwidths of PCIe, Ethernet and SATA

This work is organized as follows: in Section 2, we present related work. Section 3 deals with pattern matching and hash generation for GPUs. Section 4 concludes the paper.

2 Related Work

Several pattern matching algorithms have been proposed in the past decades. First, we start to discuss single-pattern-matching, then multi-pattern-matching algorithms. The easiest is the naive algorithm with time complexity O(n·m), comparing each character in s with each in t. Boyer-Moore[13][14] extend the search by moving s forward by more than one position. This is done by two heuristics, bad-character and good-suffix. Horspool [15] uses the bad-character strategy not with the character causing a mismatch, but the rightmost character within the text window. Sunday [16] uses the character adjacent right to the text window. The skip-search algorithm [17] truncates the text in multiple areas with length n+1 and searches within these areas. For an overview and description of these and many other algorithms, please see [18]. The Aho-Corasick algorithm, the basis for the UNIX command-line tool fgrep [1] and the Wu-Manber algorithm [2] are the most widely adopted multi-pattern matching algorithms. For the requirements of quick deep packet inspection, hardware-based solutions like reconfigurable silicon hardware [3] and TCAM-based solution [4] have been proposed and implemented, but they are usually expensive and not flexible enough. Commentz and Walter [19] combine Boyer-Moore with Aho-Corasick. In practice it is substantially faster than Aho-Corasick. Baeza-Yates [20] proposed an algorithm that combines Boyer-Moore, Horspool [15] and Aho-Corasick. We will not discuss the various works in the direction of hash calculation and cryptographic hash functions here and redirect the interested reader to [21]. We regard unkeyed hash functions to verify the integrity of a message (Modification Detection Codes - MDC[1]). The algorithm on the GPU should be protected by a hash and check itself against modification. With keyed hash functions, a user and a code section can be associated in a multi-user system. Encryption with GPUs (block cipher) is also possible and described in [26]. However, due to the timely limitations of our work, we develop a simple hash function which is not cryptographically strong but clarifies the main idea. We will therefore speak of a *hash value* instead of an MDC.

[1] In contrary to the comment of Cook et al. [22] we are of the opinion that current graphics processors are also capable to compute message authentication codes (MACs).

3 Parallel Pattern Matching and Hash Computation

Virus detection relies heavily on pattern matching. We do no focus on network security, but on workstation security for two main reasons. Firstly, the applicability, since the on-line scanning for virus signatures and the protection of code and data against modification is of concern for every user. Secondly, high bandwidth can only be achieved when regarding memory locations. Pattern matching algorithms compare data within code or data streams against a database of known viruses (*signatures*). Different signature lengths etc. require each input byte to be read and processed many times. This offers the chance to compute a hash value in parallel. Unfortunately, we already seem to have a trade-off here, since the hash should include every byte of the message and the matching should exclude as many bytes as possible to achieve high throughput. The additional needs for the pattern matching and the hash calculation in our work are:

- All algorithms must (to a great extent) support operations which can be carried out by the GPU – our task is not fulfilled if vast amounts of the workload are done by the CPU
- the implementation must be as simple as possible to exclude programming flaws
- the results from one algorithm can be efficiently (re-) used by the other one
- naturally, the implementation should be faster than the sole CPU implementation

We describe the problems separately.

Problem Definition #1: Exact (Single/ Multi)-Pattern Matching without Wildcard

Let $\Sigma \neq \emptyset$ a finite alphabet. Let $n \in \mathbb{N}$ be the length of the search string $s=s_0...s_{n-1} \in \Sigma^n$ and $m \in \mathbb{N}$ the length of the string $t=t_0...t_{m-1} \in \Sigma^m$ to be scanned. A single pattern is found, if the Hamming distance $Hd(s, t_i \ldots t_{i+m-1})=0$. Other distances can be used, if we want to support no exact match, e.g. wildcards with $Hd(s, t_i \ldots t_{i+m-1})<k$ or the Hausdorff distance [24]. If we have multiple search strings with different lengths, we have a multi-pattern match.

Problem Definition #2: Hash Function

A (compressive) hash function h maps a finite message $m=m_0...m_{n-1} \in \Sigma^n$ of arbitrary finite length to a string of fixed length. Alternatively (see [21]) for a domain D and range R with $h: D \rightarrow R$, $|D|>|R|$ results. Compressive hash functions imply the existence of collisions (pairs of inputs with identical output). A cryptographic hash function should fulfill the three well-known properties:

- Collision resistance: It is infeasible to find x, y, x ≠ y such that H(x) = H(y) in appropriate time.
- Preimage resistance: Given an output value y, it is infeasible to find x such that H(x) = y in appropriate time.
- Second preimage resistance: Given an input x', it is infeasible to find x such that H(x) = H(x') in appropriate time.

The term *appropriate time* means that the time to solve the problem with state-of-the-art computational means exceeds the lifetime and therefore the worth of the information to be changed. Note, that we changed the definition from *computationally infeasible* [21] to *appropriate time*. Figure 1 shows the basic function of the matching/hashing algorithm. The code section of the processes to monitor is transferred to the GPUs texture memory. The code section (CS) is found by using the task state segment (TSS) of the operating system. Therefore the process to transfer the code must run with operating system privileges. The GPU calculates the hash on the code section. The hash is also stored in the texture memory. In parallel to the hash calculation the pattern matching is done. Note, that the source for the transfer need not be a code section. It can e.g. be a buffer holding a data stream.

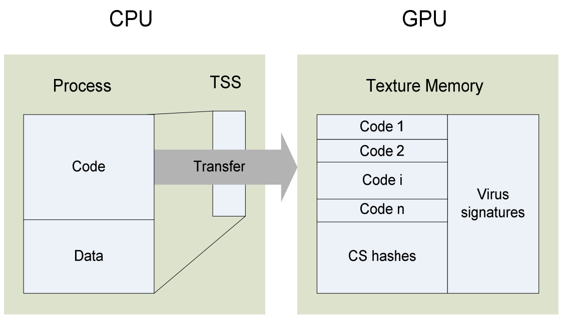
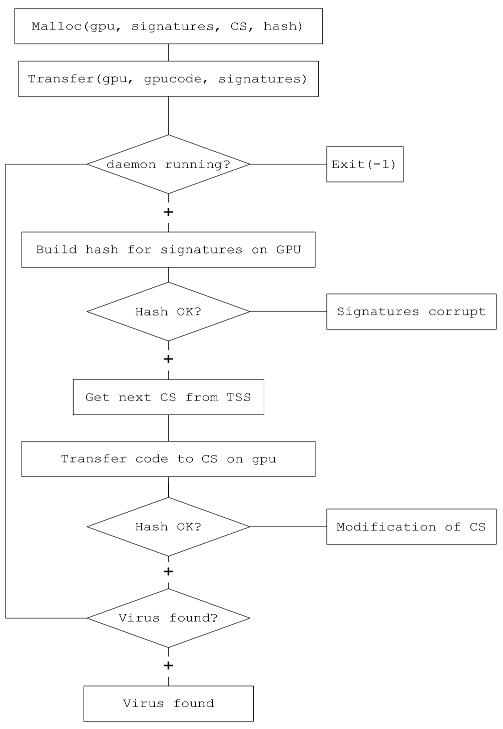


Figure 1 Basic function of GPU-based on-line virus scanning

On the GPU, we separate the code into n blocks of equal length and eventually pad it with zeroes. If larger, more transfers must be conducted. Thus, depending on the size of the code segment, there can be an arbitrary number of hash values, starting from n. For clarity, we show the initialization and execution phase of the algorithm in Listing 1. We are aware of the fact, that the signature database is verified each time. Instead, a process running at low priority could do the checking.


Malloc(gpu, signatures, CS, hash)
Transfer(gpu, gpucode, signatures)
daemon running?
Exit(-1)
+
Build hash for signatures on GPU
Hash OK?
Signatures corrupt
+
Get next CS from TSS
Transfer code to CS on gpu
Hash OK?
Modification of CS
+
Virus found?
+
Virus found

Listing 1 Initialization phase, on-line scanning and hash computation

3.1. Implementation

First, we take a look at an excerpt of the signature database from ClamAV [23] in Figure 2. We see that the input pattern can match from any offset, and that heuristics are allowed. The format is `<name>:<target-filetype>:<offset>:<signature>`. For simplicity, we do not regard the offset and regular expressions now.

```
Exploit.HTML.ObjectType:3:*:3c6f626a6563742074
7970653d222f2f2f2f2f2f2f2f2f2f2f2f
...
Email.Phishing.Webmail-
25:4:*:756e616c6c6f7965642073757070 6f727420697
320686967686c79206e6565646564203d{-
4}746f3d3230{-
10}73656375726520616e64207472616e7366657220736
f6d652066756e6473
```

Figure 2 ClamAV signature excerpt

The GPU maintains a version of the signature database which should be preprocessed in such a way that signatures can be easily found. E.g. the ClamAV [23] database (main.cvd, 20.3 MB compressed, 40 MB uncompressed) easily fits into the available GPU memory of the experimental system (2·768 MB). Signatures do not seem to be ordered in a special way. In contrary to [8], we first sort the signature database according to the length (ascending), the signature and the sum of the first 4 bytes *S* (ex. 1.2).

The first question was to answer how quick logic operations can be conducted on the graphics card. The workload consists of vector operations in dim(2^{24}) with different data types and operations, shown on the x-axis in Figure 3. The vector data is randomized in each run, logic operations were carried out on integers only and no deviation from the OpenMP-based multicore CPU implementation was detected. For the experimental setup, please see section 3.3. We additionally included floats on basic arithmetic operation to determine if the numerical algorithms could perform faster. We see that the CPU needs much more time for the execution of shift-like operations and testing for equivalence than the GPU. This clearly shows that pattern matching algorithms and hash functions can be efficiently computed on GPUs.

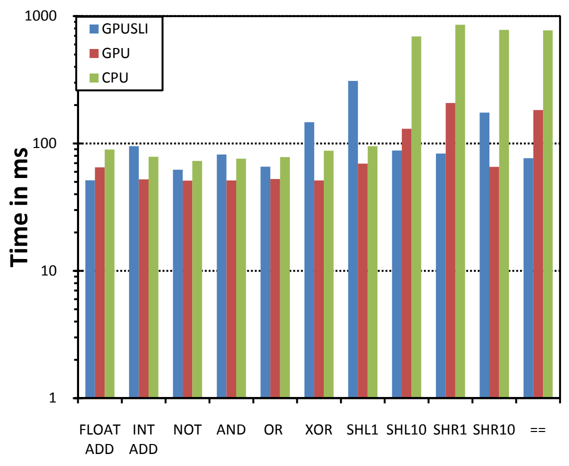


Figure 3 Execution time, logic functions, GPU/ CPU

We remark, that shifts and the tests for equality took much more time on the CPU. The results will change in each round due to the dynamic scheduling in both architectures. For best performance, we recommend a non-SLI system configuration.

3.2. Preparation and Search

We first take a look at the search string (in our case the signature) and look for symmetries, regularities. If we want to extract character probabilities p_i, we have to look at most of the characters in the search string. As this can be done in the preparation phase of the algorithm, we consider this for an implementation. The search text is split up in parts with the length of the first signature in the database, l_1. Thus, we get $t = t_1 \dots t_{l_1}, t_{l_1+1} \dots t_{2l_1,\dots,} t_{(n-1)l_1+1} \dots t_n$. Since we split the signature in different parts, we eventually have to pad the rest of t with zeroes. Now, we sum up the bytes in each section (ex. 1.1), resulting in $u = \left\lceil \frac{n}{l_1} \right\rceil$ summations. Note that the summation can be done in parallel and very efficiently on the GPU, since these are basic pixel operations. To omit an overflow, we suggest XOR. For clarity, we show an example (ex. 1.1 code, ex. 1.2 signature). S is the sum of the signature.

$$\begin{array}{l} \underbrace{0x00010203}_{\substack{6\\ [1]}}.\underbrace{0x23000000}_{\substack{35\\ [2]}}.\underbrace{0x1E243700}_{\substack{121\\ [3]}}. \\ \underbrace{0x12345678}_{\substack{276\\ [4]}}.\underbrace{0x3C6F626a}_{\substack{375\\ [5]}} \end{array} \qquad (1.1)$$

$$\underbrace{0x3C6F626a}_{S=375} \qquad (1.2)$$

We know that the signature is certainly not found in a section, iff $S > \sum_{i=n}^{n+l} t_i$. In our example, the length $l=l_1=4$. Thus, we can exclude sections 1 to 4. Example 1.1 is convenient, but usually we also have to regard both adjacent borders of two sections. Here, the signature cannot be contained, iff

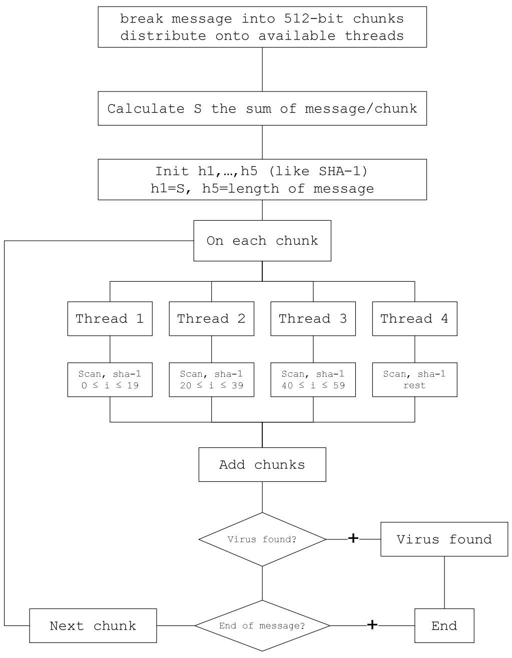
$$\begin{array}{l} \left(\left\lceil \frac{S}{2} \right\rceil > \sum_{i=n-l}^{n-1} t_i \wedge \left\lceil \frac{S}{2} \right\rceil > \sum_{i=n}^{n+l} t_i \right) \vee \\ \left(S > \sum_{i=n-l}^{n-1} t_i + \sum_{i=n}^{n+l} t_i \right) \end{array} . \qquad (1.3)$$

Therefore, we do not have to check sections 1 and 2 but 3, 4 and 5. We take a look at another example (ex. 1.4), where the search text is 0xAB (171), a single character. The methodology can be formulated recursively for different text sizes. Each entry on each level contains the sums from the two previous levels.

$$\underbrace{\begin{array}{l} \underbrace{\underbrace{\underbrace{AA \quad BA}_{\substack{356\\ [1]}} \quad \underbrace{CA \quad AA}_{\substack{372\\ [2]}}}_{728} \quad \underbrace{\underbrace{CC \quad AB}_{\substack{375\\ [3]}} \quad \underbrace{00 \quad 00}_{\substack{00\\ [4]}}}_{375}}_{1103} \\ \underbrace{\underbrace{\underbrace{00 \quad 01}_{\substack{1\\ [5]}} \quad \underbrace{02 \quad 00}_{\substack{2\\ [6]}}}_{3} \quad \underbrace{\underbrace{00 \quad AB}_{\substack{171\\ [7]}} \quad \underbrace{00 \quad BB}_{\substack{187\\ [8]}}}_{358}}_{359} \end{array}}_{1462}$$

(1.4)

Starting at the root level (bottom of ex. 1.4), we travel down the summation tree two steps (since 1452>171, 1103>171, 359>171) and can exclude sections 5 and 6. One step further, we have a match in section 7 and can exclude sections 4, 8. For the hash calculation, we focus our interest on the initial values $h_1,\dots,h_5$, e.g. of the 160 bit result of the SHA-1 [21] hash function. Note that this method can be applied to any hash function with initial values. The root level is used as part h_1 of the initial values. The length of the message is used as part h_5. The other constants can be the sums on each level of the summation tree, depending on the number of rounds and/or the coded path (0: left, 1: right) to regions where the signature is not contained (condition 1.3 fulfilled). E.g. section 4 is found by the path 011, starting at the top level. Since there can be many paths with different lengths, we have to define the right format, which is <length of path><path>, where the path length is a fixed number of bits. The other per-round integer additive constants are $y_1,\ldots,y_4$. We are aware that this is a far away from being cryptographically safe, but it can be carried out in parallel and speeds up the search. The computed hash value is stored in the dedicated hash memory on the graphics card, together with the address received from the TSS. We can support multiple entries, showing the development of a code/data section over time. For clarity, we depict the algorithm in Listing 2, based on the FIPS-180-1 pseudo-code [25]. We read 3 byte values resulting in a 3 byte index into a 2^{24}-entry array. This number of entries is needed to address at least every one of the 633992 (10/21/09) signatures from ClamAV. If we have a match, we compare the rest of the signature until we have a complete match or not.



Listing 2 Parallel on-line scanning, hash computation

3.3. Experimental Setup

Our experimental setup consists of a 6 GB main memory Core i7 system, configured with two NVidia GTX260 cards (PCIe 2.0 x16, non-SLI) and two hard disks (500 GB each, RAID 0). An SLI-system is constructed on hardware level and must be configured on software level. Either the GPUs work independently in non-SLI mode to support multi-view displays or all GPUs in a SLI configuration appear as a single unit. For the CUDA programming environment, a non-SLI system appears as a set of graphics cards, an SLI system as one graphics card. Multiple GPUs appear as multiple host threads. We applied the least aggressive clock settings (engine=500, shader=1150, memory=1900) MHz.

4 Conclusion and Future Work

In this work, we presented a first and novel approach to concurrently compute hash values for dedicated code sections and search for virus signatures. Today's processors are capable to protect code and data sections against modification. But changes from processes running on operating system level cannot be easily detected. The first experimental results show that the CPU needs much more time for the execution of shift-like operations and testing for equivalence than the GPU. This proved that pattern matching algorithms and hash functions can be efficiently computed with GPUs. The search algorithm is able to e.g. exclude regions containing a large number of zeroes etc. With the right ordering of signatures, all other signatures can be excluded with a single match. The simple hash value computations, based on summation are used to speed up the search. Our future work will consist of an implementation and experiments concerning throughput measurement. Furthermore, we will include regular expressions and the inclusion of offsets. Another part of future research is to find out if the initialization of the hash with text properties can help to detect concatenated extensions like $h(m)=h(m\cdot x)$, where m and x are messages of arbitrary length. An investigation should be carried out, if our method of parallel and independent hash computation leads to more collisions or not. To the opinion of the author this is elementary research, providing a starting point for independent hash functions which can be computed in parallel.

Literature

[1] A.V. Aho, M.J. Corasick, *Efficient string matching: An aid to bibliographic search*, Communications of the ACM, vol. 20, Session 10, pp. 761–772, 1977

[2] S. Wu and U. Manber, *A fast algorithm for multipattern searching*, Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994

[3] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, *Implementation of a content-scanning module for an internet firewall, IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31–38, 2003

[4] F. Yu, R. H. Katz, T. V. Lakshman, *Gigabit rate packet pattern-matching using TCAM*, ICNP, Oct. 5–8, 2004, pp. 174–183, 2004

[5] GPGPU, http://www.gpgpu.org, checked 10/15/09

[6] N. Jacob, C. Brodley, *Offloading IDS Computation to the GPU*, Computer Security Applications Conference (ACSAC), pp. 371-380, 2006

[7] F. Abazovic. *CUDA can speed up virus scan*, http://www.fudzilla.com/content/view/15832/1/, checked, 10/15/09

[8] E. Seamans, T. Alexander. *GPU Gems 3. Chapter 35. Fast Virus Signature Matching on the GPU.* http://http.developer.nvidia.com/GPUGems3/gpugems3_ch35.html, checked 10/15/09

[9] N. Huang, H. Hung, et al., *A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems*, pp.62-67, 22nd Int'l. Conference on Advanced Information Networking and Applications - Workshops (aina workshops 2008), 2008

[10] C. E. Spurgeon: *Ethernet. The Definitive Guide.* O'Reilly, Sebastopol, ISBN 1-56592-660-9, 2000

[11] J. D. Owens, D. Luebke, et al., *A Survey of General-Purpose Computation on Graphics Hardware*. Computer Graphics Forum, 26(1):80–113, 2007

[12] Karp, Richard M.; Rabin, Michael O. *Efficient randomized pattern-matching algorithms*. IBM Journal of Research and Development 31 (2), 249–260, March 1987

[13] R. S. Boyer; J. S. Moore: *A fast string searching algorithm*. CACM 20 (Issue 10): 762-772

[14] R. S. Boyer, J. S. Moore: *A Lemma Driven Automatic Theorem Prover for Recursive Function Theory*. IJCAI pp. 511-519, 1977

[15] R.N. Horspool: *Practical Fast Searching in Strings*. Software - Practice and Experience 10, pp. 501-506, 1980

[16] D.M. Sunday: *A Very Fast Substring Search Algorithm*. Communications of the ACM, 33, 8, pp. 132-142, 1990

[17] C. Charras, T. Lecroq, J.D. Pehoushek: *A Very Fast String Matching Algorithm for Small Alphabets and Long Patterns*. Proc. of the 9th Annual Symposium on Combinatorial Pattern Matching. LNCS 1448, Springer, 55-64, 1998

[18] http://www-igm.univ-mlv.fr/~lecroq/string/string.pdf, checked 10/10/09

[19] B. Commentz-Walter, *A string matching algorithm fast on the average*, Proc. 6th International Colloquium on Automata, Languages, and Programming, pp. 118-132, 1979

[20] R.A. Baeza-Yates, *Improved string searching*, Software — Practice and Experience 19, pp. 257-271, 1989

[21] A. J. Menezes, P.C. van Oorschot, S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, ISBN 0-8493-8523-7, pp. 321-383, 1996 (http://www.cacr.math.uwaterloo.ca/hac, checked 10/21/09)

[22] D.L. Cook, R.Baratto, A.D. Keromytis, *Remotely Keyed Cryptographics Secure Remote Display Access Using (Mostly) Untrusted Hardware*, In Proc. of ICICS, LNCS 3783, 2004

[23] http://www.clamav.net, checked 10/20/09

[24] R. Shonkwiler, *Computing the Hausdorff set distance in linear time for any Lp point distance.* Information Processing Letters, v. 38, pp. 201-207, 1991

[25] Federal Information Processing Standards Publication 180-1, *Secure Hash Standard*, 1995

[26] V. Korobitsin, S. Ilyin, *GOST-28147 Encryption Implementation on Graphics Processing Units*, pp.967-974, 3rd International Conference on Availability, Reliability and Security, 2008