Accelerating Malware Detection
via a
Graphics Processing Unit

THESIS

Nicholas S. Kovach, Civ

AFIT/GCO/ENG/10-12

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCO/ENG/10-12

Accelerating Malware Detection via a Graphics Processing Unit

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Nicholas S. Kovach, BSCS

Civ

September 2010

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCO/ENG/10-12

Accelerating Malware Detection via a Graphics Processing Unit

Nicholas S. Kovach, BSCS

Civ

Approved:

Barry Mullins — 26 Aug 10

Dr. Barry E. Mullins (Chairman) — date

26 AUG 2010

Dr. Michael R. Grimaila (Member) — date

26 AUG 2010

Dr. Gilbert L. Peterson (Member) — date

AFIT/GCO/ENG/10-12

Abstract

Real-time malware analysis requires processing large amounts of data storage to look for suspicious files. This is a time consuming process that (requires a large amount of processing power) often affecting other applications running on a personal computer. This research investigates the viability of using Graphic Processing Units (GPUs), present in many personal computers, to distribute the workload normally precessed by the standard Central Processing Unit (CPU).

Three experiments are conducted using an inductry standard GPU, the NVIDIA GeForce 9500 GT card. The goal of the first experiment is to find the optimal number of threads per block for calculating MD5 file hash. The goal of the second experiment is to find the optimal number of threads per block for searching an MD5 hash database for matches. In the third experiment, the size of the executable, executable type (benign or malicious), and processing hardware are varied in a full factorial experimental design. The experiment records if the file is benign or malicious and measure the time required to identify the executable. This information can be used to analyze the performance of GPU hardware against CPU hardware.

Experimental results show that a GPU can calculate a MD5 signature hash and scan a database of malicious signatures 82% faster then a CPU for files between 0 - 96 kB. If the file size is increased to 97 - 192 kB the GPU is 85% faster than the CPU. This demonstrates that the GPU can provide a greater performance increase over a CPU. These results could help achieve faster anti-malware products, faster network intrusion detection system response times, and faster firewall applications.

To my parents, who started me on the path of knowledge.

Acknowledgements

I would like to thank my advisor, Dr. Barry Mullins, for the support and encouragement to follow my ideas. I would also like to thank Dr. Michael Grimaila and Dr. Gilbert Peterson for their support and feedback.

I owe a special thank you to Ali and Teigan for being supportive and understanding throughout the course of this thesis effort.

Nicholas S. Kovach

Table of Contents

Page

List of Figures

Figure | Page

List of Tables

List of Abbreviations

ACCELERATING MALWARE DETECTION

VIA A

GRAPHICS PROCESSING UNIT

I. Introduction

1.1 Motivation

Everyday, data are created, collected, stored, searched, and replicated. As the amount of data grows, so does the time required to detect data that has been infected by malicious worms, viruses, trojans, spyware, and adware. Due to the large amount of time required to scan files and compare them to a database of known signatures, the user will experience a decrease in the responsiveness of their PC. As a result, they may disable the protection application such as Symantec Anti-virus [Vak10] or McAfee Anti-virus [McA09]. If the product is disabled, then the user is not protected against known malicious threats. Slow scanning times also mean that the malicious code, if executed, has more time to hide or infect other files in the system.

To help reduce the large amount of Central Processing Unit (CPU) resources anti-virus products need, the goal of this research is to offload part of the scanning and searching for signature matches to a mainstream Graphics Processing Unit (GPU). Most applications do not take advantage of GPUs for non-graphical tasks, even though they are openly available for all newer computers [NVI09b] and are often not fully utilized by the average computer user [ViG07]. The system developed in this research is designed to use the unused power of the GPU by reducing CPU resource demand and increase system security by allowing the file scanning to complete without the user noticing. Because only one video card driver can be loaded by Windows XP, the GPU was still responsible for displaying graphics on a terminal, but the monitor was turned off during the experiments to minimize the impact of graphical display on the results. If the graphical display is modified, such as changing the resolution, then memory on the GPU could be modified to support the display and cause any application running on the GPU to return an error.

GPUs at one time were only available to handle graphics. Over time they have evolved into a general purpose GPU, allowing code to be written and directly executed on the GPU. This allows applications to directly use the GPU to offload computational tasks without consuming resources of the CPU.

1.2 Overview and Goals

This research focuses on the design and analysis of a malware detection tool, called Graphic Processing Unit IDentifier (GPU ID), that uses the parallel power of the GPU to scan files by calculating a MD5 file hash and then searching a database of signatures from malicious files. The GPU ID system is designed to be used on a personal computer (PC) but may be expanded to gateway monitoring systems. For each file, the GPU ID system calculates a MD5 file hash and then searches the malware signature database. If the hash is in the database then the file is considered malicious, otherwise the file is considered benign. The calculated MD5 hashes are never transfered back to the CPU from the GPU device. Instead a set of flags indicating the malicious status of each file is transfered to the CPU and the user is alerted to files that match a database entry.

There are three goals for this research. The first goal is to find the optimal number of threads per block for calculating MD5 file hashes. To accomplish this goal a GeForce 9500 GT GPU is used to calculate MD5 file hashes, while the number of threads per block is varied. The second goal is to find the optimal number of threads per block for searching a MD5 signature database for hash matches. To accomplish this goal the Clam AV [Cla09a] MD5 signature database is used and modified, and a GeForce 9500 GT GPU is used to calculate MD5 file hashes and search the signature database, while the number of threads per block is varied only for the search part of the program. The third goal is to measure the performance of a GPU while detecting malware. To accomplish this goal the time to calculate MD5 file hashes and search the signature database are measured for groups of files and then compared to the times required for a CPU to complete the same task.

1.3 Thesis Layout

This chapter introduces the research topic, provides the motivation, and outlines the goals of the research. Chapter 2 provides background information on Portable Executable (PE) Files, static malware detection, the MD5 algorithm, CUDA GPU basics, and the GeForce 9500 GT GPU. The methodology used to develop, set up, configure, and conduct the experiment to test the performance of the GPU is outlined in Chapter 3. The experimental results are presented and analysis in Chapter 4. Chapter 5 provides a discussion of the conclusions drawn from the experimental results, the significance of the GPU ID system, and possible areas for future research. Appendix VI contains the raw data collected during the experiment.

II. Literature Review and Related Research

This chapter describes the background and related work for detecting malware with a GPGPU, referred to hereafter as GPU. Background is provided in Sections 2.1 through 2.7. Sections 2.1 through 2.3 provide background on PE files, static malware detection, and MD5 fingerprinting. Section 2.4 provides a detailed overview of the Intel Pentium Architecture, and Section 2.5 provides an overview of the PCI Express 2.0 I/O bus architecture. The NVIDIA GPU and CUDA architectures are discussed in Section 2.6, followed by Section 2.7 with an overview of Clam AV anti-virus components. Section 2.8 discusses related work with GPU malware detection.

2.1 Portable Executable Files

The Portable Executable (PE) file format is designed for use on all Microsoft Win32 operating systems. The format defines the structure of the executable file data and how the file data is interpreted. The PE file format is expected to remain part of Microsoft's operating systems for the future [Szo05]. The PE format is an updated version of the common object file format (COFF) [Mic06]. Microsoft released a new format PE+, or PE32+, for use on Win64 operating systems with the release of Windows XP 64-bit [Mic08]. The PE+ format is similar to the PE format except for modification to support 64-bit operating systems.

As shown in Figure 2.1, a PE file is composed of many components. The first component is an MS-DOS header and stub program. The stub program displays an error message, "This program cannot be run in MS-DOS mode" [Pie94]. The stub program provides compatibility for 16-bit Windows systems by not allowing the file to be executed in DOS [Szo05]. The second component, after the MS-DOS header and stub program, is the PE header, which starts with the constant of 'PE00' [Pie02]. The PE header contains information about the intended type of CPU, number of sections, characteristics, size of image, and the checksum of the PE file.

Between the headers and raw data of the sections is the section table. The section table contains a header for each section in the PE file. The section header

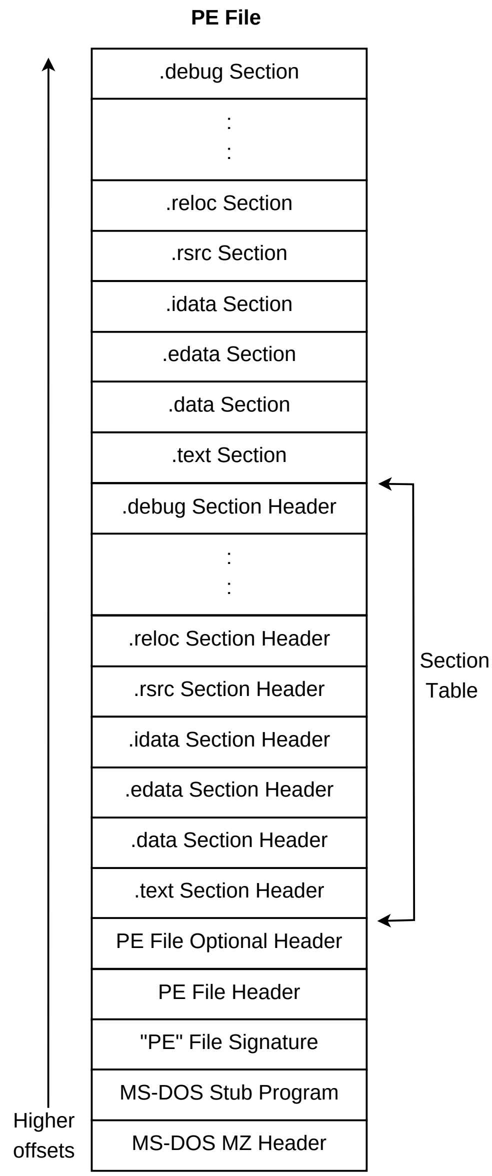


Figure 2.1: Overview of the PE File Format Structure [Pie94] [Pie02] [Szo05].

contains the name, size, address information and attributes for the section. The Microsoft Windows' memory manger will use the information in the section header to determine if the section is readable, writable, or executable [Eil05].

Common PE file sections include: .text, .data, .bss, .rsrc, .idata, .edata, .reloc, and .debug. The .text section contains the actual executable code and is normally the first section in a PE file [Szo05]. The PE file format is designed to allow executable code to be separated from data. The executable flag is set on the .text section, but the writable flag is not set because data is kept in the .data section, so there is no need to write to the .text section. This helps to keep the running program from overwriting code instructions. Data is stored in the .data section and the .bss section. The .data section contains initialized data, while the .bss section contains uninitialized static and global variables. Resources, such as images, menus, default initialization strings, etc., for the application are stored in the .rsrc section. The import table, containing a list of functions used from external libraries, is located in the .idata section, and functions exported for use by other applications are located in the .edata section. The PE format defines a .reloc section containing a base relocation table; this section has been removed from Windows 9x and later operating systems by Microsoft [Szo05]. Any debug information about the executable is located in the .debug section. This information is optional and may not be present in all PE executables because including it will increase the size of the executable.

The structure of a PE file loaded into memory looks similar to the PE file on a disk [Szo05]. Figure 2.2 shows the structure of a PE file mapped into memory. The headers and section layout remain the same, but the individual sections are page-aligned in memory. This allows the OS to assign different access permissions to the resulting pages. Sections are not page aligned on disk to avoid wasting disk space [Pie02]. When a PE file is compiled, all addresses are compiled to a fixed base memory address. The OS will try to load the PE file to this memory address, but if the address is not available the OS will choose another address. To avoid having fixed memory addresses in PE files that need to be updated if the OS cannot load the file into the fixed base memory address, Relative Virtual Addresses (RVA) are used. A RVA is just an offset in memory, which when added to the address where the PE file was actually loaded by the OS, gives the actual memory address needed by the executable code in the PE file [Pie02].

Function calls to Dynamically Loaded Libraries (DLL) are handled by the Import Address Table (IAT). The IAT contains a list of all functions (symbols) and the respective memory address for the function being imported by the application. When

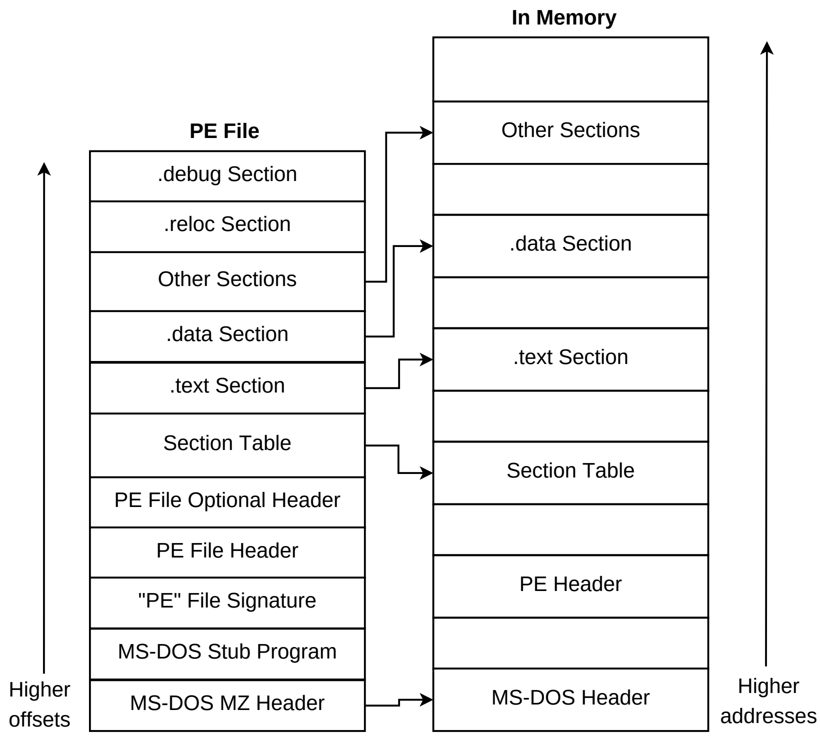


Figure 2.2: Overview of a PE File in Memory [Pie02].

the PE file is loaded into memory, the memory addresses are overwritten with the actual memory addresses of the symbols by the system loader [Mic08]. This memory address represents the address that is invoked when a call is made to imported functions [Pie02]. Since the PE file is mapped into linear address space, the application only knows the base address of where the executable was mapped into memory. By using the IAT, only the IAT has to be updated instead of the individual function calls within the executable [Szo05].

2.2 Static Malware Detection

Signature-based detection methods of malware have long been used by commercial anti-virus software. This type of detection method has been used since the late 1980's with only optimizations and improvements in algorithms since then [For04]. Commercial anti-virus software is commonly used to protect home and business computing systems from malware or unwanted programs. Generally, signature detection involves the inspection of files (usually executables) on digital storage mediums for predefined signatures [Kel09]. Recently, other file formats such as DOC, PPT, XLS, and PDF have been used to carry malware and are also inspected by commercial anti-virus software [MIT07] [MIT09b] [MIT09a].

Signatures are generated based on the composition or attribute(s) of a particular piece of malware, so the signatures are unique to that piece of malware [For04]. Signatures are generated based on either the whole file or individual code strings of the file, which signify malware behavior by applying a hashing algorithm to the file or individual sections of the file [Hey07]. In the case of PE files, the sections are identified by the information in the section table of the PE file. The predefined signature is then compared to live signatures generated by the anti-virus software tool, using the same hashing algorithm in real time. If there is a match, then file execution access on the intended machine is blocked, the file is deleted, or the user is alerted [Hey07]. This process is known as black listing.

Black listing may be reversed for trusted files in a process known as white listing. The signatures are still generated based on the file or individual code strings of the file, but if the on-the-fly and predefined signatures match, then file execution access is granted to the intended machine, otherwise the file execution is blocked [McA09]. White listing provides more protection than black listing, but decreases usability of the intended machine because the user no longer chooses which applications to trust. Another version of white listing involves signing the executable and then allowing only executables digitally signed by a trusted party to be executed. This technique is used in Microsoft Windows Operating Systems (XP, Vista, and Windows 7) to verify certified system drivers [Mic07a].

Malware may use a combination of methods to hide itself from signature-based detection software. Such methods include: altering the source code, using a packer, obfuscation, and editing the executable code [Kel09]. Each time one of these methods is used by the malware, a new signature must be generated and installed in the

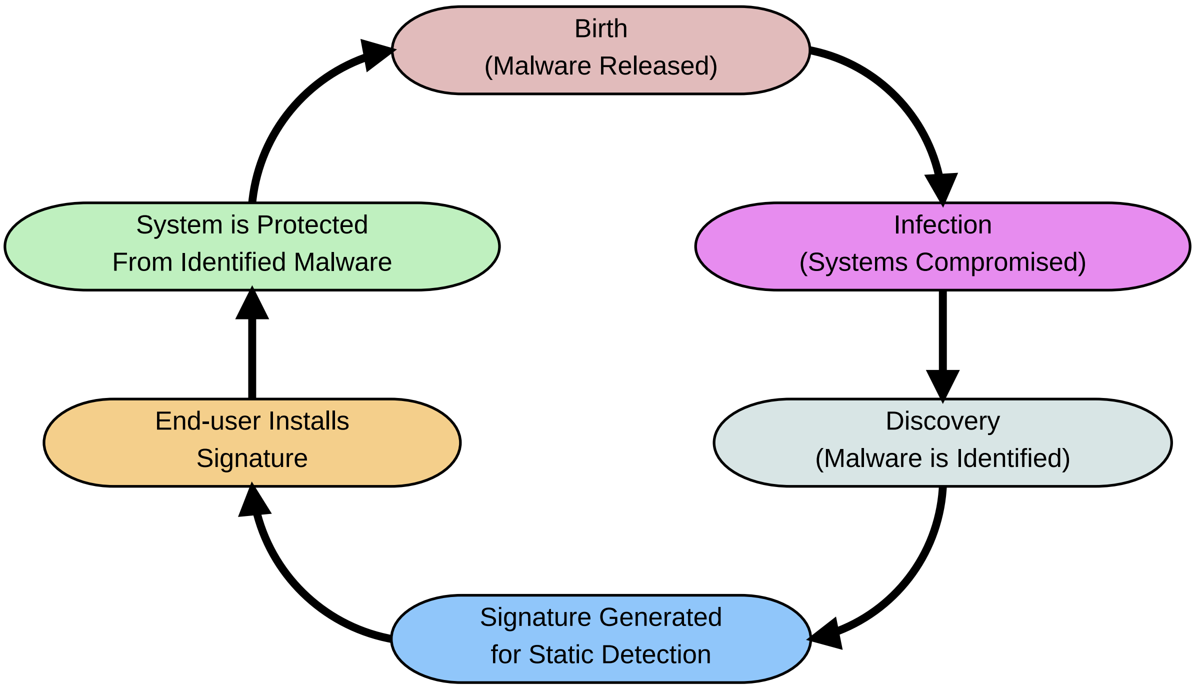


Figure 2.3: Malware Protection Process.

signature-based detection software, requiring interaction by the user or the system to be connected to a network to access the update server to automatically install the new signatures. This is in addition to the time required to discover the modified malware and generate a new signature. Figure 2.3 shows the process of protecting a system with static detection. Once the malware is released in the wild, it must infect, or compromise, vulnerable systems. A compromised system or Honeypot then discovers the malware and submits it for analysis. After analysis, a signature is generated. This signature must be installed by the user before the system is protected.

Hash algorithms, such as MD5 [Riv92], are often used to create signatures of malware [Cla09b]. The hash algorithm produces a shorter representation of the file or file attributes into a fixed length fingerprint. The fixed length fingerprint is then used as the signature. In order to be used in fingerprinting the hash algorithm is required to produce large changes in the hash result for small changes in the file or file attributes. Using hashes to fingerprint files is not always infallible.

False positives occur when a file is identified as malicious when it really contains benign code [Vak10]. Anti-virus scanners, using static detection techniques, may give a large amount of false positive alerts [NAs02]. These alerts can be costly in terms of time and resources for individuals and organizations to investigate each misidentified file [YWL07] [Vak10]. False positives are possible, since the hashes used as fingerprints are a fixed length and the number of possible strings is infinite. According to the Pigeon Hole Principle, because the number of fingerprints is less than the number of possible strings, multiple strings will be represented by the same fingerprint. False positives can be reduced by using specific signatures [Szo05], such as generating the fingerprint by calculating the file hash of the malicious file. This would reduce the number of false positives, but may increase the number of false negatives (discussed later), in the case where the malicious file varies slightly from one instance to the next [Pau08]. A recent example of a false positive is when a signature in a McAfee anti-virus product identified the core Windows XP binary svchost.exe as a virus crippling the operating system [McA10].

False negatives occur when a file is identified as benign when it really contains malicious code [Pau08]. This happens when a signature is missing from the virus database. This is possible for new malware or in cases where the database is outdated (i.e., the user does not regularly update the database to learn about new viruses). In order for static detection to be useful the malware must first be analyzed, a signature generated, and then the signature must be added to the users database. Here, the initial detection of the malware is required for the signature to be generated. Without the initial detection, anti-virus protection would be difficult or impossible [Coh86] [Coh87]. False negatives can be reduced by using generic signatures [Szo05]. A generic signature may be generated by basing the hash fingerprint on several malicious attributes shared by similar malicious software, if these attributes are found when scanning then there is a chance the file is malicious. Generic signatures have the side-effect of increasing false positives.

Using a combination of false positive and false negative reduction techniques lowers the chances of unwanted alerts (false positives) and infections (false negatives) [NAs02]. In addition, white listing of critical system files reduces the chance of one being identified as malicious.

2.3 MD5

The MD5 message digest algorithm was developed by Ronald Rivest in 1992 [Riv92]. It was developed for applications where a sequence of bytes, message, file, or other data must be represented by a small fixed length identifier. MD5 takes in a piece of data, of an arbitrary length, and outputs a 128-bit message digest. The algorithm is designed to be: easy to compute the digest; hard to compute the message from the digest; and hard to find two messages with the same digest [StL07]. Although it is known that many attacks exist on the MD5 algorithm to produce collisions [XiH05] [YJD09] or two messages with the same digest, it still provides a useful method for fingerprinting a sequence of bytes or files.

The MD5 algorithm starts by padding the raw data until its length is congruent to 448, modulo 512. A single '1' bit followed by enough '0' bits are used in the padding. At least one bit, is appended and at most 512 bits are appended to the raw data. Next, the length of the data before padding is appended to the end of the padded result. The length is represented as two bytes with the lower order byte added first. If the length of the data exceeds 2^{64}, then only the low-order 64 bits of the length are appended. Four 32-bit registers are initialized to the following constant initialization values in hexadecimal with low-order bytes first [Riv92]:

$$
\begin{array}{lcllll}
\mathrm{GA} & = & 01 & 23 & 45 & 67 \\
\mathrm{GB} & = & 89 & \mathrm{ab} & \mathrm{cd} & \mathrm{ef} \\
\mathrm{GC} & = & \mathrm{fe} & \mathrm{dc} & \mathrm{ba} & 98 \\
\mathrm{GD} & = & 76 & 54 & 32 & 10
\end{array}
$$

Four functions map three of the 32-bit registers to one 32-bit register. The functions are as follows [Riv92]:

$$F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$$
$$G(B, C, D) = (B \wedge D) \vee C \neg D$$
$$H(B, C, D) = B \otimes C \otimes D$$
$$I(B, C, D) = C \otimes (B \vee \neg D)$$

The data, message (M), is processed by the MD5 algorithm in 512-bit (64-byte) chunks. One MD5 operation is completed for each byte. An MD5 operation is shown in Figure 2.4 and starts with the local registers A, B, C, and D being initialized with the values from the global registers GA, GB, GC, and GD. For each byte (represented by [i]) of the 64 bytes in the chunk, a function from above is selected during each operation. For bytes 0 -15 function F is used, bytes 16 - 31 function G, bytes 32 - 47 function H, and bytes 48-63 function I. Each function takes registers B, C, and D as inputs. A fixed constant K is added to the byte from the message, the constant for each byte in a chunk is listed in RFC 1321 [Riv92]. A left shift (<<s) is also applied; the amount of the shifts are listed in RFC 1321 as well. The registers are then updated as follows:

$$\text{temp (register)} = D$$
$$D = C$$
$$C = B$$
$$B = B + (A + \text{function}(B,C,D) + k[i] + M[i]) << s[i]$$
$$A = \text{temp}$$

After all 64 bytes have been processed, the results in A, B, C, D and are added to the results from previous 64-byte chunks and stored in registers GA, GB, GC, GD (i.e., GA = GA + A, GB = GB + B, etc.). The message is processed in this manner until there are no more chunks left. The MD5 digest output is from the registers

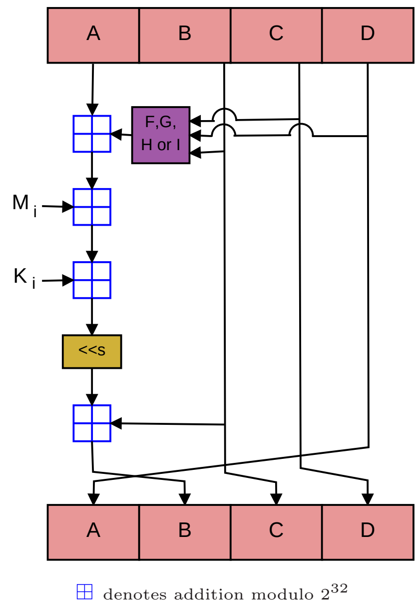


Figure 2.4: MD5 Operation [Riv92].

GA, GB, GC, GD in alphabetical order. The digest output is often converted to a 32 character ASCII hexadecimal value for readability. The small 32 character ASCII value represents a large file, making MD5 a good algorithm for fingerprinting files in malware detection.

2.4 Intel Pentium 4 (CPU)

The Intel Pentium 4 processor, or central processing unit (CPU), is manufactured using Intel's 90nm process supporting speeds of 2.40 - 2.80 GHz [Int05]. The processor has 16 KB of Level 1 (L1) data cache and 1 MB of Level 2 (L2) cache. The processor has a front side bus of 800 MHz, with support for Streaming SIMD Extensions 2 (SSE2) and Streaming SIMD Extensions 3(SSE3). SSE2 defines hardware instructions for 64-bit floating point operations [Int00], while SSE3 defines hardware instructions for thread management [Int08].

The Pentium 4 supports Hyper-Threading (HT) technology which allows a single physical processor to function as two logical processors [Int05]. Each logical processor has its own control registers, while sharing caches, execution units, and buses.

HT technology is designed to use processor resources more efficiently and improve performance of multi-threaded software [Int10a]. To use HT on the Pentium 4, a HT-enabled BIOS and operating system such as Microsoft Windows XP or newer is required.

The Pentium 4 was selected for this research because of its availability and ability to run common operating systems, such as Windows XP [Mic01], Vista [Mic07b], Windows 7 [Mic10], and many distributions of Linux [Ubu10] [Dam10] [Pup09]. The Hyper-Threading technology is enabled on the Pentium 4 to use system resources more efficiently.

2.5 PCI Express 2.0

The latest NVIDIA GPUs, including the GeForce 9500 GT, connect to the motherboard through a PCI Express 2.0 (PCIe) bus. PCIe is a third generation high performance I/O bus designed for high bandwidth peripherals (end points), such as video controllers, memory, and disk drives [BAS04]. The bus is implemented as a serial point-to-point architecture allowing communication between two PCIe devices [BAS04]. PCIe supports data rates of 128 Gbit/sec [Int10b].

The PCIe fabric is comprised of a root complex, any number of switches, and any number of endpoints. The root complex connects CPUs and memory subsystem to the PCIe fabric. PCIe switches forward packets between endpoints and the root complex. Endpoints are devices that complete PCIe transactions (transmission and reception of requests), but are not the root complex or switches.

The root complex controls and routes high-throughput bus packet traffic between endpoints [BAS04]. The root complex also transports PCIe packets from endpoints to the memory controller for direct memory access (DMA) operations. As shown in Figure 2.5, processors connect to the root complex through the front side bus. High performance peripherals such as video cards and main memory controllers connect directly to the root complex [BAS04]. Other peripherals and PCIe expansion

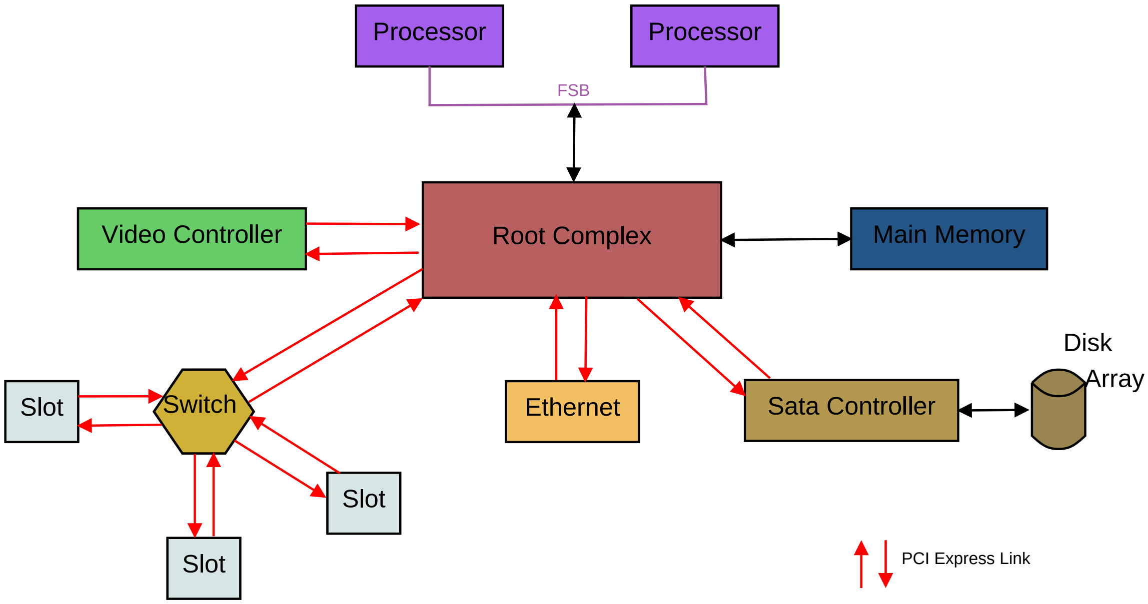


Figure 2.5: PCI Express in a Hypothetical System.

slots are connected with the system through PCIe switches [BAS04]. The switches are responsible for routing commands and data packets between various peripherals and the root complex.

Communication on the PCIe bus takes place with the transmission and reception (transaction) of transaction layer packets (TLPs). There are two types of transactions: non-posted and posted. In non-posted transactions a TLP request packet is sent to an endpoint, after the endpoint receives the request packet, a TLP completion packet is sent back to the original endpoint [BAS04]. The TLP completion packet confirms the request TLP was received. Read transactions contain the requested data in the completion TLP, while write transactions contain data in the request TLP [BAS04]. In posted transactions, a TLP request packet is sent to an endpoint, while no completion packets are sent back [BAS04]. Posted transactions are optimized for performance in quick transaction completion, at the expense of the requesting endpoint not knowing if the request was completed successfully [BAS04]. Request TLPs may contain data in posted transactions, but it is not required.

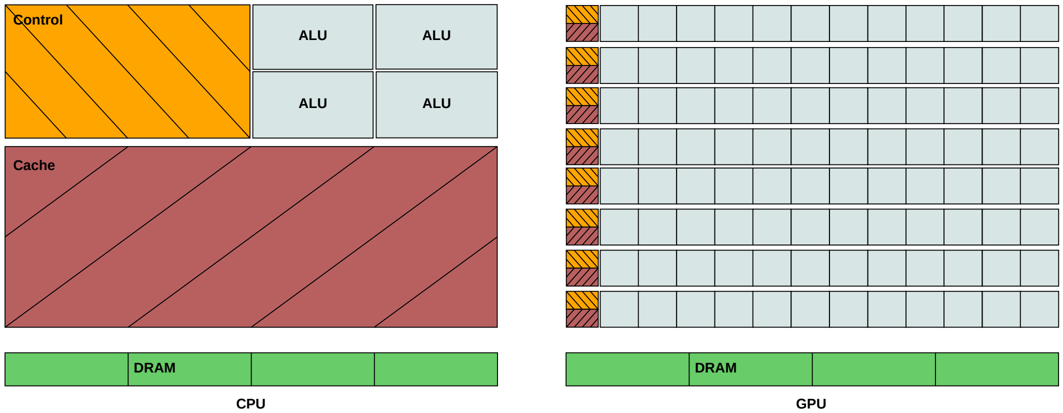


Figure 2.6: CPU and GPU similarities [NVI09b].

Each byte of data is converted into a 10 bit code (8b/10b encoding). 8b/10b encoding gives the PCIe bus greater robustness by allowing AC coupling of the differential pairs of signals (a transmit pair and a receive pair) and an embedded clock rate that improves as silicon technology is refined [PCI10]. The encoding scheme creates 25% additional overhead. PCIe 3.0 is expected to use a 128b/130b encoding scheme [PCI10]. This will reduce the overhead to about 1.6%. The expected overhead will allow higher bandwidths, decreasing the delay of memory reads and writes in global memory while increasing GPU performance.

2.6 Graphical Processing Unit (GPU)

The GPU is similar to a CPU, but is designed to handle streaming data [NVI09b]. As shown in Figure 2.6, a GPU devotes more transistors to data processing, whereas a CPU devotes more to data caching and flow control [NVI09b]. Since streaming data is already sequential, or cache-coherent, the GPU does not need a large amount of cache. This gives the GPU an advantage in highly arithmetic-intense parallel computations, where the number of arithmetic operations are far greater than memory operations. Arithmetic calculations hide memory latency on a GPU instead of data caches hiding memory latency on a CPU [NVI09b]. This means multi-threading is used to keep the GPU busy between costly memory accesses instead of fast data caches like a CPU.

Previous GPU architectures were based on a single instruction multiple data (SIMD) programming model, but recent GPU architectures, including CUDA (discussed later) [NVI09b], are based on a single instruction, multiple thread (SIMT) programming model. In SIMT, hardware multithreading leverages thread-level parallelism. SIMT is similar to single instruction, multiple data (SIMD) except programmers have the ability to write code for coordinated threads and independent threads. This is referred to as a single program, multiple data (SPMD) programming model; which is a subset of the multiple instructions, multiple data (MIMD) programming model. SPMD consists of multiple SIMT multiprocessors running the same program, but each multiprocessor may execute a different instruction [HTA08]. In addition, each multiprocessor may have many threads, each operating on different data [NVI09b].

The NVIDIA GPU contains multi-threaded Streaming Multiprocessors (SMs) [NVI09b]. The number of SMs varies by version of the GPU; the GeForce 9500GT from NVIDIA (discussed later) contains four SMs. Individually a multiprocessor executes one instruction at a time, but each thread may operate on different data or choose to idle while other threads execute the instruction. This means the multiprocessor follows the SIMD programming model. Since each multiprocessor may execute a different instruction, the GPU as a whole follows the SPMD programming model.

2.6.1 NVIDIA GPU Basics. CUDA (discussed later) allows functions, called kernels, to be defined. A kernel is the entry point for the code to be executed on the GPU. On the GPU the kernel is executed by a grid of equally sized thread blocks [NVI09b]. Figure 2.7 shows the CUDA object abstractions, where a grid is made up of thread blocks and thread blocks are made up of multiple CUDA threads. A grid is a group of blocks with no synchronization between individual blocks. There is only one grid per kernel, allowing only one kernel to be executed at a time. Each kernel may be executed by many lightweight CUDA threads.

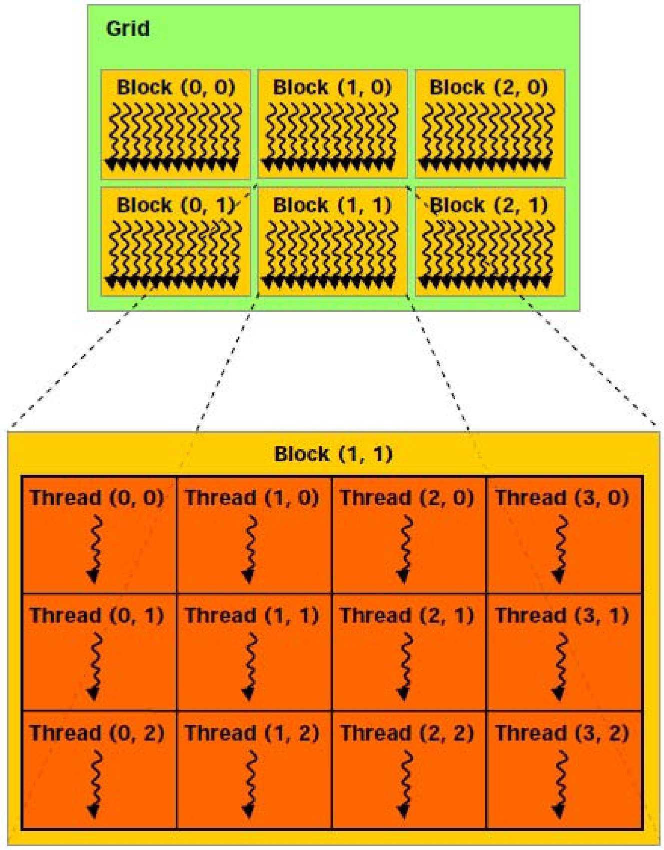


Figure 2.7: CUDA Grid, Thread Blocks, and Threads [NVI09b].

CUDA assumes all threads execute on a separate secondary device, such as a GPU, from the central processor; with the secondary device operating as a coprocessor to the central processor [NVI09b]. All threads created with CUDA are lightweight, with little creation overhead and fast context switching. Threads are created, managed, scheduled, and executed as a group of 32 parallel threads called a warp. Each individual thread of a warp will start with the same program address, but will have its own instruction address counter and register state [NVI09b]. A thread will execute on a single multiprocessor and will not migrate to another after it has been created. Every thread has access to global, shared, local, texture and constant memory. Threads will also have registers (8192 registers divided equally by all threads in a block).

Conditional branching statements should be avoided within a thread context because all threads walk through each of the possible execution paths caused by

```
__global__ void function(int4* x) {

    if(threadIdx.x >= 4) {

        // Code Section 1
    } else {

        // Code Section 2
    }
}
```

Figure 2.8: CUDA Example Code: Thread Divergence.

conditional branching. For example, the code in Figure 2.8 shows an example CUDA GPU kernel. The variable *threadIdx.x* refers to the thread ID and is provided by the CUDA runtime. If the conditional fails for some threads but not all, then all threads will walk through code section 1, with the failing threads idling. After code section 1 finishes, code section 2 is executed with previously idle threads executing and previously executing threads idling. Maximum efficiency is reached when all threads in a warp agree on the execution path [NVI09b].

Thread blocks are blocks of CUDA threads running the same kernel. Each block can contain 512 threads due to memory limits. The number of threads per block should be a multiple of the warp size to maximize performance [NVI09b]. Each thread block is required to execute independently of other thread blocks and must be able to execute in series or parallel with other blocks. Thread blocks execute independently to allow for scalability; a GPU with more cores can execute a program faster than a GPU with fewer cores [NVI09b]. Because of the independence of thread blocks, conditional statements may be used within the context of a thread block with no performance impact.

Several thread blocks reside concurrently on one multiprocessor, limited only by the amount of registers and shared memory available on the multiprocessor. The registers are partitioned among all threads in a block equally and shared memory is partitioned among all thread blocks on the multiprocessor [NVI09b]. Threads in a block may share data and coordinate through shared memory, while threads from different blocks may not share data or coordinate. The CUDA architecture assumes all thread blocks run to completion without pre-emption.

A CUDA program should create as many thread blocks as multiprocessors on the device. This allows each multiprocessor to have a task (a kernel to execute). It is possible to execute fewer thread blocks than multiprocessors but doing so reduces performance. If there is only one block per multiprocessor, the multiprocessor may be forced to idle during thread synchronization and device memory reads [NVI09b]. Therefore it is more efficient to have as many thread blocks as possible allowing the GPU hardware to efficiently manage thread synchronization and device memory reads/writes.

Memory space available to a GPU includes global, local, shared, constant, and texture memory. The host and device are responsible for managing their own memory spaces in DRAM. Table 2.1 shows the characteristics of the available memory under CUDA 1.1, while Figure 2.9 shows a graphical representation of the memory visibility in relation to grids, thread blocks, and threads. The global, constant, and texture memory are persisted across kernel launches by the same application and can be accessed by all active threads on the GPU as well as the host CPU, because each is located off the GPU chip. Texture and constant memory are the only memory spaces cached on a GPU, but can only be read by the GPU with no write access allowed [NVI09b]. A multiprocessor takes four clock cycles to issue one memory instruction for a warp when accessing global or local memory [NVI09b]. Each type of memory is discussed in greater detail in the following paragraphs. Additional memory is available on chip through shared memory and registers. Shared memory can be accessed by all threads in the same thread block, while registers may only be accessed by the thread the register was assigned to when the kernel was launched.

Global memory is accessible by all active threads and the host CPU. The data lifetime (the period of time data remains in memory) of the global memory is from

Table 2.1: CUDA Memory Characteristics [NVI09c] [NVI09b].

Memory	Location	Cached	Access	Visibility
Registers	on chip	Resident	Read/Write	single thread
Global	off chip	No	Read/Write	All threads and host CPU
Shared	on chip	Resident	Read/Write	All threads in a single block
Local	off chip	No	Read/Write	Single thread
Texture	off chip	Yes	Read	All threads and host CPU
Constant	off chip	Yes	Read	All threads and host CPU

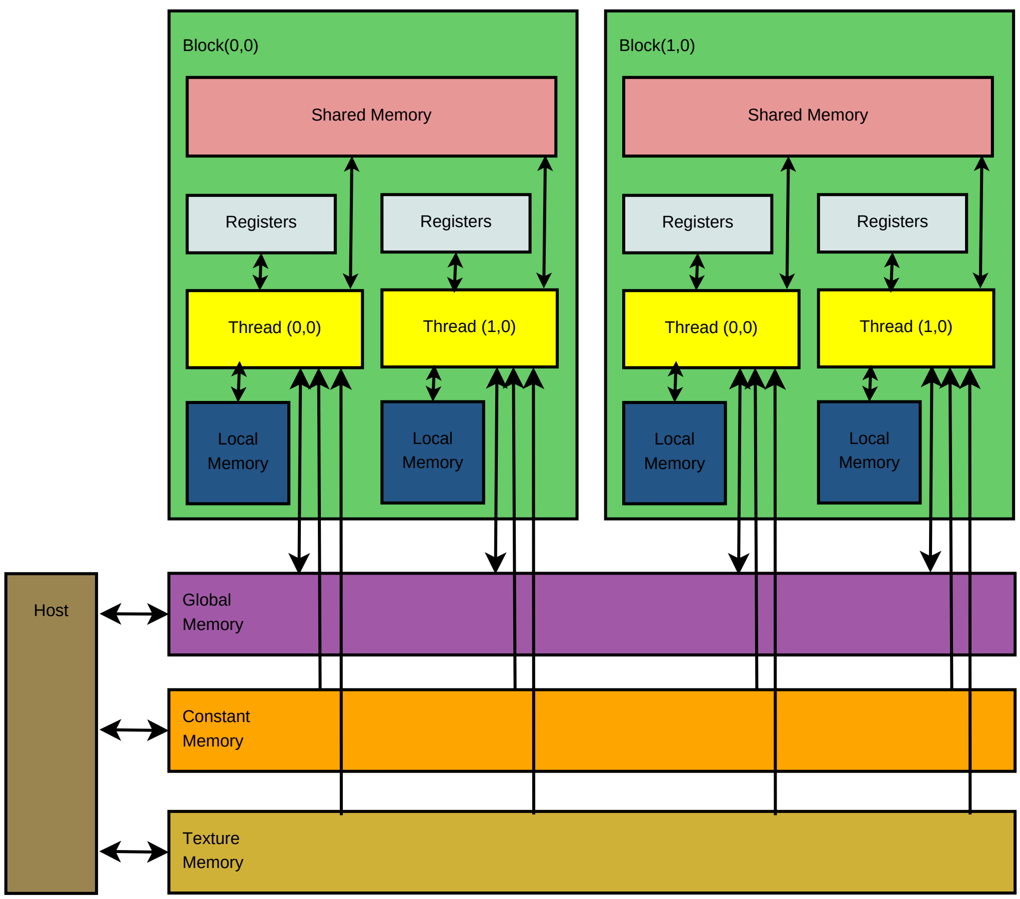


Figure 2.9: Overview of Visible Memory under CUDA [NVI09b].

allocation to deallocation. Memory accesses are not cached, reducing the performance of the GPU for each access. A global memory request for a warp of threads is split into two memory requests, one for the first 16 threads and one for the last 16 threads. Global memory bandwidth is most efficiently used when memory accesses by a thread half-warp are combined into a single memory transaction maximizing PCIe bandwidth [NVI09b]. A single instruction can fetch 32, 64, or 128-bit words into registers from global memory [NVI09b].

Local memory is located off the multiprocessor chip in DRAM and cannot be accessed by the host. The memory retains data for the lifetime of the device thread, since the local memory is per thread. The cost of accessing local memory is as expensive as accessing global memory because local memory is not cached. This means local memory should be used sparingly. Local memory is similar to global memory except a single thread is the only one allowed to modify the data. This ensures data integrity for the thread's individual data.

Shared memory is on chip and assigned per thread block. The data lifetime of shared memory is equal to the life of the block. It is divided into equally-sized memory banks, with different banks being accessed simultaneously [NVI09b]. This allows the maximum number of serviceable simultaneous memory requests to be the same as the number of memory addresses falling in to unique memory banks. If memory bank conflicts are avoided, then memory accesses can be as fast as registers. Caution must be used since multiple threads can access the same data; all threads must be synchronized after a write operation. The CUDA architecture includes a hardware synchronization instruction that idles a thread when it is executed. After all threads have executed the same synchronization instruction, all threads resume execution at the next instruction. All threads must execute the synchronization instruction before further execution is allowed. If all threads are not guaranteed to execute the synchronization instruction, then the *NVCC* compiler driver will return an error when compiling the source CUDA code. Synchronizing a thread after a memory write operation guarantees every thread sees the same data in memory.

Registers are provided by thread block and are evenly divided between all threads in a thread block. The life of the data in the register is equal to the life of the thread assigned to the register. A register access takes zero clock cycles per instruction, making it the fastest form of memory [NVI09b]. If there are not enough registers for a thread, some data may be placed in local memory. This will result in slow performance due to the high latency cost of accessing local memory. If there are not enough registers and not enough local memory available for register data, then the kernel execution will fail and an error code will be returned from the GPU.

Texture and constant memory are only readable by the device. Texture memory holds an object for reading data, and the data is cached. The host code binds data to a texture object and the kernel reads the data by fetching it from memory via a function on the texture object. A texture is optimized for 2D spatial locality, so maximum efficiency is reached when threads read texture addresses that are close together [NVI09b]. Textures are better at hiding latency of addressing calculations because they are designed for streaming fetches with a constant latency. In a texture, each cache hit reduces demand for the DRAM bandwidth, while fetch latency remains the same [NVI09b]. Constant memory is cached and is designed to hold data required by every thread. It can only be written to by the host and remains constant once the kernel starts to execute. When all threads in a warp read from the same address the access is as fast as a register, but when threads read multiple locations each access will be serialized. Pre-fetching of data will often eliminate cache misses on first constant memory access, since when there is a cache hit there is only one cycle of latency even though constant memory is in DRAM [NVI09b].

2.6.2 CUDA by NVIDIA. NVIDIA released the Compute Unified Device Architecture (CUDA) in November of 2006 to provide developers with a general purpose computing architecture that leverages the parallel compute engine in NVIDIA GPGPUs. It facilitates the heterogeneous computing of CPU and GPU environments by allowing the code executing on the host (CPU) to link, load, and start the code intended for execution on the device (GPU). CUDA provides a software development environment that allows developers to use C/C++ as the high-level programming language for programming GPUs and predefined data structures and methods that build upon the C/C++ programming languages to aid in parallel development through extensions to the C language [NVI09b]. The environment also provides access to CUDA device management, memory management, multi-threading, and execution control APIs for integration with host applications. CUDA supports other high-level languages such as FORTRAN, with support for more languages planned by NVIDIA [NVI09b].

2.6.2.1 NVCC. *NVCC* is a compiler driver provided with the CUDA Toolkit. *NVCC* invokes all of the necessary tools and compilers included with the CUDA toolkit required to compile device code. Any kernels written in parallel thread execution (PTX) (CUDA instruction set architecture) or a high-level language like C must be compiled by *NVCC* into binary (cubin) code before being executed on the device [NVI09b]. Source code of a program may consist of sections of code intended for execution on the host and sections of code intended for execution on the device. Figure 2.10 provides an overview of compiling within the *NVCC* paradigm. *NVCC* is responsible for separating all of the host source code from device source code and producing the GPU binary object used for linking into the host code [NVI09b]. Device code is compiled into PTX or binary form by *NVCC*. The host code is then output either as C code by *NVCC* or *NVCC* may directly invoke a C/C++ compiler to produce object files for the host source code.

Applications can then load and execute the PTX code or cubin objects from *NVCC* using the CUDA driver API, allowing applications to ignore any generated host code produced when the PTX code or cubin objects were generated [NVI09b]. Applications may also link to any generated host code because the host code contains the necessary CUDA C runtime function calls to load and launch all PTX code or compiled kernels. Any PTX code loaded for execution by an application is compiled

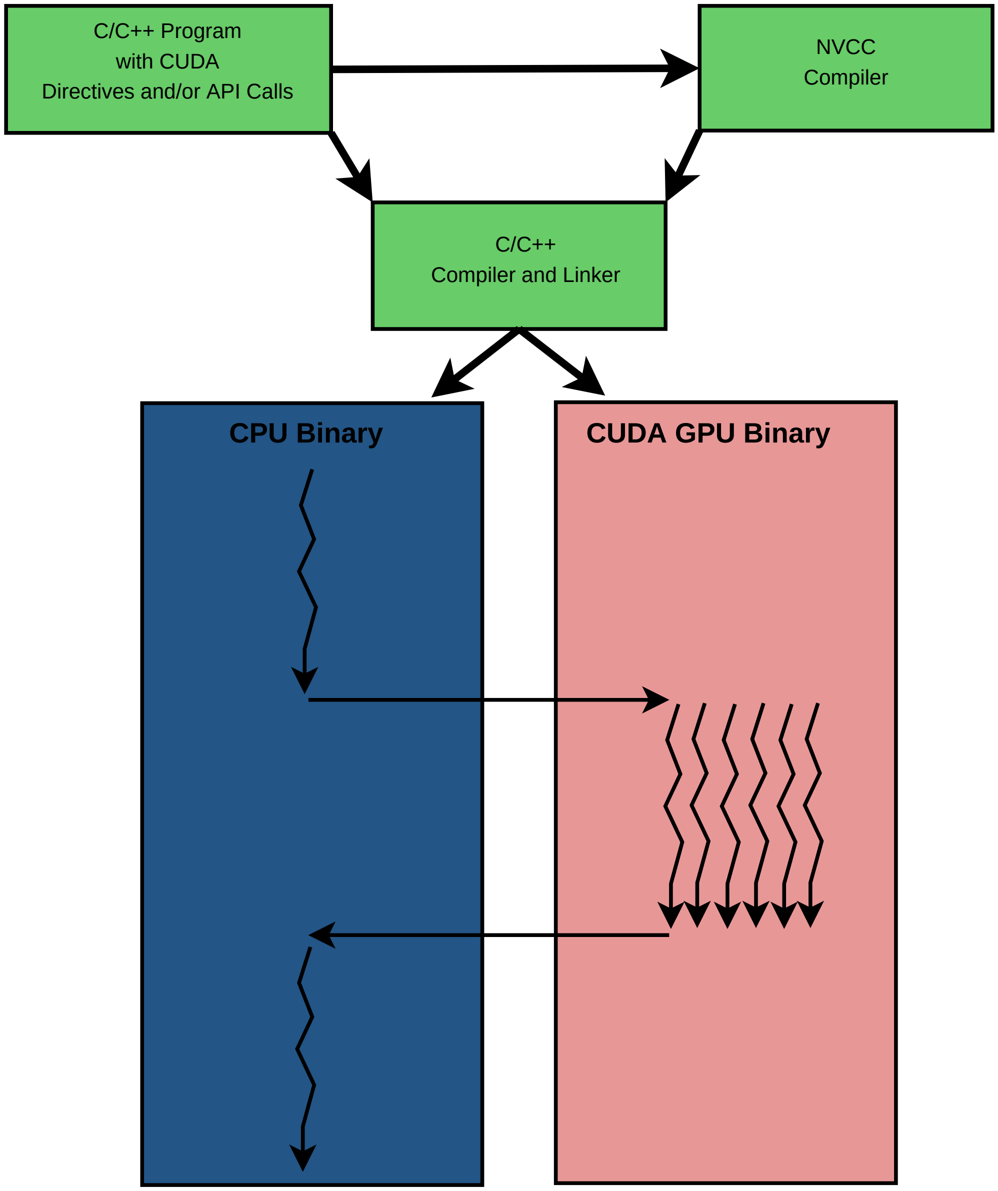


Figure 2.10: CUDA Nvcc Paradigm [NVI09b].

at runtime into binary code by the device driver for the GPU. This just-in-time-compilation does slow down the execution start, but allows applications to execute on devices that did not exist when the application was compiled [NVI09b].

PTX defines a virtual machine and instruction set for parallel thread execution on a GPU. The PTX architecture is designed for efficiency on NVIDIA GPUs [NVI09a]. At execution time, PTX instructions are translated and optimized for the target GPU architecture. This provides a scalable programming model for programming general purpose graphics processing units by allowing the binary code to be optimized just before execution to take advantage of new hardware. Since cubin binaries are compiled and contain hardware specific optimizations for the GPU hardware on which the binary is intended to run, the binaries are not guaranteed to run on different GPU hardware [NVI09b]. The cubin binary will start execution sooner than PTX code, but will be less flexible with hardware upgrades.

Figure 2.10 also shows how host code and PTX code (or cubin objects) interact during execution. The host thread is created and begins execution. The host thread will load the code to be executed on the GPU. When the code is executed on the GPU, multiple CUDA threads are created. After the CUDA threads finish, control returns to the host thread. This process may be repeated multiple times depending on the application.

2.6.2.2 CUDA Software Stack. CUDA includes three ways for an application to execute code on a GPU through the CUDA software stack. Figure 2.11 shows the overview of the CUDA software stack and how an application would interact with each part of the stack individually or indirectly through other parts of the stack. The CUDA software stack includes: the CUDA Driver, the CUDA Runtime, and the CUDA libraries. An application may directly use all, anyone, or a combination of these to execute code on a GPU. Each part of the CUDA software stack is discussed in detail in the following paragraphs.

The CUDA driver API is an imperative API based on handles [NVI09b]. Functions implemented in the *nvcuda* dynamic library manipulate objects referenced by opaque handles. Table 2.2 lists the objects supported by the driver API. The device object contains numerous properties that track the state of the device and allow the status of the GPU to be easily checked. The context object must be created and attached to a device object before the host thread can execute any code. A context object creates a CPU-like process on the GPU used to execute the kernel and transfer

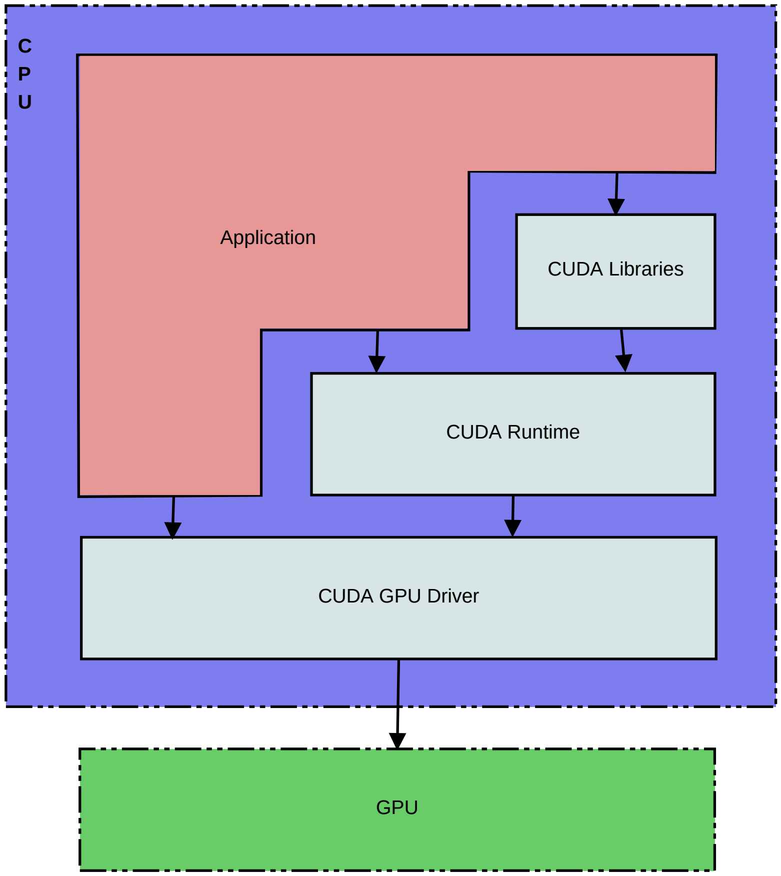


Figure 2.11: CUDA Software Stack [NVI09b].

data to and from device memory. A module object is similar to a dynamic library and is loaded by the CUDA Driver prior to execution. Multiple module objects may exist if multiple libraries are required for execution. A kernel is represented by a function object, representing the entry point for the GPU code execution. The heap memory, CUDA array, and texture reference objects are representations of memory structures. The heap memory object is a pointer to the heap in device memory. A CUDA array object is a container for array data on the GPU while the texture reference object provides a way to access texture data.

Since the context object must be created before the CUDA Driver will pass any instructions to the GPU, the runtime and libraries will create the context object the first time a function is used from either the runtime or libraries. This means that

Table 2.2: CUDA Driver API Objects [NVI09b].

Object	Description
Device	CUDA enabled device
Context	Roughly equivalent to a CPU process
Module	Roughly equivalent to a dynamic library
Function	Kernel
Heap Memory	Pointer to device memory
CUDA array	Opaque container for 1D or 2D data on device
Texture reference	Describes how to interpret texture data

knowledge of the CUDA driver functionality is not required because the runtime and libraries ensure the driver is properly initialized.

A CUDA context is created when a host thread first calls into the CUDA runtime library; the host thread that made the first call is the only thread with access to the CUDA context [NVI09b]. If a host system has multiple devices, any number of threads may execute device code on the same device. A thread on the host is limited to executing on one device at a time. If the host would like to execute on multiple devices simultaneously then multiple host threads (equal to the number of devices) would be required [NVI09b].

Two levels are provided by the CUDA Runtime API: the C API and the C++ API [NVI09c]. The C API provides an interface for C code and can be compiled using any C compiler and does not require the use of *NVCC*. The C++ API provides an interface for C++ code and can be compiled using any C++ compiler. The API also contains CUDA wrappers dealing with special device functions and requires the use of *NVCC* to correctly generate the necessary GPU instruction code. The CUDA Runtime API uses the CUDA Driver API to execute code on a GPU. Because only a single version of the CUDA driver can be installed on any one system and the Runtime and libraries are dependent on the CUDA Driver, all applications and libraries on a system are required to use the same version of the CUDA driver API [NVI09b].

CUDA provides a set of libraries for use in CUDA based applications. The *cublas* and *cufft* libraries are provided in the CUDA toolkit [NVI09c] [NVI09b]. The *cublas* library provides helper functions for error handling, memory allocation, and data transfer. The *cufft* library provides functions for parallel computation of the Fast Fourier Transform algorithm. The libraries are available to be integrated into C/C++ applications to assist with parallel code development for the GPU. The libraries are installed as part of the CUDA Toolkit.

2.6.3 GeForce 9500 GT. The XFX 9500 GT graphics card is built around a CUDA 1.1 enabled GeForce GPU by NVIDIA. It is made by XFX and is considered a mainstream graphics card [XFX09]. The low-profile design of the graphics card allows the card to be used in small compact desktops that may not be intended for gaming or powerful workstations. As shown in Table 2.3, the card contains 1 GB of DDR2 memory with a speed of 800 MHz and a 128-bit bus. The card supports resolutions up to 2560x1600, SLI configurations, and has a clock rate of 1.35 GHz. The PCI-Express 2.0 bus connects the graphics card to the host system. As shown in Table 2.4, the 9500 GT has 64 kB of constant memory and 16 kB of shared memory. It also supports concurrent memory copies and kernel execution, and places runtime limits on kernels to prevent runaway code. The GPU has four multiprocessors, each with eight cores, and allows multiple host threads to access the GPU simultaneously. The CUDA driver version 3.0 is installed on the host system to operate the 9500 GT graphics card. Version 2.30 of the CUDA runtime with Compute Capability 1.1 is also installed. The Compute Capability defines the hardware features each device is to implement and make available.

The GPU has a compute mode property set by the NVIDIA Control Panel, currently available only for Linux, that controls if the card is available for execution of a kernel. The default compute mode defines that multiple host threads may use the device. The exclusive compute mode limits device usage to only one host thread at a time. Prohibited compute mode disallows any thread to use the device. The

Table 2.3: XFX 9500 GT Hardware Specifications [XFX09].

Hardware Item	Value
Chipset	GeForce 9500 GT
Engine Clock	550 MHz
Bus Type	PCI-E 2.0
Number of Stream Processors	32
Memory Bus	128
Memory Type	DDR2
Memory Size	1GB
Memory Speed	800 MHz
Shader Clock	1375 MHz
Features	CUDA, DirectX 10, PhysX

Table 2.4: CUDA Memory Characteristics.

Property	Value
CUDA Driver Version	3.0
CUDA Runtime Version	2.30
CUDA Compute Capability	1.1
Total amount of global memory	1073454544 bytes
Number of multiprocessors	4
Number of cores	32
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	16384 bytes
Total number of registers available per block	8192
Concurrent copy and execution	Yes
Run time limit on kernels	Yes
Compute Mode	Multiple host threads

compute mode can be checked by retrieving the computer mode property from the device. If an application is requesting a specific device, then it is necessary to verify the device's compute mode to ensure the device is available [NVI09b].

2.7 ClamAV Engine

ClamAV is an open source anti-virus toolkit. The toolkit consists of a shared library and virus database. The malware database includes support for standard, compressed, obfuscated, or packed PE files [Cla09a]. It serves as the base for the ClamWin Anti-virus program for Microsoft Windows [Cla09c].

The ClamAV Virus Database is a [.cvd] file containing a 512 byte header and a compressed section of signature databases. The header contains various information about the CVD including MD5 checksum and a digital signature. The header has the following format [Cla09b]:

> *ClamAV-VDB:build time:version:number of signatures: functionality level required:MD5 checksum:digital signature: builder name: build time(sec)*

The compressed section of signature databases contains multiple databases. The header must be removed before the databases can be decompressed. Each database contains MD5 hashes or hex strings as the signature and serves a different purpose. Table 2.5 gives the databases with purpose and entry syntax. The [.hdb] and [.mdb] databases contain MD5 signatures for PE files. [.ndb] and [.db] databases contain hex signatures for PE files, while [.zmd] and [.rmd] databases contain CRC32 signatures for the meta data inside ZIP and RAR files. The [.fp] database contains a list of signatures that are white listed in all of the other databases.

The shared library is designed for a serial CPU and is not designed for use on a GPU. Therefore it is necessary to develop a parallel library. The Clam AV library will serve as a good example, providing code that can be ported to work on a GPU.

2.8 Related Work

The parallel nature of the GPU makes it a good choice for linear algebra and cryptography applications. Recently the GPU has been used in molecular biology, physics, chemistry, and weather prediction to increase the performance of algorithms [NVI10] [MiV08]. GPUs have also been successfully applied to image and signal processing, database management, financial services, and audio encoding and

Table 2.5: ClamAV Databases with Purpose and Signature Format [Cla09b].

Database	Purpose	Format
.hdb	MD5 signatures for PE files	MD5:number:filename
.mdb	MD5 signatures for PE file sections	PESectionSize:MD5:MalwareName
.ndb	Hex signatures with wild-card characters for PE files	MalwareName:TargetType:Offset: HexSignature[:MinEngine Funcationl-ityLevel:[max]]
.db	Hex signatures for PE files	MalwareName=HexSignature
.zmd	CRC32 signatures based on metadata inside ZIP archive files	virname:encrypted:filename:normal size:csize:crc32:cmethod:fileno:max depth
.rmd	CRC32 signatures based on metadata inside RAR archive files	virname:encrypted:filename:normal size:csize:crc32:cmethod:fileno:max depth
.fp	List of signatures in the other databases that are white listed.	db name:line number:signature name

decoding [NVI10] [HoW04]. These are just a few of the uses of the GPU; there are many more applications.

Hu *et al.*, proposed a high throughput GPU implementation of the MD5 algorithm [HMH09]. The proposed method is based on the standard MD5 algorithm, but breaks the data into smaller blocks. Each block is hashed using MD5 individually, then the resulting hashes are then hashed using MD5 to produce a master hash result. The master hash can then be used as a fingerprint for the data. This implementation has been shown to increase the throughput of MD5 algorithm on the GPU 20 times over the standard implementation of MD5 [HMH09]. While the throughput of the MD5 algorithm has increased, the results (hashes) will be different than those produced by the standard MD5 algorithm. This means this method is not compatible with current malware databases based on the MD5 algorithm. This method could be used in future malware databases designed to leverage the parallel power of the GPU.

Collange *et al.* successfully applied the parallel power of the GPU to forensics data carving [CDD09]. They use a GPU to detect image file byte patterns in sample individual disk clusters. The patterns are fingerprinted by hashing (using the CRC64 algorithm) and the hashes are then used for matching. The hashes of the patterns are compared against hashes of patterns from known images. The GPU implementation with all data in graphics memory was shown to outperform a software implementation on a CPU and improve the search process performance 13-fold by providing higher data throughput [CDD09]. This shows the GPU can increase the performance of hashing and hash searches (or hash matching).

Nigel Jacob and Carla Brodley proposed PixelSnort, a GPU port of the popular open source intrusion detection system (IDS) Snort [JaB06]. The authors noticed that the performance of Snort significantly decreases when the load on the IDS-host increases. PixelSnort is designed to off-load some of the IDS computation to a GPU [JaB06]. The GPU uses a string-matching algorithm to identify network packets; the authors use a simple algorithm and acknowledge it may not be optimal for a GPU. PixelSnort outperforms Snort by up to 40% under heavy loads [JaB06]. While the authors did not have a significant speed up under normal load conditions; PixelSnort demonstrates the GPU can be used for off-loading computational intensive tasks while providing performance increases.

Huang *et al.* also used a GPU to increase the performance of an IDS [HHL08]. The authors proposed an algorithm similar to the Wu-Manber algorithm designed to take advantage of the GPU's parallel nature. Their proposed approach increases performance by two fold over the modified Wu-Manber algorithm used in Snort. The proposed approach can be applied to signature-based anti-virus systems to detect malware.

Kouzinopoulos and Margaritis explored using a GPU for string matching [KoM09]. This process looks for a small subset of string data within a larger set of data. By using the parallel architecture of the GPU, the authors were able to obtain a twenty-four fold increase over the serial implementation on a CPU. String matching is often used in malware detection. Some malware databases, like the one used in Clam AV, contain strings that appear within malware, these strings are then compared to the file contents allowing for additional detailed detection. This shows a GPU increases the performance in string matching algorithms and supports the idea that a GPU could be used in commercial anti-virus products.

Mario Juric [Jur08] used a GPU and CPU to calculate hashes of strings and then compare each hash to a given hash database. The research determined that the optimal number of threads per block on a GPU for a GeForce 8800 Ultra is 63. It also showed that the GPU was 36 times faster than the CPU when executing the same code. The research was limited to strings of 56 characters, so all data would fit in shared memory. This research shows a GPU can increase the performance of MD5 hashing and database searching of strings.

Bhattarakosol and Suttichaya [BhS07] proposed using multiple threads and file size grouping to increase the speed of malware detection. This method makes use of the multiple threads on a standard CPU. The files are grouped according to size, with a thread assigned to each group. Malware detection speeds increased when compared to using a single threaded process. This research displays the advantage to using multiple threads during malware detection to maximize efficiency on the CPU, giving promise to the potential speed increase using a GPU with multiple lightweight threads. It also shows that grouping files by size for each thread block may provide a performance increase by reducing the time finished threads in the thread block idle, waiting on other threads to finish.

GPUs are used in two volunteer computing projects to achieve performance increases. Folding@Home is a community volunteer project that looks at protein folding [Sta10]. The project supports heterogeneous hardware (CPU and GPU). Folding@Home distributes a problem over all CPUs and GPUs in the community. The project has shown that GPUs give a 10 fold performance increase over a CPU [Sta10]. BONIC is another community volunteer computing project. BONIC solves various scientific applications instead of just concentrating on protein folding like Folding@Home [Ber10]. It uses GPUs, but the performance increases have not been quantified. This shows the diversity of the GPU and how it has been applied to solve problems.

2.9 Summary

This chapter presents background information on static malware detection. The Portable Executable File Format used in Microsoft Windows operating systems, the use of MD5 for fingerprinting files, and the Pentium 4 CPU are also discussed. PCIe, the I/O bus connecting the GPU to the host system, is explored and its effects on data transfers discussed. The advancements of GPUs for general purpose computing are studied in detail, and the Clam AV database is presented. Finally, related work and research are discussed. Based on the information in this chapter, a GPU appears to be a good choice for offloading file fingerprinting and MD5 hash searches.

III. Methodology

This chapter outlines the methodology used to evaluate the performance of the GPU ID system using time to inspect executables and the number of correct identification as performance metrics. Section 3.1 discusses the goals and hypotheses, and Section 3.2 discusses the approach. The system boundaries are discussed in Section 3.3; the system services are discussed in Section 3.4. A description of the workload is presented in Section 3.5; performance metrics and system parameters are presented in Section 3.6 and Section 3.7, respectively. The factors are discussed in Section 3.8, followed by the evaluation technique in Section 3.9. Finally, the experimental design is discussed in Section 3.10.

3.1 Goals and Hypothesis

The primary goal of this research is to use a GPU to correctly discriminate between malicious and benign files using predetermined signatures. Current techniques of detecting malware uses a serial scan of files, which can lead to increased scanning time as the number and size of the files increase. It is expected that the GPU will be able to rapidly hash the binary code of a file and compare the hash to a database, with 100% detection rate of known malware, because of its ability to operate like a CPU. It is also expected that since the GPU is highly parallelized it will simultaneously inspect multiple files at the same time.

The second goal of this research is to measure the performance of using a GPU for detection of malware. This determines whether the approach is feasible for products such as commercial anti-virus products. It is expected that GPU will increase the speed of detection and will result in faster processing of the executables because there is higher memory bandwidth available to a GPU, over a CPU.

The third goal of this research is to find the optimal number of threads per block for calculating MD5 hashes with the GPU ID system and for searching the signature database for matches. The GPU ID system uses two CUDA kernels, one for calculating the MD5 hashes of the files, and one for searching the MD5 database

Table 3.1: Graphic Processing Unit IDentifier Experiment Summary.

Experiment	Metric	Goal
1	Time to Calculate MD5 Hash of All Files	Find optimal number of threads per block for MD5 hashing.
2	Time to Search Signature Database	Find optimal number of threads per block for searching the database
3	Probability of Detection	Detect malicious and benign files using predefined signatures
3	Detection Time	Measure performance of the GPU during detection

for a signature match. Since two kernels are used, each kernel may have a different number of threads per block. It is expected that the number of threads per block for calculating MD5 hashes will be 63; this is based on previous research by Mario Juric [Jur08]. The number of threads per block for searching the signature database is expected to be 512 (the maximum number of threads per block allowed). This is because the cost of loading the computed hashes to shared device memory first is best distributed across the maximum number of threads per block allowed.

For Goal #1, detecting Malicious and Benign Files Using Predetermined Signatures, the hypothesis is a GPU would detect 100% of the known malware with no false positives (disregarding MD5 collisions). For Goal #2, measuring the Performance of a GPU, the hypothesis is a GPU will decrease detection time, while processing executables faster than a CPU for a given number of threads per block. For Goal #3, finding the Optimal Number of Threads per Block, the hypothesis is the optimal number of threads per block for calculating MD5 hashes is 63, while the optimal number for searching the database for a signature match is 512.

Three experiments are conducted to determine if the GPU ID system meets the stated goals and hypotheses. Table 3.1 summarizes the metrics and goals used in the experiments to evaluate the GPU ID system.

3.2 Approach

The GPU ID system was developed on an NVIDIA GeForce 9500 GT graphics card by XFX. The reason the GPU ID system is developed on the GeForce 9500 GT is because the GPU supports the CUDA architecture and is considered a mainstream GPU. Since it is a mainstream GPU it is available in desktops intended for everyday use, and not those only intended for gaming or specific applications. The GPU is used without modifications to the factory settings and with the driver supplied by NVIDIA (driver version 3.0). The software used with the GPU is based on the MD5 algorithm (as described in RFC 1321) and Clam AV (version 0.95.3) open source project, while the software implementation for the CPU is based on the software for the GPU with minor changes (discussed later in Section 3.2.1). The signature databases used in the experiments are modified versions (discussed later in Section 3.2.3) of those included in Clam AV.

3.2.1 Software. The GPU ID software consists of initialization host code, two kernels implemented in CUDA, and completion host code, as shown in Figure 3.1. The initialization host code, running on the host, initializes the device (GPU) using the CUDA Runtime libraries and then loads files and databases from disk on the host (CPU) to device memory. The device code is divided into two kernels. The first kernel calculates the MD5 hashes for all files loaded into memory and saves the hashes to device memory. The second kernel loads the calculated hashes to shared memory on the device and then allows each thread to retrieve one signature from the database and compare it with each of the generated MD5 hashes searching for a match. If a match is found, a corresponding flag is set in device memory. The hashes are first loaded to shared memory to reduce the memory latency when accessing the values. The MD5 hash from the database is loaded into four 32-bit registers for each thread so the signature is loaded only once from the database in memory. The completion host code runs on the host and copies the match flags from device memory to the host for processing (i.e., print results to screen). Pilot tests reveal that a linear search

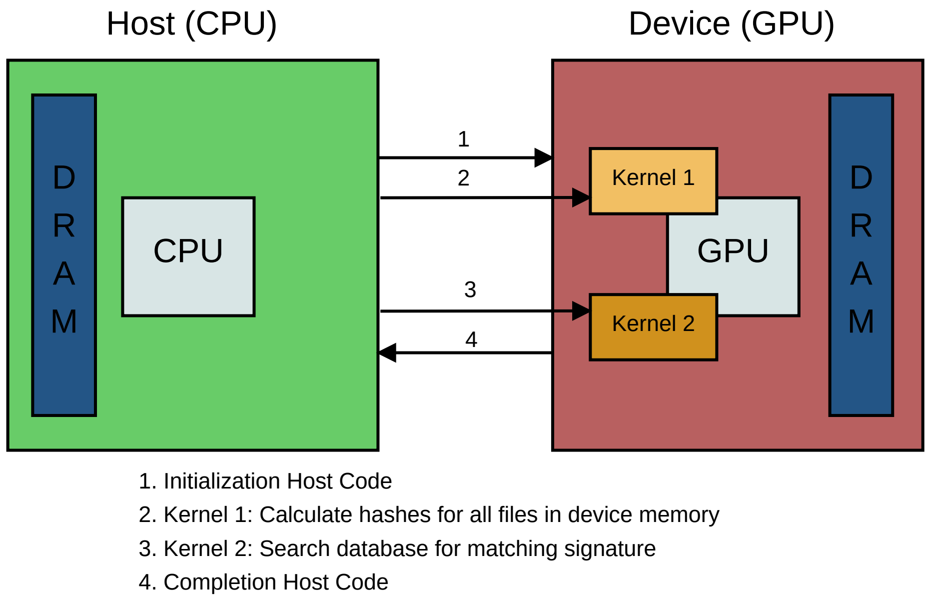
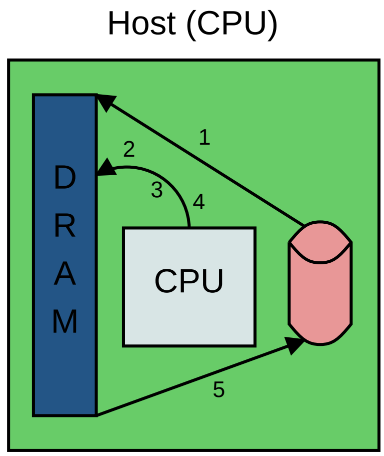


Figure 3.1: Overview of the GPU ID System

is faster on the GPU than sorting and using a binary search. This is most likely due to the large amounts of costly global memory accesses required to sort the database and then perform the binary search.

As shown in Figure 3.2, the software implementation used on the CPU is similar to GPU ID, except all code runs on the host and uses built-in Windows system libraries. First the files and signature databases are loaded into memory. Then the MD5 hashes are calculated for all files in memory and the hashes stored in memory. Next the signatures are sorted using the built in Quick Sort function in C++. Each generated MD5 hash is compared to the hashes in the signature database using a binary search. If there is a match it is recorded in memory for later processing. Pilot tests reveal that a sorted database with a binary search performs better on the CPU than a linear search.

3.2.2 Malicious and Benign Files. A total of 1,024 executable files were collected from a Microsoft Windows XP system; all of which were less than 192 kB in size. The file size was limited to files less than 192 kB because the files collected from the active Windows XP system only provided enough files (1,024) for the ex-



1. Load files and database to memory
2. Calculate MD5 hashes for files
3. Quick sort the signature database
4. Search the database using a binary search
5. Record matches for processing

Figure 3.2: Overview of the GPU ID System Implementation on a CPU.

periments at this level. These files were then divided into four groups: two groups of executables less than 96 kB (small executables) and two groups 96 kB or greater (large executables). The files were split at 96 kB because this division gave enough files for each group (256 files). One group from each of the small executables and large executables are further classified as malicious or benign. This classification was made randomly.

3.2.3 Signature Databases. The signature database used in the experiments is based on the Clam AV malware databases of full hashed executables with MD5 signatures. The databases in Clam AV with MD5 signatures are combined into one database, and then the hashes of the 512 files representing malicious files are randomly inserted in the database. The database has a total of 730,336 MD5 signatures. The database is checked for the MD5 signatures of the 256 files representing benign files to verify they are not listed. All MD5 signatures for the building and validation of the database were computed by HashCalc version 2.02 [Sla10].

3.2.4 GPU ID Algorithm. The GPU ID algorithm is comprised of the following steps:

1. Calculate the MD5 hashes for the 256 files loaded into memory.
2. For each thread block: load the 256 hashes to shared memory.
3. Each thread retrieves a different signature from the database in memory.
4. Compare each file hash to the signature from the database.
5. If there is a match, record file as malicious.
6. If there is not a match, assume file is benign and do not record.

Step 1 is done in a separate kernel from Steps 2 - 6. This is done to allow for more efficient use of the GPU hardware by using different configurations for each kernel.

3.3 System Boundaries

Figure 3.3 shows the system under test, the GPU ID System. It includes a Dell Optiplex GX620 with a Intel Pentium 4 processor with Hyper-Threading enabled and 3 GB of RAM. The PC has minimal I/O devices (monitor, mouse, keyboard, and a disk drive), Microsoft Windows XP operating system version 2002 SP3, the CUDA Toolkit and SDK 2.3 from NVIDIA, a mainstream top-of-the-line NVIDIA GeForce 9500 GT graphics card, and GPU ID program to load the signature database and scan the executables.

The component under test is the GPU ID program. Figure 3.4 shows the component under test.

The workload parameters include benign and malicious executables. The system parameters are the executable size, executable type (benign or malicious), and the processing hardware (GPU or CPU). The metrics include the execution time and the identification result which is used to calculate the probability of detection for known malware.

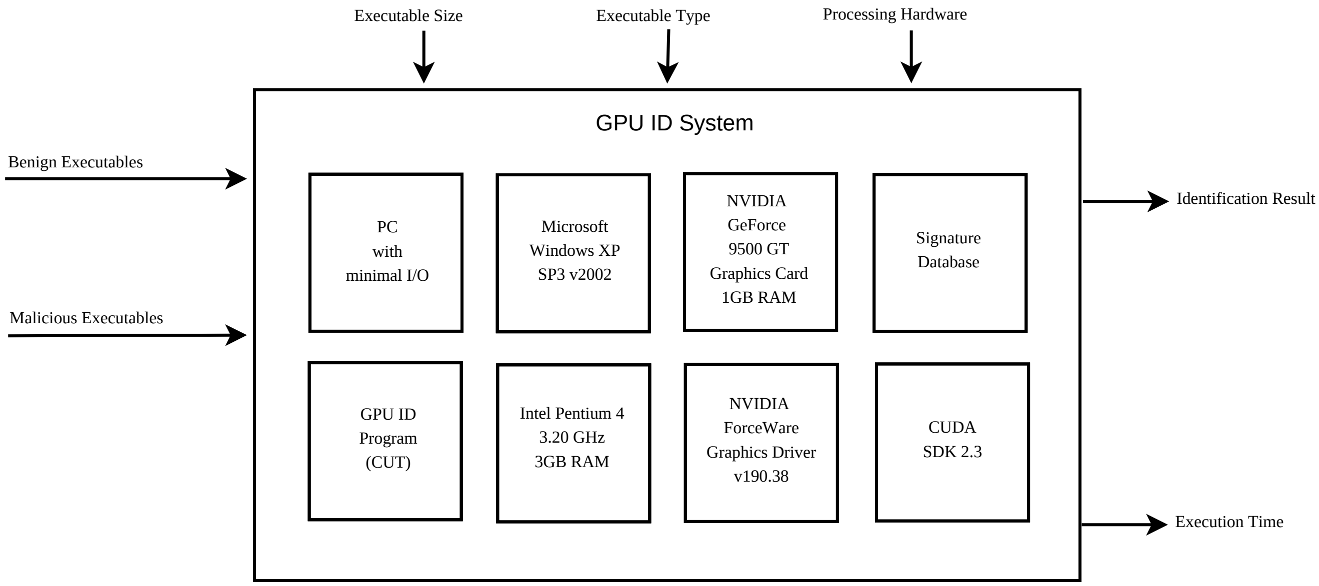


Figure 3.3: The GPU ID System.

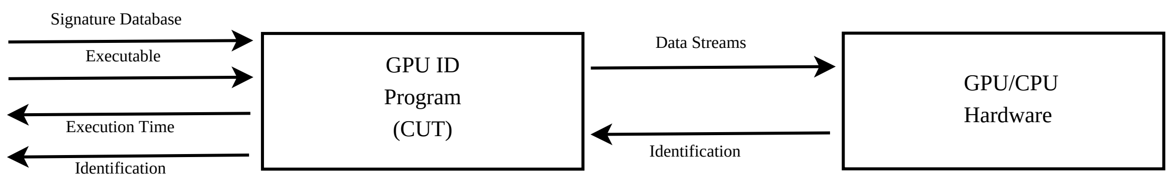


Figure 3.4: Component Under Test (CUT).

Experiments 1 and 2 use the system under test but varies the number of threads per block on the GPU. The component under test is the same except execution time is measured for each individual kernel execution. The identification result returned from the GPU is used only to verify the system is functioning correctly.

To determine the performance using a GPU for Experiment 3, the same system under test is used. The GPU ID system's performance is compared to a software implementation on a CPU. Both implementations use the same executable and signature database, with the difference being intended execution hardware (GPU or CPU).

3.4 System Services

The service provided by the GPU ID system is to identify an executable as benign or malicious. The GPU ID system is designed to assist in the detection of malware or files of interest when there are a large amount of files for processing.

The system is successful when the following happens for all files loaded onto the GPU:

- The file's hash is correctly calculated (i.e., the results are correct).
- If the file's hash is in the database, the file is identified as malicious.
- If the file's hash is not in the database, the file is identified as benign.
- The GPU device does not return an error code at any time.

A failure occurs when any of the following happen:

- The file's hash is incorrectly calculated (i.e., the results are not correct).
- The file's hash is in the database, but the file is identified as benign.
- The file's hash is not in the database, but the file is identified as malicious.
- The GPU device returns an error code at any time.

These failures are possible if any of the inspection algorithms are flawed, the time limit for kernel execution on a GPU is reached, or memory on the GPU is cannibalized for display purposes.

It is possible that two MD5 hashes will collide, resulting in a benign file being identified as malicious (false positive). Collisions are not considered in this system, because the chance of collision is 1 in 2^{128}. If collisions were a concern, the system could use a different form of signatures, such as string matching, or a different hashing algorithm.

3.5 Workload

The workload consists of executables which are labeled as either benign or malicious. Each executable contains binary code for different functionality (i.e., no two executables are the same). The workload is varied by changing: the size of the executable, executable type, and the processing hardware.

The workload consists of 1,024 executable files from a Microsoft Windows XP system. Half of the executable files are designated as malicious by randomly inserting the MD5 hash signature into the malware database. The other half of the files are considered benign, and it is verified that the MD5 hash for these files are not in the malware signature database.

The size of the executable is the most important factor of the workload since it directly affects the time needed to inspect the binary code. The size of the executable is measured as the size of the file, not necessarily the size set aside by the Windows XP operating system to store the file on disk. Executables of different sizes test the flexibility of the system.

3.6 Performance Metrics

Two performance metrics are used to evaluate the GPU ID system; they are the identification result and the execution time.

3.6.1 Identification Result. The identification result demonstrates that each system is producing correct results and serves as a quantity used to validate if the system is working correctly. A correct identification is when malware is identified as malware and benign files are not identified. A malicious file will have a match in the malware database. The identification result is used in all three experiments to validate each experiment is correctly identifying the files.

3.6.2 Execution Time. The execution time, or the time required to process the executables, is measured differently for each experiment. For Experiment 1, execution time starts immediately after the group of executables are sent to the GPU and stops when the MD5 file hashes are completed and the GPU returns a success code. Execution for Experiment 2 starts from the time the search of the malware database starts (kernel execution starts) and stops when results are returned from the GPU, or when a failure notice is returned. For Experiment 3, this time is measured from the time immediately after the group of executables are sent to the GPU/CPU until the results are returned from the GPU/CPU or when a failure notice is returned.

3.7 System Parameters

The three GPU ID system parameters are the executable size, executable type, and the processing hardware. In all of the experiments, the executable size is varied by using two different sizes of executables. The executable type is always malicious or benign, with benign representing the worst case scenario for the system because all MD5 hashes in the database must be searched. The processing hardware is either the GPU for the GPU ID system or the CPU for the software implementation.

3.8 Factors

In all experiments the executable size and executable type are varied. The size of the executable has two levels: small and large. Small is defined as 96 kB or less; large is defined as greater than 96 kB, but less than 192 kB. These two sizes are chosen because malware is generally small and can travel fast over a network to avoid detection. This factor is varied because the time required to identify an executable as malicious should increase with the size of the executable.

Executable type is defined by the executable having its MD5 signature listed in the malware database, or the file being benign (not listed in the malware database). This factor is varied because extra time will be needed to set the malicious flag for the file in memory. Depending on the executable type, this will result in different memory access patterns.

Table 3.2: Factors and Associated Levels for Experiments 1 and 2.

Factors	Levels
Executable Size	Small - 96 kB or less Large - greater than 96 kB, less than 192 kB
Executable Type	Benign Malicious
Number of Threads per Block	1-256 for Experiment 1 256-512 for Experiment 2

Experiments 1 and 2 are only run on the GPU, but the number of threads per block are varied. For Experiment 1, calculating the MD5 hashes of files, 1-256 threads per block are used. Since there are only 256 files in each group and the MD5 algorithm cannot be split into smaller pieces, there is no reason to try more than 256 threads per block. For Experiment 2, searching the malware database for MD5 hash matches, 256 - 512 threads per block are used. Since there are 256 file hashes that must be loaded into shared memory on the GPU, it is not cost effective to use fewer threads. The maximum number of threads per block on this GPU is 512; a number greater than 512 will cause the GPU to return an 'unavailable resource' error instead of results. Table 3.2 summarizes the factors for Experiments 1 and 2.

In Experiment 3, the processing hardware is varied in addition to the executable size and executable type. Processing hardware is the type of processing unit performing the calculations on the file stream. This factor is varied because the time required to scan files should decrease with the use of a GPU due to its highly parallel architecture and the CPU's serial architecture. Table 3.3 summaries the factors for Experiment 3.

3.9 Evaluation Technique

Direct measurement is selected as the evaluation technique for the experiments because all resources are readily available. In addition, the identification (or classi-

Table 3.3: Factors and Associated Levels for Experiment 3.

Factors	Levels
Executable Size	Small - 96 kB or less Large - greater than 96 kB, less than 192 kB
Executable Type	Benign Malicious
Processing Hardware	GPU CPU

fication as benign or malicious) can be stored and the time needed to process the executable easily measured. Simulation and analytical analysis of graphics cards is not practical since the cards are proprietary and not all implementation details are available.

The following hardware is used in the experimental configuration:

- The PC is a mainstream Dell Optiplex GX620. The processor is an Intel Pentium 4 CPU running at 3.20 GHz with Hyper-Threading enabled. It contains 3 GB of DDR2 memory in a dual channel configuration. Table 3.4 shows detailed specifications of the PC.
- The GPU is a mainstream graphics card - NVIDIA GeForce 9500 GT graphics card (XFX). The GPU has 32 stream processors and features one GB of DDR2 memory. Table 3.5 shows detailed information on the XFX 9500 GT graphics card.

To determine the performance of the GPU, its execution time is monitored. The CUDA API provides a system independent way to track execution time. Using the API, a timer with 32-bit resolution can be created, started, and stopped. The timer measures elapsed time in milliseconds. This method may be used for execution timing on a GPU or CPU. The first experiment measures only the execution time required

Table 3.4: PC Specification Overview.

Item	Values
PC Manufacturer	Dell
Processor	Intel Pentium 4 640
Processor Package	Socket 775 LGA
Processor Speed	3.20 GHz
Front Side Bus	800 MHz
Memory Type	DDR2
Memory Size	3 GB
Memory Configuration	Dual
Hyper-Threading	Enabled

Table 3.5: GeForce 9500 GT Specification Overview.

Item	Values
Chipset	GFGF 9500 GT
Engine Clock	550 MHz
Bus Type	PCI-E 2.0
Stream Processors	32
Memory Bus	128-bit
Memory Type	DDR2
Memory Size	1 GB
Memory Speed	800 MHz
Shader Clock	1375 MHz
Features	CUDA, DX 10DX, PhysX

to calculate all file hashes. The second experiment measures only the execution time required to search the malware database for possible matches.

The execution time is monitored the same way in Experiments 1 and 2, except the experiment measures the time required to calculate all file hashes and search the malware database for possible matches respectively. This time is compared to the time used for the same group of executables to be scanned on a CPU. The GPU code to detect malware is validated by comparing the number of malicious files found by the GPU to the number of malicious files found by the CPU. These numbers should be the same because the same malware database is used.

The following assumptions are valid for this experiment:

- The GPU is not handling graphical display. All monitors were unplugged from the PC and the Scheduled Task feature of Windows XP was used to load and start the program.
- The CPU is not taxed with running software, only the OS is functioning.
- All files and the malware database for each experiment are loaded into memory before the experiment starts.

3.10 Experimental Design

3.10.1 Experiment 1. A full factorial experimental design will be used to fully measure the effect of varying the number of threads per block on execution time. One run is executed for each level of executable size (2), executable type (2), and number of threads per block (512). Each experiment is run 50 times for a total of 102,400 runs. For execution time, a one-variable t-test is used to determine the mean execution time of the first kernel along with the standard deviation, and the standard error of the mean. A 95% confidence interval is used for the mean. A 100% probability of correctly identifying the file as malicious or benign is required. This is necessary to ensure the system is functioning properly and none of the executables are mislabeled.

3.10.2 Experiment 2. A full factorial experimental design will be used to fully measure the effect of varying the number of threads per block on execution time. One run is executed for each level of executable size (2), executable type (2), and number of threads per block (512). Each experiment is run 50 times for a total of 102,400 runs. For execution time, a one-variable t-test is used to determine the mean execution time of the second kernel along with the standard deviation, and the standard error of the mean. A 95% confidence interval is used for the mean. A 100% probability of correctly identifying the file as malicious or benign is required. This is necessary to ensure the system is functioning properly and none of the executables are mislabeled.

3.10.3 Experiment 3. A full factorial experimental design will be used to fully measure the effect of the size of the executable and the effect of the type of executable against the effect of the type of processing hardware. One run is executed for each level of executable size (2), executable type (2), and number of threads per block (512). Each experiment is run 100 times for a total of 800 runs. For execution time, a one-variable t-test is used to determine the mean execution time of the application along with the standard deviation, and the standard error of the mean. A 95% confidence interval is used for the mean. A 100% probability of correctly identifying the file as malicious or benign is required. This is necessary to ensure the system is functioning properly and none of the executables are mislabeled.

3.11 Methodology Summary

A GPU and CPU are used to classify executables as malicious or benign. The size of the executable, executable type (malicious or benign), and number of threads per block are varied in a full factorial experimental design in the first and second experiments. The experiments record if the file is benign or malicious and measure the time required to calculate MD5 hashes for the files and the time to search the malware database for a match. This information is used to analyze the performance of GPU hardware in relation to the number of threads per block, which allows the GPU ID system to be optimized in Experiment 3.

The size of the executable, executable type (benign or malicious), and processing hardware are varied in a full factorial experimental design in Experiment 3. The experiment records if the file is benign or malicious and measure the time required to identify the executable. This information can be used to analyze the performance of GPU hardware against CPU hardware.

IV. Results and Analysis

This chapter details and analyzes the experimental results of the three experiments. First, the results for Experiment 1 are discussed in Section 4.1. Section 4.2 details the results and analysis for Experiment 2. Section 4.3 presents the results and analysis from Experiment 3. Finally, an overall analysis of all results is given in Section 4.4, and a chapter summary is presented in Section 4.5.

4.1 Results and Analysis of Experiment 1

In Experiment 1 a GPU calculated teh MD5 hashes of 256 files. The number of threads per block were varied for calculating the MD5 file hashes. Looking at the plotted results of the mean MD5 hash times on a GPU in Figure 4.1 the following qualitative observations are made:

- Using less than 44 threads per block decreases performance of calculating MD5 hashes on a GPU by increasing the execution time 4% to 105%.
- There is no clear best number of threads per block for calculating MD5 hashes. The average performance for small benign files is between 0.0164550 and 0.0181401 milliseconds, small malicious files is between 0.0164988 and 0.0181774, large benign files is between 0.0166952 and 0.0182450, and large malicious files is between 0.0164176 and 0.0198692 for any thread per block value between 44 and 256.
- For the large malicious (Figure 4.1(d)) hash test, the means have greater variance (0.00345156 ms) from one mean to the next when compared to the large benign (Figure 4.1(c)) hash tests (0.00154980 ms).
- For the small benign (Figure 4.1(a) hash test, the means have greater variance (0.00168508 ms) from one mean to the next when compared to the small malicious (Figure 4.1(b)) hash tests (0.00167865 ms).

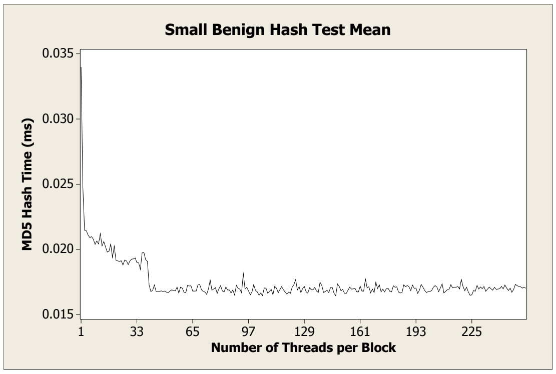
- Since there are only 256 files to calculate the MD5 hash for, the number of threads per block has a maximum of 256 threads per block - one thread for each file.
- For each set of files, there is a large dip between 37 to 43 threads per block.

Using less than 44 threads per block yields a decrease in the performance of calculating MD5 hashes on a GPU by 4% to 105%. Since threads are managed in groups of 32, the memory latency is better hidden with 44 or more threads. With 44 threads per block, this gives the GPU multiprocessor 2 warps to switch between during memory requests helping to hide memory latency. Also using greater than 256 threads per block would not yield any performance improvements since the threads above 256 would idle or return without executing any code.

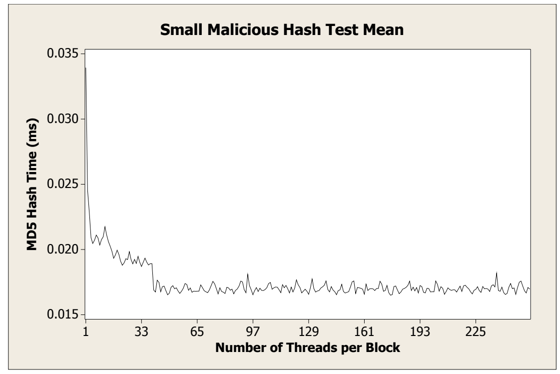
The mean of the small benign files have a greater variance(0.00168508 ms) when compared to the small malicious files (0.00167865 ms) and so do the large malicious files (0.00345156 ms) when compared to the large benign files (0.00154980 ms). It is expected that both the large and small malicious files would have a greater variance in the means than the small and malicious benign files due to the extra memory write required to set the malicious flag in global memory, but this does not happen in this experiment. The reason for this difference in variance is unknown.

Based on the mean times in Figure 4.1 and similar research [Jur08], 63 threads per block are used in Experiment 3 for calculating the MD5 hash. The number of optimal threads per block is not clearly identifiable, but it is clear more than 43 threads per block should be used.

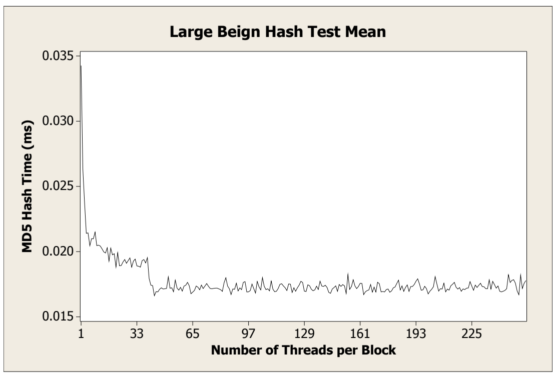
For each set of files there is a large dip between 37 and 43 threads per block. This dip is caused by the GPUs advanced thread scheduling hiding memory latency. As the number of threads increase from 37 to 43, the GPU has better ability to schedule threads performing computation, while other threads are waiting for memory requests to be fulfilled. This allows the GPU to keep the hardware busy with computations instead of idling, waiting on memory requests.



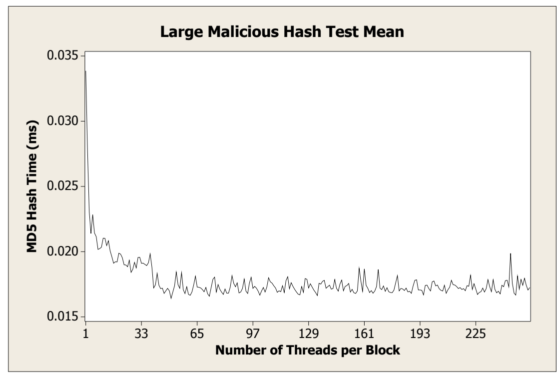
(a) Mean MD5 hash times for 1 - 256 threads per block for small benign files.



(b) Mean MD5 hash times for 1 - 256 threads per block for small malicious files.



(c) Mean MD5 hash times for 1 - 256 threads per block for large benign files.



(d) Mean MD5 hash times for 1 - 256 threads per block for large malicious files.

Figure 4.1: Mean MD5 hash times for 1 - 256 threads per block on a GPU.

With 44 threads per block, this gives the GPU multiprocessor 2 warps to switch between during memory requests helping to hide memory latency.

4.2 Results and Analysis of Experiment 2

In Experiment 2 a GPU compared 256 file hashes to a database of 730,336 using a linear search. The number of threads per block were varied when searching the database for MD5 hash matches. Looking at the plotted results of the mean MD5 database search times on a GPU in Figure 4.2 the following observations are made:

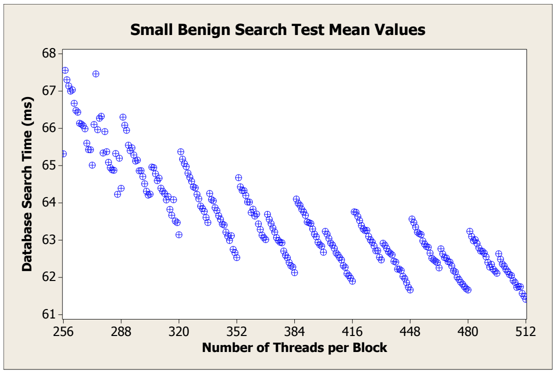
- Using fewer than 256 threads per block decreases the performance of the GPU, by increasing the time required to process the files by 600 to 800 milliseconds.

- An overall exponential decrease is seen as the number of warps (groups of 32 threads) increases.
- A grouping of 16 threads in a line and two lines to a group is seen on all four graphs.
- Figure 4.2(a)(b)(c)(d) shows the best number of threads per block are the maximum number of threads allowed in a block - 512.

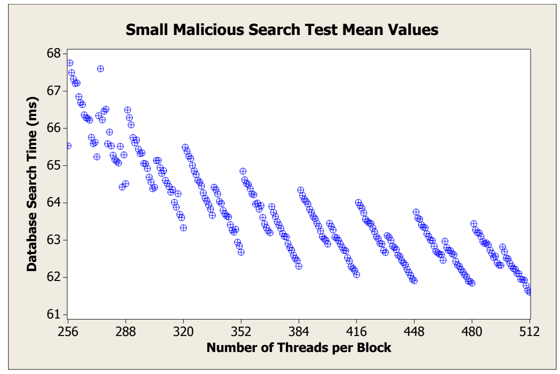
Using fewer than 256 threads per block causes some threads to make multiple memory reads to move data from global to shared memory. This decreases the performance of the GPU. A pilot test revealed that fewer than 256 threads per block would increase the time of processing files on a GPU by 600 to 800 milliseconds. This increase is from the conditional branching required for 256 threads per block to completely load shared memory and from the multiple memory accesses each thread must make.

As the number of threads increases, an overall exponential decreasing pattern is seen. This is due to the ability of each additional thread to take advantage of the data loaded into shared memory by the first 256 threads in a block. In this experiment 256 threads per block represents the worst case for taking advantage of shared memory and 512 threads per block represents the best case for taking advantage of shared memory. It is possible that if the GPU hardware allowed more threads per block than 512, search performance could be increased.

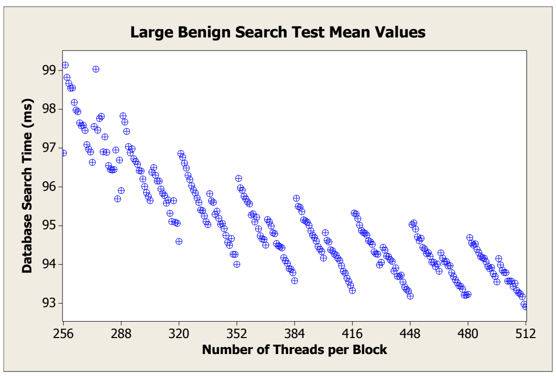
A grouping of 16 threads in a line and two lines to a group is seen on all four graphs. This is from the GPU management of threads in a warp. Memory requests are made for a warp and are combined into two memory requests, one for the first 16 threads and one for the second 16 threads of the warp. Combining the memory accesses for 32 threads into two memory transactions allows for more efficient use of the memory bandwidth.



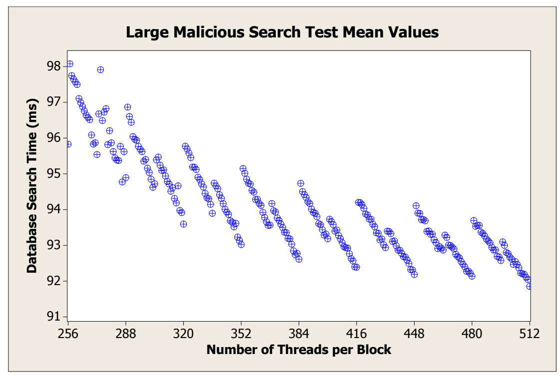
(a) Mean database search times and confidence intervals for 256 - 512 threads per block for small benign files.



(b) Mean database search times and confidence intervals for 256 - 512 threads per block for small malicious files.



(c) Mean database search times and confidence intervals for 256 - 512 threads per block for large benign files.



(d) Mean database search times and confidence intervals for 256 - 512 threads per block for large malicious files.

Figure 4.2: Mean database search times and confidence intervals for 256 - 512 threads per block on a GPU.

Based on the search times in Figure 4.2, 512 threads per block are used for Experiment 3 in searching the malware database for MD5 hash matches. This is the optimal number of threads per block on a XFX GeForce 9500 GT GPU.

4.3 Results and Analysis of Experiment 3

Experiment 3 tested the performance of a GPU against the performance of a CPU performing similar tasks. Both sets of hardware calculated MD5 file hashes, then compared each hash to a database of 730,336 MD5 hashes. The GPU used a linear search, while the CPU used a binary search, when locating matches in the database.

Table 4.1: Probability of Correctly Identifying Files.

Hardware	File Types	Probability of Correct Identification
GPU	Small Benign	1.0
GPU	Small Malicious	1.0
GPU	Large Benign	1.0
GPU	Large Malicious	1.0
CPU	Small Benign	1.0
CPU	Small Malicious	1.0
CPU	Large Benign	1.0
CPU	Large Malicious	1.0

Table 4.2: GPU ID Times (ms).

Configuration	N (Events)	Mean	Standard Deviation	Standard Error of the Mean	(95%) Confidence Interval
Small Benign	100	56.9169	0.1297	0.0130	(56.8912, 56.9426)
Small Malicious	100	56.7815	0.1044	0.0104	(56.7608, 56.8022)
Large Benign	100	93.231	1.030	0.103	(93.027, 93.436)
Large Malicious	100	91.963	1.025	0.102	(91.760, 92.167)

Table 4.1 shows the probability of each type of hardware correctly identifying files. It should be noted that in all experiments all files were correctly identified for all hardware. After the experiments are completed the calculated hashes are downloaded from device memory and compared to those calculated using HashCalc version 2.02. This is done after the experiments so the memory transfer does not affect the experimental results.

Table 4.2 shows the results of a one variable t-test performed on the different configurations run on the GPU. The table gives the number of trials, the mean time to complete the file scan, the standard deviation, the standard error of the mean, and a 95% confidence interval for the mean. The mean value is listed in milliseconds. The time required for the GPU to process the files ranges from 56.7815 to 93.963 milliseconds

Table 4.3: CPU Implementation Times (ms).

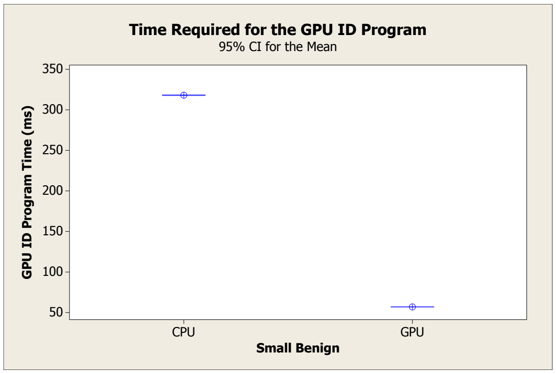
Configuration	N (Events)	Mean	Standard Deviation	Standard Error of the Mean	(95%) Confidence Interval
Small Benign	100	317.973	1.938	0.194	(317.589, 318.358)
Small Malicious	100	315.256	1.893	0.189	(314.881, 315.632)
Large Benign	100	636.513	3.351	0.335	(635.848, 637.178)
Large Malicious	100	625.963	7.739	0.774	(624.427, 627.499)

Table 4.3 shows the results of a one variable t-test performed on the different configurations run on the CPU. The table gives the number of trials, the mean time to complete the file scan, the standard deviation, the standard error of the mean, and a 95% confidence interval for the mean. The mean value is listed in milliseconds. The time required for the CPU to process the files ranges from 315.256 to 636.513 milliseconds.

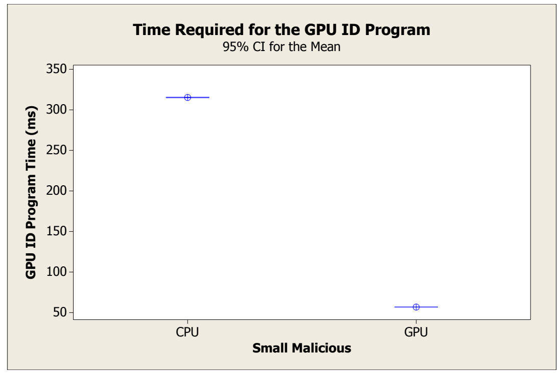
Figure 4.3 shows the 95% confidence interval plots of the time required to scan and identify files. In all cases the confidence intervals do not overlap, which suggests that the differences are statistically significant. The GPU performs better than the CPU for all groups of files. For small benign files the GPU is on average, 261.0561 milliseconds faster than the CPU, 258.4745 milliseconds faster for small malicious files, 543.282 milliseconds faster for large benign files, and 534 milliseconds faster for large malicious files. The figures also show that the benign files take slightly longer on both sets of hardware.

Hypothesis tests are performed between the GPU and CPU, to further determine the statistical significance of these results. As shown in Table 4.4, the p-value for the one-sided test for all four file groupings is 0.000, indicating a strong statistical certainty that the GPU outperforms the CPU in all cases.

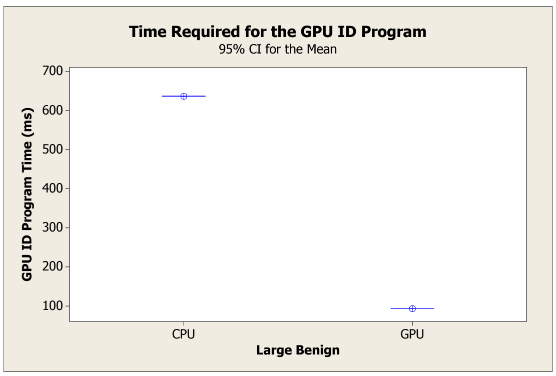
Table 4.5 shows the percentage change from CPU for the GPU for all configurations. Analyzing the data in this table, combined with the data from Tables 4.2 and 4.3, the following observations are made:



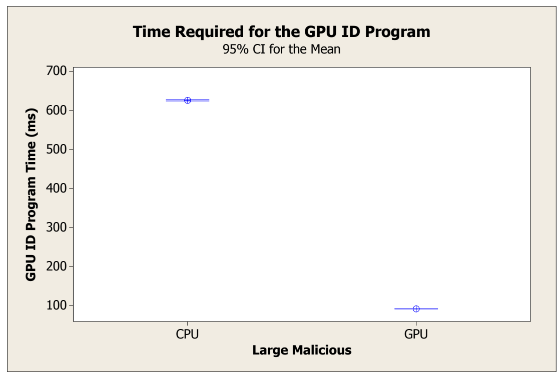
(a) Time required for small benign files.



(b) Time required for small malicious files.



(c) Time required for large benign files.



(d) Time required for large malicious files.

Figure 4.3: Time Required for the GPU ID Program to Identify Files.

Table 4.4: Hypothesis Testing on Performance of the CPU.

Alternative Hypothesis with 95% Confidence Interval	Estimate for Difference	T Value of Difference Test	P Value of Difference Test
GPU(Small Benign)< CPU(Small Benign)	261.056	1343.92	0.000
GPU(Small Malicious)< CPU(Small Malicious)	258.475	1363.42	0.000
GPU(Large Benign)< CPU(Large Benign)	543.282	1549.73	0.000
GPU(Large Malicious)< CPU(Large Malicious)	534.000	684.02	0.000

Table 4.5: Percentage Change of Configurations of a GPU from a CPU.

Configuration	Percentage Change from CPU
Small Benign	82.10%
Small Malicious	81.99%
Large Benign	85.35%
Large Malicious	85.31%

- For scanning files 0 - 96 kB on a GPU, system performance increased 82% over the CPU.
- For scanning files 96 - 192 kB on a GPU, system performance increased 85% over the CPU.

4.4 Overall Analysis

The results from Experiment 1 and 2 assist in the configuration of Experiment 3. While Experiment 1 does not provide clear results as to the correct number of threads per block to use, it does provide an answer as to what not to use. With this information and other research [Jur08], a reasonable number of threads per block, 63, is used in Experiment 3. Experiment 2 presents clear evidence to use 512 thread per block when searching the database, the maximum number of threads per block allowed on the GeForce 9500 GT GPU. It is possible that performance could increase if a GPU that allowed more threads per block were used. Experiment 3 reveals the increased performance a GPU offers over a CPU. The GPU increased performance over 82% even with a slower processor clock. There are four reasons for this large increase in performance:

- The file data is cache coherent and is only accessed once during hashing so there is no gain from cached memory as found with a CPU.
- The GPU has four stream processors compared to the one processor on the CPU. The four processors can each work individually, in parallel, unlike the single core CPU.

- The thread scheduling ability of the GPU hides memory latency and maximizes bandwidth.
- The GPU allows memory transactions to be combined into a single transaction reducing the amount of memory requests and increasing performance.

4.5 Summary

This chapter details and analyzes the results from the three experiments. A statistical analysis of the performance metric, execution time, is performed for Experiment 3. Finally an overall analysis of the results from the experiments is provided. The results show that a GPU increases performance over 82% from a CPU, while correctly identifying the files 100% of the time.

V. Conclusions

This chapter presents the conclusions drawn from the research. Section 5.1 compares the research goals with the experimental results to determine if the research objectives were met. The significance of the research is presented in Section 5.2. Finally, Section 5.3 provides recommendations for future work and expansion for this research.

For Goal #1, detecting Malicious and Benign Files Using Predetermined Signatures, the hypothesis is a GPU would detect 100% of the known malware. For Goal #2, measuring the Performance of a GPU, the hypothesis is a GPU will decrease detection time, while processing executables faster than a CPU. For Goal #3, finding the Optimal Number of Threads per Block, the hypothesis is the optimal number of threads per block for calculating MD5 hashes is 63, while the optimal number for searching the database for a signature match is 512.

5.1 Conclusions of Research

5.1.1 Goal #1: Correctly Detect Malicious and Benign File Using Predetermined Signatures. The first goal of this research is to correctly detect malicious and benign files using predetermined signatures on a GPU. The GPU ID system and the software implementation for the CPU are both able to correctly identify 100% of the files in all three experiments. The 100% accuracy of the system meets the stated goal and proves the hypothesis.

5.1.2 Goal #2: Measure the Performance of a GPU. The second goal of this research is to measure the performance of a GPU while detecting malware using predetermined signatures. The GPU ID system is tested against a CPU performing the same task and the required execution times compared. Experiment 3 reveals that the GPU ID system is at least 82% faster than the CPU implementation. The increase in performance when using a GPU, instead of a CPU, meets the stated goal and proves the hypothesis.

5.1.3 Goal #3: Find the Optimal Number of Threads per Block. The third goal of this research is to find the optimal number of threads per block for calculating MD5 files hashes and search a database of MD5 signatures on a GPU. Experiment 1 reveals that there is not a clear answer to calculating MD5 file hashes part of this goal, thus failing to meet the hypothesis of 63 threads per block. Experiment 1 did reveal that using a number of threads per block less than 40 would be suboptimal. Experiment 2 reveals that 512 threads per block is optimal when searching the database, thus meeting the goal and proving the hypothesis.

5.2 Significance of Research

This research provides the Air Force and other government agencies with a faster method to scan large amounts of files quickly for a predetermined signature. This system differs from other methods because it offloads part of the computation to a mainstream GPU. Since this is a mainstream GPU, it is readily available in newer PCs. It also reduces the overall load on the PC; increasing the usability of the PC to the user. Finally, this system can be easily expanded to include additional file types and hashing algorithms.

The GPU ID system is a passive system and therefore attractive to network administrators. The use of a GPU requires only that a supported GPU be installed on the target machine, and the GPU ID system be installed. In the event the GPU is not available then the system would continue protecting the target machine by using the CPU. This gives the system flexibility in case of a GPU failure.

When fully implemented, the GPU ID system is an effective tool in the fight against malware. It will decrease the scanning time allowing for quicker notification of an infected file and reduce the resource contention on the PC allowing greater usability of the system while scanning is taking place. It can also be used to scan large shares of files, either for malicious files or for changes made to files. This will increase the protection offered to both the files and users.

Finally, the GPU ID system should be considered as a tool to quickly scan recovered media or data for keywords, attributes, or other identifying markers. This could be of use to forensic investigators, custom agents, law enforcement agencies, network intrusion detection systems, firewall based applications, and anti-malware applications that would need to quickly identify a small subset of data from a larger set. This would reduce the amount of time required and could easily be adapted to just about any environment.

5.3 Recommendations for Future Research

The next logical step for this research is to expand the system and look at files of all sizes, not just files between 0 - 192 kB. The GPU ID system should be tested on a workstation that would mimic that of an actual user. The workstation should include files of all types including executable, html, pdf, Microsoft Office files, etc., so the system can be throughly tested. It would also be a good idea to explore the performance impact of grouping files by size on a GPU, similar to Bhattarakosol and Suttichaya's research [BhS07].

MD5 hashing is not the only way to detect malicious files. Future research could include expanding the system to use string matching techniques or other analysis to classify files. These techniques could even be mixed to other an improved and efficient detection tool. These techniques could also include using a different hashing algorithm that is more efficient in a parallel environment.

Applying the GPU to deobfuscation and unpacking of files before scanning is another area of future research. This research assumes that the files are not encrypted or obfuscated. Given the large amount of malicious files that are obfuscated or packed, it would be a good idea to offload part of this capability to the GPU to reduce the resource cost on the CPU. This would require research into the possible techniques that would and would not work on a GPU.

Lastly, the GPU ID system could be applied to network traffic, by programming it to look for the signatures of network attacks or network security problems. The system may be capable of processing a large amount of network traffic at a gateway, refining the results, and presenting a network administrator with a clear picture of the state of the network. This would help to detect and begin mitigation steps on reducing a cyber attack to a government network.

VI. Experimental Data

This appendix contains the raw data collected during the experiments. Section A.1 contains data from Experiment 1. Section A.2 contains data from Experiment 2. Section A.3 contains data from Experiment 3.

6.1 Experimental Data of Experiment 1

The means are only presented here because of space requirements. All means are in milliseconds (ms). In Table F.1, Small Benign is abbreviated as SB, Large Benign is abbreviated as LB, Small Malicious is abbreviated as SM, and Large Malicious is abbreviated as LM.

		Mean Data (ms)			
Threads Per Block	Events	SB	LB	SM	LM
1	50	0.033985632	0.03426362	0.033941116	0.03386122
2	50	0.024770054	0.02633144	0.024567304	0.02775737
3	50	0.02146952	0.023717544	0.022855154	0.023036406
4	50	0.02143729	0.021402952	0.020912238	0.021390478
5	50	0.021107526	0.021434282	0.02046363	0.022841732
6	50	0.020908478	0.020450438	0.020710486	0.02142827
7	50	0.02098051	0.020989078	0.021104568	0.021149928
8	50	0.020778806	0.020978852	0.020881814	0.020172312
9	50	0.020399214	0.021491978	0.020341814	0.020234104
10	50	0.020644472	0.020459768	0.020754848	0.020339568
11	50	0.020426588	0.020490344	0.020972182	0.021029576
12	50	0.021197692	0.02042763	0.021738732	0.021024724
13	50	0.020282728	0.020209442	0.021089826	0.020466236
14	50	0.020611788	0.01997717	0.0205946	0.020830592
15	50	0.020198314	0.019893512	0.020249248	0.020055668
16	50	0.019802988	0.020293048	0.019870952	0.019606344
17	50	0.019858528	0.019273772	0.019324004	0.019109514
18	50	0.020410042	0.020266434	0.019578122	0.019221042
19	50	0.019402588	0.019754064	0.019947096	0.019210864
20	50	0.020245436	0.01983577	0.019621084	0.019881478
21	50	0.019178236	0.01881774	0.019101596	0.01981011
22	50	0.019125856	0.019909998	0.018790476	0.019540692
23	50	0.019078986	0.018910972	0.018973938	0.018990472
24	50	0.019130666	0.018930624	0.01928806	0.018975634
25	50	0.01881148	0.01919668	0.019226652	0.018842604
26	50	0.019184746	0.01939332	0.019828444	0.019338186
27	50	0.01912199	0.0191057	0.019224354	0.01841263
28	50	0.018840092	0.019307048	0.0189014	0.018648022
29	50	0.01912014	0.019517532	0.019237928	0.019173322
30	50	0.01925458	0.01880867	0.018928716	0.01874967
31	50	0.019278288	0.019269958	0.019470868	0.019555474
32	50	0.019347658	0.01942415	0.019012876	0.019566546
33	50	0.018987412	0.018896734	0.018698344	0.019100894
34	50	0.018985506	0.018873638	0.01901964	0.019116236

Continued on next page

Table F.1 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
35	50	0.018474938	0.018807764	0.01933377	0.019017888
36	50	0.01974956	0.019304652	0.019031874	0.018930024
37	50	0.019772658	0.019388004	0.018811574	0.019146404
38	50	0.019217232	0.019132726	0.018910922	0.019830554
39	50	0.019084954	0.01953678	0.01891142	0.018779894
40	50	0.017265646	0.017934948	0.016891366	0.017213006
41	50	0.016766908	0.017424684	0.01674239	0.01744869
42	50	0.016866396	0.017391056	0.017645432	0.018334398
43	50	0.017277868	0.016619846	0.017479426	0.017422436
44	50	0.016780836	0.016931156	0.016758326	0.017161074
45	50	0.016753668	0.016913018	0.017148498	0.017197716
46	50	0.01679051	0.017112162	0.017191012	0.016796326
47	50	0.016824398	0.017223234	0.016843944	0.016996368
48	50	0.016774574	0.017133512	0.01652756	0.017185598
49	50	0.016811108	0.017147956	0.016657684	0.017002286
50	50	0.016719484	0.017177476	0.017104638	0.016417642
51	50	0.016696678	0.018059806	0.01722178	0.016901742
52	50	0.01681542	0.017190006	0.016990254	0.01737341
53	50	0.0169062	0.01722223	0.017060732	0.018437048
54	50	0.016844998	0.0169252	0.016820536	0.017468996
55	50	0.01683046	0.017805928	0.016634974	0.017192154
56	50	0.017104738	0.017198978	0.01680209	0.018389132
57	50	0.016657782	0.017042586	0.01703096	0.01707316
58	50	0.017102534	0.017256268	0.017387594	0.016786296
59	50	0.01703607	0.01696971	0.017316614	0.017304534
60	50	0.016724298	0.017352352	0.016871508	0.016722892
61	50	0.016681236	0.017421374	0.01704986	0.016655078
62	50	0.017236814	0.01764158	0.01672264	0.016917132
63	50	0.017196066	0.017377212	0.016808002	0.01746383
64	50	0.017214762	0.016770518	0.016794564	0.018113192
65	50	0.01680931	0.016892062	0.016803286	0.017283086
66	50	0.016811262	0.017027752	0.0168065	0.017265542
67	50	0.016819032	0.017361326	0.017284186	0.017216122
68	50	0.01728208	0.017268246	0.017079472	0.017088854
69	50	0.017318368	0.01707301	0.016843992	0.01692409
70	50	0.016934568	0.017432152	0.016757832	0.01725371
71	50	0.016779286	0.01716664	0.016689814	0.016772014
72	50	0.016741638	0.017353508	0.016904438	0.016584298
73	50	0.016560446	0.017549096	0.017235566	0.017241134
74	50	0.01697853	0.017340424	0.017556062	0.01787651
75	50	0.017638118	0.017171058	0.017370146	0.01804647
76	50	0.01688134	0.017148902	0.01699362	0.01690475
77	50	0.016964446	0.017178426	0.016593824	0.017478878
78	50	0.017070356	0.017193804	0.017059468	0.017114764
79	50	0.01663999	0.017196972	0.016782042	0.016898422
80	50	0.016987694	0.017140426	0.016728654	0.016725296
81	50	0.017217116	0.017082238	0.016624352	0.017135364
82	50	0.016820286	0.016950354	0.017101586	0.01681397
83	50	0.016745352	0.017617464	0.017050158	0.016820194
84	50	0.017116026	0.017991696	0.01687672	0.017286838
85	50	0.016893768	0.017358322	0.016960984	0.01816206
86	50	0.016895674	0.01714714	0.016602044	0.017561172
87	50	0.016664446	0.016708562	0.01686169	0.017306688
88	50	0.016890206	0.017131906	0.016970152	0.017606136

Continued on next page

Table F.1 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
89	50	0.01650931	0.017111456	0.017193208	0.016832464
90	50	0.017247242	0.017575406	0.017577064	0.01691071
91	50	0.017021232	0.016984794	0.017507748	0.01717923
92	50	0.01695733	0.017216272	0.016917276	0.017898256
93	50	0.016687704	0.01708138	0.01668215	0.01697627
94	50	0.01814006	0.017638514	0.018093792	0.016792516
95	50	0.016970408	0.017698314	0.017233514	0.017594354
96	50	0.01709246	0.016905242	0.01690635	0.018034002
97	50	0.016816924	0.016915014	0.016531872	0.017182188
98	50	0.016525754	0.017017124	0.01685026	0.017338068
99	50	0.016752122	0.017511902	0.017080328	0.017206296
100	50	0.01729872	0.017163936	0.016780034	0.016980584
101	50	0.016861638	0.016823904	0.017028604	0.016664546
102	50	0.016752912	0.017708842	0.016865798	0.016992062
103	50	0.016491528	0.017299572	0.016851264	0.01725211
104	50	0.01666881	0.017066596	0.016930018	0.016903946
105	50	0.01645588	0.018006528	0.01707086	0.0172693
106	50	0.01704449	0.017331656	0.017388744	0.017999208
107	50	0.017034716	0.017082486	0.01746819	0.017684636
108	50	0.016646806	0.017147742	0.01695331	0.017582726
109	50	0.016792366	0.016989252	0.01705337	0.01737902
110	50	0.016932314	0.017733402	0.017123836	0.017208
111	50	0.016528108	0.017111762	0.017106338	0.016885094
112	50	0.017171354	0.016921136	0.016916974	0.016999626
113	50	0.01703842	0.01695933	0.016689804	0.016933372
114	50	0.01668791	0.017172158	0.017303086	0.017385692
115	50	0.016929856	0.017491346	0.0170672	0.01687056
116	50	0.017146346	0.017534654	0.017242588	0.017781168
117	50	0.016913112	0.017366786	0.01698685	0.018073598
118	50	0.016731216	0.017225792	0.016670008	0.01715366
119	50	0.016579638	0.01701617	0.017110906	0.017615712
120	50	0.016732056	0.017519564	0.016744902	0.017337672
121	50	0.01663718	0.017503922	0.017037426	0.0171042
122	50	0.017097472	0.016869662	0.017668434	0.016910058
123	50	0.017297176	0.01722664	0.01728545	0.016736074
124	50	0.017679064	0.016964352	0.017082092	0.016690608
125	50	0.0169042	0.017222382	0.016674578	0.017319168
126	50	0.017146198	0.017426442	0.016809912	0.016876768
127	50	0.016697226	0.017719518	0.016954464	0.017937452
128	50	0.017001836	0.017630098	0.016800194	0.01788007
129	50	0.017156572	0.016968702	0.016566548	0.017213104
130	50	0.016762184	0.016941338	0.017053106	0.017532648
131	50	0.01665137	0.017452408	0.017739668	0.01729121
132	50	0.01705753	0.017382632	0.017002078	0.01704885
133	50	0.016908954	0.017113408	0.016736734	0.016864742
134	50	0.016917882	0.017000128	0.016817526	0.016634878
135	50	0.017088842	0.017063038	0.016856878	0.017575304
136	50	0.016810614	0.017346388	0.017043342	0.017510454
137	50	0.016709312	0.017096964	0.01713517	0.017722524
138	50	0.017481432	0.0177331	0.017254164	0.017799718
139	50	0.017229654	0.017355312	0.017590844	0.017193308
140	50	0.016704096	0.01732764	0.016935724	0.017315118
141	50	0.016774774	0.017420376	0.01675022	0.017429348
142	50	0.016874366	0.017253006	0.017152956	0.017124738

Continued on next page

Table F.1 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
143	50	0.016694022	0.01687382	0.016890812	0.017167504
144	50	0.017065994	0.016799582	0.016764602	0.017859012
145	50	0.017080488	0.017129604	0.016535982	0.0171873
146	50	0.016640938	0.01724253	0.016848006	0.016974572
147	50	0.016454976	0.017372006	0.016901838	0.017512406
148	50	0.017354512	0.016956074	0.017328854	0.017816156
149	50	0.017171852	0.01747396	0.016780588	0.016997776
150	50	0.016834182	0.017332508	0.01666214	0.017337566
151	50	0.016931856	0.01746939	0.016715328	0.017399622
152	50	0.016645254	0.017376664	0.0167662	0.017559966
153	50	0.01662134	0.016786544	0.017116678	0.016903896
154	50	0.016866152	0.01824502	0.017519018	0.017108452
155	50	0.01711321	0.017104646	0.01758718	0.016849458
156	50	0.01696534	0.017320172	0.016629612	0.016794522
157	50	0.016975524	0.01783806	0.017083788	0.016995522
158	50	0.017027552	0.017364632	0.017059582	0.018720844
159	50	0.016753622	0.017161986	0.017023738	0.017759114
160	50	0.016745846	0.01732133	0.016956776	0.016958726
161	50	0.017187042	0.017571354	0.016572876	0.01867523
162	50	0.01681132	0.017557814	0.017345038	0.017420628
163	50	0.016809064	0.016702642	0.016855368	0.017184036
164	50	0.017714608	0.016917978	0.01698294	0.016863446
165	50	0.017047444	0.017011362	0.016996672	0.017010864
166	50	0.017140784	0.017061094	0.01698389	0.01681472
167	50	0.01667157	0.017479616	0.01685267	0.01698229
168	50	0.016956174	0.016897928	0.017008548	0.01731837
169	50	0.016693572	0.017275212	0.01700805	0.018634878
170	50	0.017484542	0.016903796	0.017560622	0.017194706
171	50	0.017006248	0.017647084	0.017339378	0.01708915
172	50	0.017284036	0.017537816	0.01679913	0.017370948
173	50	0.016754924	0.016912312	0.016858274	0.016963444
174	50	0.016954268	0.01694009	0.017246502	0.017231658
175	50	0.016743296	0.016910314	0.016691216	0.016929602
176	50	0.01699838	0.017091056	0.016498792	0.016868302
177	50	0.017030764	0.017129204	0.01653728	0.016853522
178	50	0.01678696	0.016888108	0.017159376	0.017047996
179	50	0.0167459	0.016985494	0.017200578	0.017609042
180	50	0.017215464	0.01721812	0.016901584	0.018163518
181	50	0.017098774	0.017286592	0.016613572	0.016992962
182	50	0.017136066	0.017556868	0.016738982	0.017176462
183	50	0.016734376	0.01780663	0.01695783	0.017140176
184	50	0.016667504	0.017058578	0.017012868	0.017022888
185	50	0.016697726	0.017370654	0.017099674	0.017203442
186	50	0.01728494	0.016953916	0.017243786	0.016916776
187	50	0.017141934	0.017247898	0.017572254	0.016963646
188	50	0.017235166	0.017446136	0.016854364	0.016804396
189	50	0.017133118	0.01695036	0.017105896	0.017359614
190	50	0.01684635	0.017317866	0.016826498	0.017803926
191	50	0.017266388	0.017323478	0.017078168	0.017841724
192	50	0.017023138	0.017478564	0.01666575	0.01706379
193	50	0.017042686	0.017727084	0.017231558	0.017061082
194	50	0.016630764	0.017911188	0.017155114	0.017035866
195	50	0.016880632	0.017515654	0.01671497	0.016684692
196	50	0.01732082	0.017040984	0.01667192	0.017409098

Continued on next page

Table F.1 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
197	50	0.017104488	0.017065846	0.017028454	0.017432752
198	50	0.016862942	0.017336164	0.0170082	0.017175714
199	50	0.016695022	0.017161236	0.016747252	0.016987852
200	50	0.016769008	0.016754868	0.016758178	0.01768242
201	50	0.01677071	0.016962234	0.016757686	0.017751246
202	50	0.016852262	0.017107994	0.01758824	0.017372308
203	50	0.01710635	0.017299728	0.017322384	0.017438114
204	50	0.017231058	0.01808332	0.016807806	0.017159478
205	50	0.017117372	0.017513102	0.017142988	0.017035416
206	50	0.016763152	0.016972458	0.01702068	0.016983292
207	50	0.01705096	0.017150852	0.016568006	0.017422778
208	50	0.017361178	0.01712875	0.016777332	0.016811766
209	50	0.01666966	0.01757516	0.01705608	0.017079368
210	50	0.016789866	0.016915528	0.016929564	0.017270054
211	50	0.016920328	0.017125242	0.016868802	0.017802018
212	50	0.016944944	0.017044588	0.016909206	0.01746508
213	50	0.017294814	0.017378322	0.01690585	0.017439578
214	50	0.017054814	0.017496662	0.016738184	0.017324182
215	50	0.01710614	0.01764032	0.016967746	0.017170608
216	50	0.017101078	0.01726719	0.017182886	0.017242038
217	50	0.017150204	0.01712524	0.016783082	0.017098022
218	50	0.016975718	0.017238226	0.017210344	0.017162634
219	50	0.017693306	0.016961228	0.017258532	0.01700644
220	50	0.017153866	0.017136918	0.017099734	0.017389646
221	50	0.01684324	0.017084948	0.016982236	0.017345744
222	50	0.017096814	0.01719301	0.016756018	0.018183814
223	50	0.016738826	0.01720925	0.016594372	0.017097022
224	50	0.01649011	0.017693956	0.016875626	0.017540522
225	50	0.01652139	0.01735611	0.016938682	0.017137376
226	50	0.016850258	0.017038974	0.017141624	0.016719436
227	50	0.016798078	0.017432504	0.016854524	0.01687096
228	50	0.017235066	0.017248402	0.016711062	0.01693472
229	50	0.016972956	0.017416012	0.017162382	0.017205536
230	50	0.01711957	0.017572102	0.016986036	0.016904448
231	50	0.017000934	0.017317708	0.017011958	0.01716489
232	50	0.01715547	0.017301928	0.016967246	0.017855806
233	50	0.016794818	0.01715877	0.016781936	0.017315312
234	50	0.016959074	0.017151408	0.017209302	0.016932258
235	50	0.01711642	0.0178332	0.017295808	0.017844284
236	50	0.016989554	0.01712855	0.017131464	0.017108706
237	50	0.016879322	0.01771786	0.018177446	0.01684505
238	50	0.017089202	0.017471048	0.016855766	0.016966092
239	50	0.01694308	0.017197416	0.016798682	0.016764192
240	50	0.016968508	0.017361884	0.017110706	0.017421678
241	50	0.01698705	0.016960884	0.016698334	0.017310952
242	50	0.017162888	0.016911356	0.016553222	0.017772146
243	50	0.016983086	0.016961186	0.01662861	0.017810044
244	50	0.017268152	0.017025246	0.017100176	0.01729626
245	50	0.017038272	0.01715527	0.017388094	0.019869204
246	50	0.016860932	0.018220006	0.01699757	0.017577514
247	50	0.017031458	0.01762995	0.017049	0.016815426
248	50	0.01672234	0.017791944	0.016558594	0.016671012
249	50	0.016935024	0.017846982	0.01707196	0.018125526
250	50	0.017323932	0.01758258	0.017487738	0.01712559

Continued on next page

Table F.1 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
251	50	0.017217468	0.016989764	0.01757701	0.017865326
252	50	0.01716955	0.016695224	0.017186088	0.017419122
253	50	0.017134214	0.0181975	0.016835474	0.017953994
254	50	0.017037566	0.017171252	0.016666554	0.017429546
255	50	0.017094312	0.017557666	0.017085852	0.01705732
256	50	0.017031062	0.01777706	0.016964392	0.017246836

Table F.1: Optimal Number of Threads Per Block for Experiment 1.

6.2 Experimental Data of Experiment 2

The means are only presented here because of space requirements. All means are in milliseconds (ms). In Table F.2, Small Benign is abbreviated as SB, Large Benign is abbreviated as LB, Small Malicious is abbreviated as SM, and Large Malicious is abbreviated as LM.

		Mean Data (ms)			
Threads Per Block	Events	SB	LB	SM	LM
256	50	65.315242	95.82758	65.532198	95.82758
257	50	67.557452	98.073684	67.756028	98.073684
258	50	67.302384	97.742796	67.494524	97.742796
259	50	67.136188	97.64751	67.321912	97.64751
260	50	66.999628	97.568236	67.204166	97.568236
261	50	67.032302	97.496492	67.217302	97.496492
262	50	66.671286	97.10458	66.850304	97.10458
263	50	66.478972	96.989086	66.690924	96.989086
264	50	66.43307	96.880842	66.635874	96.880842
265	50	66.136344	96.759748	66.36113	96.759748
266	50	66.112208	96.642652	66.28433	96.642652
267	50	66.06874	96.572778	66.264904	96.572778
268	50	65.987558	96.51462	66.219234	96.51462
269	50	65.601828	96.087762	65.759404	96.087762
270	50	65.427296	95.830512	65.58991	95.830512
271	50	65.425154	95.868418	65.62395	95.868418
272	50	65.011944	95.53882	65.23428	95.53882
273	50	66.101154	96.676564	66.338932	96.676564
274	50	67.460226	97.908978	67.597926	97.908978
275	50	65.96239	96.489398	66.227466	96.489398
276	50	66.2712	96.72472	66.459178	96.72472
277	50	66.323886	96.823126	66.512492	96.823126
278	50	65.338768	95.815034	65.585032	95.815034
279	50	65.912806	96.205762	65.8999	96.205762
280	50	65.372304	95.86751	65.529636	95.86751
281	50	65.088562	95.622568	65.277294	95.622568
282	50	64.9428	95.445018	65.160282	95.445018
283	50	64.885304	95.3851	65.117222	95.3851

Continued on next page

Table F.2 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
284	50	64.87458	95.371966	65.071016	95.371966
285	50	65.323522	95.768272	65.51569	95.768272
286	50	64.229158	94.771724	64.422972	94.771724
287	50	65.199404	95.616952	65.287358	95.616952
288	50	64.390472	94.893002	64.514688	94.893002
289	50	66.300652	96.864924	66.493224	96.864924
290	50	66.08282	96.60152	66.28701	96.60152
291	50	65.94644	96.43864	66.093736	96.43864
292	50	65.547944	96.039588	65.753258	96.039588
293	50	65.399924	95.965816	65.612612	95.965816
294	50	65.475452	95.935952	65.68686	95.935952
295	50	65.284866	95.76794	65.443076	95.76794
296	50	65.12121	95.683188	65.320462	95.683188
297	50	65.1475	95.620014	65.339908	95.620014
298	50	64.860512	95.35295	65.059598	95.35295
299	50	64.864318	95.401916	65.047534	95.401916
300	50	64.705052	95.157358	64.919576	95.157358
301	50	64.51116	95.033116	64.690658	95.033116
302	50	64.312096	94.848932	64.558292	94.848932
303	50	64.209418	94.621436	64.38068	94.621436
304	50	64.2339	94.720758	64.418568	94.720758
305	50	64.959198	95.390044	65.13455	95.390044
306	50	64.94553	95.466592	65.140472	95.466592
307	50	64.779194	95.24051	64.946382	95.24051
308	50	64.596276	95.092078	64.793706	95.092078
309	50	64.663138	95.10945	64.862446	95.10945
310	50	64.393858	94.93793	64.602902	94.93793
311	50	64.310286	94.78691	64.52489	94.78691
312	50	64.254248	94.716652	64.440542	94.716652
313	50	64.079022	94.512206	64.28607	94.512206
314	50	64.168322	94.612736	64.352884	94.612736
315	50	63.825052	94.310496	64.01627	94.310496
316	50	63.666558	94.186396	63.885118	94.186396
317	50	64.083692	94.663736	64.243912	94.663736
318	50	63.512436	93.974344	63.689616	93.974344
319	50	63.469924	93.915186	63.603016	93.915186
320	50	63.14337	93.593726	63.330578	93.593726
321	50	65.370584	95.773888	65.493868	95.773888
322	50	65.173384	95.69544	65.383504	95.69544
323	50	65.053894	95.579412	65.254584	95.579412
324	50	64.97242	95.45083	65.188516	95.45083
325	50	64.798452	95.189308	65.002704	95.189308
326	50	64.685892	95.186118	64.856922	95.186118
327	50	64.574676	95.119612	64.76029	95.119612
328	50	64.42804	94.903306	64.606516	94.903306
329	50	64.401846	94.837634	64.55703	94.837634
330	50	64.227266	94.71826	64.452038	94.71826
331	50	64.108818	94.630516	64.266516	94.630516
332	50	63.914188	94.452776	64.124716	94.452776
333	50	63.847954	94.328866	64.054416	94.328866
334	50	63.770588	94.300128	63.946838	94.300128
335	50	63.606924	94.147938	63.83268	94.147938
336	50	63.472638	93.895482	63.66446	93.895482
337	50	64.251736	94.741706	64.419198	94.741706

Continued on next page

Table F.2 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
338	50	64.088538	94.662294	64.353996	94.662294
339	50	64.055002	94.583154	64.237048	94.583154
340	50	63.869338	94.414286	64.03989	94.414286
341	50	63.791504	94.31332	63.988828	94.31332
342	50	63.655708	94.166006	63.797328	94.166006
343	50	63.555204	94.010008	63.704816	94.010008
344	50	63.432052	93.92824	63.645416	93.92824
345	50	63.406742	93.86474	63.619936	93.86474
346	50	63.215298	93.688868	63.412912	93.688868
347	50	63.106672	93.647592	63.268336	93.647592
348	50	62.991786	93.51893	63.215994	93.51893
349	50	63.118286	93.61704	63.291878	93.61704
350	50	62.751586	93.231936	62.944704	93.231936
351	50	62.661338	93.133198	62.841844	93.133198
352	50	62.530526	93.021826	62.674158	93.021826
353	50	64.676048	95.148926	64.85002	95.148926
354	50	64.429994	95.014716	64.61851	95.014716
355	50	64.331802	94.848504	64.519016	94.848504
356	50	64.328104	94.738078	64.495294	94.738078
357	50	64.198052	94.71588	64.378088	94.71588
358	50	64.02508	94.53461	64.24125	94.53461
359	50	64.021872	94.487848	64.216606	94.487848
360	50	63.73946	94.280746	63.96648	94.280746
361	50	63.830722	94.286746	63.993664	94.286746
362	50	63.645776	94.169028	63.829334	94.169028
363	50	63.699612	94.11661	63.925296	94.11661
364	50	63.435804	93.921762	63.609254	93.921762
365	50	63.285568	93.781378	63.433412	93.781378
366	50	63.132746	93.656992	63.335134	93.656992
367	50	63.065618	93.558216	63.249596	93.558216
368	50	63.012422	93.566222	63.196512	93.566222
369	50	63.691096	94.168848	63.899078	94.168848
370	50	63.558424	93.969266	63.743788	93.969266
371	50	63.442128	93.931194	63.629316	93.931194
372	50	63.325536	93.765742	63.494702	93.765742
373	50	63.21567	93.69636	63.405478	93.69636
374	50	63.0661	93.59342	63.31403	93.59342
375	50	62.992054	93.500614	63.179002	93.500614
376	50	62.943298	93.368722	63.099856	93.368722
377	50	62.92936	93.354962	63.087716	93.354962
378	50	62.707368	93.195364	62.91014	93.195364
379	50	62.592104	93.182356	62.796876	93.182356
380	50	62.525594	93.03382	62.728872	93.03382
381	50	62.399842	92.84631	62.59874	92.84631
382	50	62.310162	92.735986	62.508854	92.735986
383	50	62.278196	92.774822	62.453384	92.774822
384	50	62.12574	92.611128	62.299898	92.611128
385	50	64.101746	94.735632	64.34411	94.735632
386	50	63.988914	94.502162	64.198688	94.502162
387	50	63.935596	94.419648	64.092784	94.419648
388	50	63.826088	94.339022	64.040514	94.339022
389	50	63.759188	94.218724	63.968312	94.218724
390	50	63.664824	94.16959	63.82966	94.16959
391	50	63.488544	94.014088	63.743044	94.014088

Continued on next page

Table F.2 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
392	50	63.46659	93.920026	63.628742	93.920026
393	50	63.444748	93.899484	63.576378	93.899484
394	50	63.299156	93.802126	63.45515	93.802126
395	50	63.16238	93.609504	63.374006	93.609504
396	50	63.089912	93.575938	63.247104	93.575938
397	50	62.943582	93.436866	63.09315	93.436866
398	50	62.874564	93.287234	63.0362	93.287234
399	50	62.836612	93.31493	62.995902	93.31493
400	50	62.67531	93.185296	62.89574	93.185296
401	50	63.236386	93.732246	63.451574	93.732246
402	50	63.160856	93.66701	63.36876	93.66701
403	50	63.044694	93.579368	63.273718	93.579368
404	50	62.922228	93.39722	63.080496	93.39722
405	50	62.868514	93.430324	63.057376	93.430324
406	50	62.780672	93.276422	62.991924	93.276422
407	50	62.656542	93.185818	62.89401	93.185818
408	50	62.603488	93.075668	62.813334	93.075668
409	50	62.529432	92.96498	62.7263	92.96498
410	50	62.468682	92.933612	62.715838	92.933612
411	50	62.31576	92.922064	62.528106	92.922064
412	50	62.271434	92.765148	62.43548	92.765148
413	50	62.10282	92.630796	62.288746	92.630796
414	50	62.048958	92.56442	62.240122	92.56442
415	50	61.980828	92.396708	62.182354	92.396708
416	50	61.898248	92.390418	62.07067	92.390418
417	50	63.757822	94.199252	64.011126	94.199252
418	50	63.747238	94.193492	63.926652	94.193492
419	50	63.63372	94.140228	63.85627	94.140228
420	50	63.524168	94.033878	63.729844	94.033878
421	50	63.39298	93.876492	63.557572	93.876492
422	50	63.297534	93.851686	63.505244	93.851686
423	50	63.253914	93.732274	63.45577	93.732274
424	50	63.26186	93.729204	63.445584	93.729204
425	50	63.098382	93.593184	63.326354	93.593184
426	50	63.004078	93.521806	63.22156	93.521806
427	50	62.945276	93.348684	63.090242	93.348684
428	50	62.844156	93.337032	63.038864	93.337032
429	50	62.713314	93.149962	62.912086	93.149962
430	50	62.717388	93.179434	62.901272	93.179434
431	50	62.540686	93.010102	62.725642	93.010102
432	50	62.46921	92.936708	62.667358	92.936708
433	50	62.916698	93.397638	63.123184	93.397638
434	50	62.864822	93.388858	63.08192	93.388858
435	50	62.777958	93.326872	62.989626	93.326872
436	50	62.688196	93.108924	62.839966	93.108924
437	50	62.64639	93.122962	62.802934	93.122962
438	50	62.60318	92.986972	62.744756	92.986972
439	50	62.43262	92.866108	62.597068	92.866108
440	50	62.410448	92.863992	62.565716	92.863992
441	50	62.222272	92.775338	62.450836	92.775338
442	50	62.223828	92.681492	62.39524	92.681492
443	50	62.183922	92.673244	62.334622	92.673244
444	50	62.028716	92.590634	62.21471	92.590634
445	50	61.972136	92.49041	62.13308	92.49041

Continued on next page

Table F.2 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
446	50	61.856486	92.32328	62.051722	92.32328
447	50	61.73686	92.314742	61.953212	92.314742
448	50	61.664184	92.18511	61.910536	92.18511
449	50	63.56632	94.109006	63.755468	94.109006
450	50	63.470874	93.901104	63.589456	93.901104
451	50	63.35012	93.891086	63.563962	93.891086
452	50	63.214684	93.716108	63.411672	93.716108
453	50	63.17108	93.723206	63.350994	93.723206
454	50	63.151602	93.686376	63.33025	93.686376
455	50	62.978232	93.388168	63.170262	93.388168
456	50	62.892108	93.403178	63.099096	93.403178
457	50	62.829068	93.308392	62.996032	93.308392
458	50	62.809028	93.315524	62.988626	93.315524
459	50	62.652454	93.153136	62.83862	93.153136
460	50	62.515168	93.080182	62.702172	93.080182
461	50	62.470394	92.91045	62.660478	92.91045
462	50	62.432032	92.96542	62.61926	92.96542
463	50	62.394368	92.921378	62.611048	92.921378
464	50	62.255846	92.867358	62.459372	92.867358
465	50	62.76694	93.287336	62.971302	93.287336
466	50	62.631706	93.22226	62.79788	93.22226
467	50	62.531724	93.008108	62.726274	93.008108
468	50	62.519618	92.997274	62.718062	92.997274
469	50	62.422652	92.949038	62.632436	92.949038
470	50	62.411128	92.898304	62.609642	92.898304
471	50	62.306274	92.749788	62.429138	92.749788
472	50	62.178252	92.691684	62.327106	92.691684
473	50	62.145268	92.650702	62.298736	92.650702
474	50	62.007194	92.546266	62.203042	92.546266
475	50	61.946034	92.471782	62.147474	92.471782
476	50	61.86966	92.399158	62.060872	92.399158
477	50	61.818658	92.274294	62.013942	92.274294
478	50	61.751688	92.279796	61.89728	92.279796
479	50	61.696632	92.222286	61.889994	92.222286
480	50	61.66354	92.136778	61.845024	92.136778
481	50	63.237168	93.6948	63.44131	93.6948
482	50	63.099736	93.532616	63.274786	93.532616
483	50	62.984074	93.561094	63.202538	93.561094
484	50	63.010254	93.559472	63.196944	93.559472
485	50	62.911478	93.370686	63.094124	93.370686
486	50	62.79191	93.336546	62.963298	93.336546
487	50	62.707348	93.26485	62.91257	93.26485
488	50	62.702436	93.175306	62.924982	93.175306
489	50	62.65132	93.124708	62.874196	93.124708
490	50	62.55083	93.069484	62.734496	93.069484
491	50	62.403118	92.943548	62.64749	92.943548
492	50	62.278298	92.869124	62.537142	92.869124
493	50	62.351302	92.874166	62.570412	92.874166
494	50	62.218724	92.691466	62.390366	92.691466
495	50	62.162724	92.677014	62.330142	92.677014
496	50	62.111538	92.575016	62.323948	92.575016
497	50	62.633926	93.09796	62.815728	93.09796
498	50	62.486764	92.999614	62.687048	92.999614
499	50	62.359592	92.802046	62.51899	92.802046

Continued on next page

Table F.2 – continued from previous page

Threads Per Block	Events	SB	LB	SM	LM
500	50	62.308164	92.77276	62.49983	92.77276
501	50	62.213168	92.667128	62.38126	92.667128
502	50	62.1551	92.617644	62.299024	92.617644
503	50	62.078394	92.46834	62.237738	92.46834
504	50	62.049758	92.547482	62.21845	92.547482
505	50	61.914326	92.46992	62.107588	92.46992
506	50	61.867752	92.368892	62.101636	92.368892
507	50	61.73346	92.227582	61.950372	92.227582
508	50	61.756856	92.204648	61.944842	92.204648
509	50	61.748814	92.191808	61.926304	92.191808
510	50	61.568434	92.105052	61.78038	92.105052
511	50	61.496866	92.047068	61.651456	92.047068
512	50	61.41264	91.854308	61.60497	91.854308

Table F.2: Optimal Number of Threads Per Block for Experiment2.

6.3 Experimental Data of Experiment 3

All data is in milliseconds (ms). In Table F.3, Small Benign is abbreviated as SB, Large Benign is abbreviated as LB, Small Malicious is abbreviated as SM, and Large Malicious is abbreviated as LM.

Events	SB CPU	LB CPU	SM CPU	LM CPU	SB GPU	LB GPU	SM GPU	LM GPU
1	314.248	636.847	312.474	623.717	56.8802	92.1577	56.7218	91.5893
2	317.053	637.315	315.446	622.332	57.0006	92.3109	57.0752	92.504
3	321.086	631.214	315.459	624.36	56.9189	92.581	56.8204	92.7973
4	316.816	634.637	316.574	620.267	57.0405	93.2986	56.839	90.7832
5	317.277	639.881	318.111	627.557	56.9328	91.5904	56.7066	90.0244
6	320.972	633.035	315.461	622.042	56.9852	93.6586	56.754	92.1479
7	319.679	639.744	316.05	625.443	56.7821	92.5316	56.8074	92.8506
8	317.541	640.488	316.297	622.254	56.7744	93.2836	56.5845	91.8376
9	317.681	639.032	315.577	620.971	57.0198	92.7988	56.769	90.5407
10	317.859	634.569	316.711	627.648	56.8579	92.3656	56.8629	91.2413
11	317.999	634.839	315.283	624.99	56.8328	93.9353	56.7711	93.3886
12	319.088	634.002	314.454	627.789	57.0334	92.8371	56.9185	93.1833
13	316.653	636.847	316.565	628.592	56.9617	93.5508	56.7978	91.7595
14	315.651	640.689	317.272	632.158	57.0402	94.0533	56.8191	93.3791
15	320.555	639.851	312.474	622.269	56.9277	94.379	57.278	92.6727
16	320.015	635.18	314.463	627.684	56.9486	92.6155	56.8097	92.7205
17	316.95	633.241	314.876	628.423	57.0859	94.75	56.7299	91.2154
18	316.204	637.137	317.623	623.619	56.9034	93.7253	56.6835	92.1806
19	320.228	636.769	313.625	622.848	56.7484	92.92	56.7581	91.7458
20	317.53	628.466	313.932	626.35	57.08	92.7985	56.903	93.5025
21	318.009	638.32	313.912	624.68	57.0178	94.58	56.7351	91.6921
22	319.448	636.382	313.579	629.376	57.011	93.2763	56.8623	94.1166
23	318.237	632.51	313.862	624.223	56.9086	93.3404	56.807	92.6381
24	317.978	640.774	316.191	627.215	57.0348	93.5992	56.803	91.0175

Continued on next page

Table F.3 – continued from previous page

Events	SB CPU	LB CPU	SM CPU	LM CPU	SB GPU	LB GPU	SM GPU	LM GPU
25	314.248	638.636	313.67	625.215	56.9855	92.515	56.7565	93.9315
26	319.499	644.978	314.623	623.717	57.0095	94.3517	56.9475	93.2017
27	320.223	636.911	314.36	621.843	57.0354	92.8553	56.8606	94.4907
28	317.371	639.348	313.546	628.074	56.9845	94.2184	56.7861	90.834
29	319.228	639.521	316.068	616.857	56.9302	94.1694	56.6356	90.4273
30	320.119	634.829	317.73	620.787	56.9857	93.1342	56.7136	91.01
31	318.409	637.068	319.5	617.947	56.9044	93.9552	56.9167	90.6423
32	317.316	634.555	317.14	620.298	56.9988	91.2111	56.9387	90.9696
33	321.855	634.985	316.503	631.721	56.971	93.6552	56.613	91.2384
34	319.228	629.789	315.411	622.921	56.8139	94.5369	56.7056	91.348
35	318.903	633.934	314.54	616.182	56.9161	94.6496	56.8231	93.6968
36	317.669	634.349	317.046	626.144	56.8856	93.5987	56.8026	92.2297
37	321.163	634.17	318.151	624.959	56.8225	93.9171	56.733	91.1016
38	315.424	638.834	313.442	621.492	57.0079	91.92	56.6338	90.8833
39	315.833	641.672	314.662	624.453	56.9047	93.4767	56.8077	91.4768
40	319.039	640.15	319.942	620.65	56.8323	91.8321	56.6708	90.2705
41	320.296	639.799	314.189	619.895	56.9453	91.1203	56.8006	91.4463
42	320.376	636.582	315.906	628.11	56.8566	92.2345	56.8921	92.7863
43	318.237	634.554	312.548	622.961	56.9028	93.1009	56.6661	91.1957
44	316.822	630.137	314.653	625.054	56.5871	92.8612	56.8528	92.4402
45	319.414	636.068	313.53	623.481	56.5444	93.0483	56.6442	91.7361
46	315.742	641.039	318.273	623.787	56.5848	94.568	56.69	90.9666
47	315.959	637.581	313.091	623.641	56.5931	94.1953	56.8295	91.1006
48	317.895	637.443	316.228	630.141	56.7814	92.494	56.9042	93.3762
49	316.357	634.062	314.291	622.091	56.6786	94.5461	56.8021	94.161
50	318.103	642.219	316.689	624.703	56.57	95.6105	56.6916	91.6719
51	321.446	640.849	317.301	623.924	56.5119	94.5341	56.7481	91.5844
52	317.526	628.831	314.052	622.871	56.5606	91.7812	56.8866	91.1363
53	317.038	633.643	317.033	623.766	56.8693	92.6903	56.6355	92.0767
54	316.941	634.186	315.224	622.48	56.9229	92.401	56.819	90.5543
55	316.623	636.339	313.989	630.772	56.8393	94.38	56.6604	91.8796
56	313.886	634.302	314.306	626.74	56.9926	91.6432	56.8668	91.6131
57	316.285	640.792	312.25	627.367	57.0382	91.5493	56.6873	91.8275
58	320.895	641.558	314.677	623.54	56.9854	94.7163	56.8358	91.0681
59	322.426	635.103	319.288	621.794	56.8186	94.2499	56.6319	93.0172
60	318.339	634.281	315.621	627.073	56.9656	93.3501	56.6606	93.8003
61	320.518	629.81	313.448	617.635	57.1212	92.0155	56.743	92.3425
62	319.192	636.001	315.107	619.992	56.9131	92.1162	56.764	91.5158
63	321.941	637.206	315.075	674.175	56.9188	92.2805	56.7833	91.4504
64	320.1	633.353	317.133	666.431	57.0765	94.0517	56.7397	91.1646
65	316.021	636.021	313.574	625.213	56.9214	92.1783	56.7502	93.0524
66	317.736	637.913	319.961	623.635	56.8097	94.2574	56.8589	90.2496
67	318.897	635.28	315.016	632.502	56.949	92.2331	56.8504	91.1577
68	317.526	640.842	313.494	623.817	56.966	94.9444	56.6479	91.8856
69	317.967	633.72	315.02	623.978	57.0885	91.8362	56.6392	92.1431
70	318.335	641.812	316.134	626.033	56.9062	91.7786	56.8347	92.3844
71	316.929	637.779	313.061	626.316	56.8684	93.315	56.8383	92.5953
72	317.905	638.729	311.967	627.543	57.028	92.8946	56.6964	91.4264
73	317.011	633.496	315.293	620.725	57.0377	92.2739	56.6975	93.3162
74	317.497	631.598	315.638	621.432	56.9354	94.7177	56.7671	91.2687
75	315.613	638.801	315.123	628.937	56.9129	94.4715	56.8517	90.8369
76	315.228	633.443	317.299	619.111	57.0138	93.0534	56.679	92.1561
77	319.568	635.717	313.32	617.643	56.8708	94.0017	56.6531	91.4744
78	315.28	635.669	312.817	639.08	56.956	91.0853	56.7399	91.0825

Continued on next page

Table F.3 – continued from previous page

Events	SB CPU	LB CPU	SM CPU	LM CPU	SB GPU	LB GPU	SM GPU	LM GPU
79	312.189	638.198	319.035	628.254	56.8856	93.8419	56.762	91.1963
80	317.128	636.207	313.39	624.873	57.0389	93.5832	56.9052	92.1684
81	321.068	635.257	317.81	623.291	56.9676	94.5498	56.8341	91.5094
82	315.558	634.026	314.784	633.334	57.0167	92.5997	56.8334	93.9118
83	316.719	629.716	315.318	622.476	56.8687	91.7519	56.6493	92.8766
84	317.833	636.363	312.753	626.102	56.9779	92.9861	56.8124	94.1468
85	317.22	640.01	312.312	637.749	56.8061	93.9571	56.6881	91.3004
86	318.81	641.49	314.317	625.762	56.8697	92.3616	56.8316	91.4182
87	314.966	632.147	316.5	619.12	56.9178	94.1009	56.8215	91.6478
88	317.396	635.126	315.273	619.186	57.0847	93.3692	56.8636	91.0651
89	316.113	638.566	315.567	634.533	56.8984	93.8183	56.9485	91.4977
90	318.241	642.984	316.766	624.654	56.8726	91.71	56.7729	91.4903
91	317.309	638.839	313.682	622.968	56.8897	94.5237	56.7063	92.728
92	318.524	639.166	317.07	626.768	56.9673	94.1277	56.7357	92.2819
93	317.032	636.477	318.321	636.919	57.0142	92.6922	56.7876	91.2981
94	319.395	637.979	316.637	624.007	57.1011	94.361	56.8256	92.2703
95	315.955	634.894	313.991	626.747	56.9203	91.9113	56.8243	94.2366
96	316.867	637.25	316.223	633.234	56.9941	92.741	56.9016	92.1287
97	317.254	633.589	313.603	625.673	57.0684	93.843	56.6861	90.8741
98	318.888	637.862	313.037	627.081	57.0931	92.4767	56.7543	92.2072
99	317.633	640.664	310.596	622.362	56.8937	94.2231	56.6375	92.2826
100	321.062	632.418	314.432	632.743	56.9874	94.0779	56.8707	91.5797

Table F.3: Execution Time for Experiment 3.

Bibliography

BAS04. Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*, volume 1.0 of *PC System Architecture Series*. MindShare, Inc., Boston, 2004.

Ber10. Berkeley University. Bonic. http://bonic.berkeley.edu/, 2010.

BhS07. P. Bhattarakosol and V. Suttichaya. Multiple Equivalent Scale Scan: An Enhancing Technique for Malware Detection. In *IEEE 2nd Int. Conf. on Systems and Networks Communications*, page 71, 2007.

CDD09. Sylvain Collange, Yoginder S. Dandass, Marc Daumas, and David Defour. Using graphics processors for parallelizing hash-based data carving. In *42nd Hawaii International Conference on System Sciences*, pages 1 – 10. IEEE, 2009.

Cla09a. ClamAV. About clamav. http://www.clamav.net/, 2009.

Cla09b. ClamAV. Creating signatures for clamav. http://www.clamav.com/doc/latest/signatures.pdf, 2009.

Cla09c. ClamWin. Free antivirus for windows. http://www.clamwin.com/, 2009.

Coh86. F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1986.

Coh87. F. Cohen. Computer viruses - theory and experiements. *Computers & Security*, 6:22–35, 1987.

Dam10. Damn Small Linux. Minimum hardware requirements. http://www.damnsmalllinux.org/wiki/index.php/Minimum_Hardware_Requirements, Accessed July 2010, 2010.

Eil05. Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005.

For04. Richard Ford. The Wrong Stuff? [computer viruses]. *Security & Privacy, IEEE*, 2(3):86–89, May-June 2004.

Hey07. Karen Heyman. New Attack Tricks Antivirus Software. *Computer*, 40(5):18–20, May 2007.

HHL08. Nen-Fu Huang, Hsien-Wei Hung, Sheng-Hung Lai, Yen-Ming Chu, and Wen-Yen Tsai. A gpu-based multiple-pattern matching algorithm for network intrustion detection systems. In *22nd International Conference on Advanced Information Networking and Applications - Workshops*, pages 62 –67. IEEE, 2008.

HMH09. Guang Hu, Jianhua Ma, and Benxiong Huang. High throughput implementation of md5 algorithm on gpu. In *IEEE*. IEEE, 2009.

HoW04. Jin Young Hong and May D. Wang. High speed processing of biomedical images using programmable gpu. In *2004 International Conference on Image Processing (ICIP)*, pages 2455 – 2458. IEEE, 2004.

HTA08. Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. The Synchronization Power of Coalesced Memory Accesses. In *DISC 2008, LNCS 5218*, pages 320–334. Springer-Verlag Berlin Heidelberg, 2008.

Int00. Intel Corporation. Using streaming simd extensions (sse2) to perform big multiplications. Intel Product Information 2.0, September 2000.

Int05. Intel Corporation. *Intel Pentium 4 Processor on 90 nm Process*, datasheet edition, February 2005.

Int08. Intel Corporation. Processors: Define sse2 and sse3. http://www.intel.com/support/processors/sb/CS-030123.htm, Accessed July 2010, 2008.

Int10a. Intel Corporation. Intel hyper-threading technology (intel ht technology). http://www.intel.com/technology/platform-technology/hyper-threading/, 2010.

Int10b. Intel Corporation. Intel developer network for pci express architecture. www.intel.com/technology/pciexpress/index.htm, Accessed July 2010.

JaB06. Nigel Jacob and Carla Brodley. Offloading ids computation to the gpu. In *22nd Annual Computer Security Applications Conference (ACSAC '06)*, pages 14 –18. IEEE, 2006.

Jur08. Mario Juric. Notes: Cuda md5 hashing experiments. http://majuric.org/software/cudamd5/, 2008.

Kel09. Jim Kelly. Defeating AntiVirus Software. *Hakin9*, pages 28–34, January 2009.

KoM09. Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis. String matching on a multicore gpu using cuda. In *2009 13th Panhellenic Conference on Informatics*, pages 14 – 18. IEEE, 2009.

McA09. McAfee. McAfee Acquires Solidcore. Executive summary, McAfee, 2009.

McA10. McAfee Community. W32/wecorl.a 0-day? http://community.mcafee.com/thread/24056?start=0&tstart=0, 2010.

Mic01. Microsoft Corporation. Windows xp professional system requirements. https://www.microsoft.com/windowsxp/sysreqs/pro.mspx, Accessed July 2010, 2001.

Mic06. Microsoft. Common object file format (coff). *MSDN*, November 2006. Revision 4.1.

Mic07a. Microsoft Corporation. *Digital Signatures for Kernel Modules on Systems Running Windows Vista*, July 2007.

Mic07b. Microsoft Corporation. System requirements for windows vista. http://support.microsoft.com/kb/919183, Accessed July 2010, 2007.

Mic08. Microsoft. *Microsoft Portable Executable and Common Object File Format Specification*. Microsoft Corporation, 8.1 edition, February 2008.

Mic10. Microsoft Corporation. Windows 7 system requirements. http://windows.microsoft.com/systemrequirements, Accessed July 2010, 2010.

MIT07. The MITRE Corporation. Cve-2007-0870. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0870, 2007.

MIT09a. The MITRE Corporation. Cve-2009-0238. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0238, 2009.

MIT09b. The MITRE Corporation. Cve-2009-0556. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0556, 2009.

MiV08. John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *IEEE*, 2008.

NAs02. Network Associates. Advanced virus detection scan engine and dats. Executive white paper, McAfee Security, 2002. http://www.mcafee.com/us/local_content/white_papers/wp_scan_engine.pdf.

NVI09a. NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*, 1.4 edition, March 2009.

NVI09b. NVIDIA. *NVIDIA CUDA Programming Guide*, 2.3.1 edition, 2009.

NVI09c. NVIDIA. *NVIDIA CUDA Reference Manual*, 2.2 edition, April 2009.

NVI10. NVIDIA Corporation. Cuda zone. http://www.nvidia.com/object/cuda_home.html, 2010.

Pau08. Nathanael Paul. *Disk-Level Behavioral Malware Detection*. PhD thesis, University of Virginia, May 2008.

PCI10. PCI-SIG. Pci express 3.0 frequently asked question. http://www.pcisig.com/news_room/faqs/pcie3.0_faq/PCIe_3_0_External_FAQ_Nereus.pdf, Accessed July 2010.

Pie94. Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. *Microsoft Systems Journal*, March 1994.

Pie02. Matt Pietrek. An In-Depth Look into the Win32 Portble Executable File Format. *MSDN Magazine*, February 2002.

Pup09. Puppy Linux. Minimum hardware requirements. http://www.puppylinux.org/wikka/MinReq, Accessed July 2010, 2009.

Riv92. R.L. Rivest. The md5 message-digest algorithm. *RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

Sla10. SlavaSoft Inc. Slavasoft hashcalc: Hash, crc, and hmac calculator. http://www.slavasoft.com/hashcalc/index.htm, July 2010.

Sta10. Stanford University. Folding@home. http://folding.stanford.edu/English/Main/, 2010.

StL07. Mark Stamp and Richard M. Low. *Applied Cryptanalysis Breaking Ciphers in the Real World*. Wiley-Interscience, New Jersey, 2007.

Szo05. Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, New York, May 2005.

Ubu10. Ubuntu Community. Installation system requirements. https://help.ubuntu.com/community/Installation/SystemRequirements, Accessed July 2010, 2010.

Vak10. Nimesh Vakharia. Is it malware? - you make the call. *Symantec Security Blog*, January 2010. http://www.symantec.com/connect/blogs/it-malware-you-make-call, Accessed July 2010.

ViG07. Richard Viney and Richard Green. Gpu-accelerated computer vision on the linux platform. In *Proceedings of Image and Vision Computing New Zealand 2007*, pages 143–147, Hamilton, New Zealand, December 2007.

XFX09. XFX. *Specification Sheet*, geforce 9500 gt edition, 2009.

XiH05. Wang Xiao-yun and Yu Hong-bo. How to break md5 and other hash functions. In *EUROCRYPT*, pages 19–35, 2005.

YJD09. Wang Yu, Chen Jianhua, and He Debiao. A new collision attack on md5. *Networks Security, Wireless Communications and Trusted Computing, 2009. NSWCTC '09. International Conference on*, 2:767 –770, April 2009.

YWL07. Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye. IMDS: intelligent malware detection system. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1043–1047, New York, NY, USA, 2007. ACM.

Index

The index is conceptual and does not designate every occurrence of a keyword.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
16-09-2010	Master's Thesis	Sept 2008-Sept 2010

4. TITLE AND SUBTITLE

Accelerating Malware Detection via a Graphics Processing Unit

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Nicholas S. Kovach

5d. PROJECT NUMBER

N/A

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCO/ENG/10-12

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

688th Information Operations Wing
Attn: Mr. Robert J. Kaufman
102 Hall Boulevard, Suite 345
San Antonio, TX 78243
(210) 977-5377; robert.kaufman@lackland.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

688th Information Operations Wing

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Real-time malware analysis requires processing large amounts of data storage to look for suspicious files. This is a time consuming process that (requires a large amount of processing power) often affecting other applications running on a personal computer. This research investigates the viability of using Graphic Processing Units (GPUs), present in many personal computers, to distribute the workload normally precessed by the standard Central Processing Unit (CPU). Three experiments are conducted using an industry standard GPU, the NVIDIA GeForce 9500 GT card. Experimental results show that a GPU can calculate a MD5 signature hash and scan a database of malicious signatures 82% faster then a CPU for files between 0 - 96 kB. If the file size is increased to 97 - 192 kB the GPU is 85% faster than the CPU. This demonstrates that the GPU can provide a greater performance increase over a CPU. These results could help achieve faster anti-malware products, faster network intrusion detection system response times, and faster firewall applications.

15. SUBJECT TERMS

malware detection, graphics processing unit, GPU, NVIDIA CUDA, parallel static detections

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON / 19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	97	Dr. Barry E. Mullins / (937) 255-3636, ext 7979; barry.mullins@afit.edu

Reset

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18