Detecting Very Large Sets Of Referenced Files At 40/100 Gbe, Especially Mp4 Files

By

Adrien Larbanet, Jonas Lerebours and Jean Pierre David

From the proceedings of

The Digital Forensic Research Conference

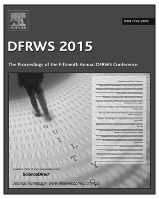
DFRWS 2015 USA

Philadelphia, PA (Aug 9th - 13th)

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2015 US

Detecting very large sets of referenced files at 40/100 GbE, especially MP4 files

A. Larbanet*, J. Lerebours*, J.P. David*

École Polytechnique de Montréal, 2900, boul. Édouard-Montpetit, Campus de l'Université de Montréal 2500, chemin de Polytechnique, Montréal H3T 1J4, Quebec, Canada

Keywords:
Video fingerprinting
Network monitoring
Deep packet inspection
Content-based detection
GPU computing

A B S T R A C T

Internet traffic monitoring is an increasingly challenging task because of the high bandwidths, especially at Internet Service Provider routers and/or Internet backbones. We propose a parallel implementation of the max-hashing algorithm that enables the detection of millions of referenced files by deep packet inspection over high bandwidth connections. We also propose a method to extract high-entropy signatures from MP4 files compatible with the max-hashing algorithm in order to have low false positive rates. The system first computes a set of fingerprints, which are small subsets of the referenced files a priori unique and easily identifiable. At detection time, the max-hashing algorithm eliminates the need to reconstruct the flows. A Graphics Processing Unit (GPU) card computes the fingerprints of all the IP packets in parallel and searches for hits in the onboard collection of fingerprints. Our application, dedicated to the detection of known MP4 video files, enables the detection of millions of fingerprints and demonstrates a sustained processing rate of 50 Gbps per card. Furthermore, a null false positive rate was observed for our 28.25 GB transfer test. The proposed implementation also features the detection of suspect flows based on IP addresses and ports in order to carry out deeper investigations off line.

Introduction

In 2017, an average of 283 Tb including 914,100 minutes of video will cross the Internet every second as estimated by Cisco (2012). While such performance is desirable and well exploited in most circumstances, it can also serve less glorious aims. Network supervision becomes more and more mandatory to prevent data leakage, to block malicious programs and to detect attacks. The forensics domain is also concerned with the transfer of files related to illegal activities such as child pornography, fraud, copyright infringement and others. All these fields of application involve *referenced data recognition*, a process that is usually composed of two steps:

1. Collection and reference of sensible data: these data are used to create a reference database.
2. “Live” analysis: a system collects new data and compares it to the reference database. A match will raise an alert or trigger an action.

Detecting referenced data over network connections is not obvious because the initial content is split into small packets (usually 500–1500 bytes for Ethernet) and embedded in various protocols. If the monitoring application has access to the initial content, a trivial approach consists of using standard hash values as fingerprints. As soon as a new object is present, its hash value is computed and compared against a set of reference hash values. Such a

* Corresponding authors.
E-mail addresses: adrien.larbanet@polymtl.ca (A. Larbanet), jonaslerebours@gmail.com (J. Lerebours), jpdavid@polymtl.ca (J.P. David).

http://dx.doi.org/10.1016/j.diin.2015.05.011

method can easily be implemented in a data center to monitor file uploads or in a proxy to monitor file downloads. Nevertheless, in general it is not possible to access the initial content or to reconstruct it for several reasons:

- The transfer must be detected before it is complete (e.g., data leakage prevention).
- The bandwidth and/or the required memory may be too high for real-time rebuilding (e.g. an Internet backbone).
- Not all packets may be available. Networks are usually interconnected via multiple links and the route taken by a given packet cannot be known in advance.
- The content may be embedded inside a container.
- The target applications and their protocols may be unknown.

Many research papers address the problem of detecting a limited number (thousands) of small signatures, typically regular expressions (Yu and Becchi (2013)). This paper addresses the problem of detecting very large numbers (tens of millions) of large objects, typically image or video files, in real time at very high bandwidth. In addition, the proposed GPU implementation enables the extraction of suspect TCP/UDP flows in the same traffic. The highly parallel architecture of GPUs, made of hundreds of processing cores, perfectly matches the hundreds of IP packets to be processed in parallel and independently. Furthermore, the large and fast memory available in GPU cards enables the detection of millions of known files and the extraction of as many flows within a single card.

The detection approach is founded on the max-hashing algorithm proposed by David (2013). A 128-bit sliding window generates 64-bit hash values, from which we only keep the one that maximizes a given criterion (actually we keep the highest hash value). Since the signatures are computed for very small data blocks, the max-hashing algorithm requires that the blocks contain high-entropy data (ideally unique).

In information theory, the entropy is the average amount of information, which is usually measured in bits. A low-entropy data source means that the data are redundant while a high-entropy source means that each symbol has the same probability to occur, which may be related to uniqueness as detailed in Section “False positive”. An important contribution of this paper is the computation of high-entropy signatures from MP4 files.

Our results show that a single GPU card is able to process up to 63 Gbps. Nevertheless, the PCIe 2.0 link used to feed the card is a severe bottleneck that limits the global processing rate to 50 Gbps. State-of-the-art PCIe 3.0 GPU cards or using two cards should enable the monitoring of 100 GbE connections quite straightforwardly.

The remainder of this paper is organized as follows: Section “Related work” describes previous work in the field of known data recognition. Section “The max-hashing algorithm” briefly presents said algorithm, the bedrock of the present work. Section “Referencing MP4 files with max-hashing” presents our proposed method for referencing MP4 files. Section “GPU computing” gives some background about the GPU architecture to allow a good understanding of the implementation proposed in Section “Max-hashing with GPU”. Section “Detecting TCP/UDP flows” presents our method for the extraction of suspect TCP/UDP flows. Section “Results” describes some test cases and reports the associated performance. Section “Discussion” addresses a few challenges related to the complete system configuration. Finally, Section “Conclusion” concludes this work and foresees some future contributions.

Related work

Known data recognition

Analyzing, classifying and identifying information for exponentially growing data sizes and transmission rates has led to new developments in the field of hashing and fingerprinting. These techniques all share a common objective: reducing large data sets to short representations while maintaining the ability to make fast and precise queries at a later time. MD5 (Message Digest #5) and SHA-1 (Secure Hash Algorithm) are two very common examples of hashing algorithms that are used in cryptography and forensics. Many other methods have been developed for cryptography (Bakhtiari et al., 1995), requiring a strong non-reversible property, but also in other fields with different goals such as simplified mathematics, easy hardware implementation and others (Knott, 1975). Fingerprinting in the forensics field is covered in detail by Roussev (2009, 2010, 2011).

Hash functions usually take a whole file as input and compute a fixed length fingerprint so that one can assume that two files with the same fingerprints are likely to be identical, with a very low probability of a false positive. This property is mainly used for file identification or authentication. One can therefore check for changes, updates or alterations of critical files by only inspecting their fingerprints. On the Internet, large files to download are usually accompanied by their hash value. When the download is complete, the user can compute the file's hash value and verify that it is the expected one. A different hash value means that the file is corrupted. The same principle is used when investigators need to examine data found on seized equipment. It is possible to quickly filter irrelevant files (Chawathe, 2009; National Institute of Standards and Technology, 2003) or to point out sensitive content by using a library of known hash values. Each file from the seized equipment is hashed and a lookup access is made to the hash value library. If the hash value exists, then the file is known and no further processing is required for that file. This is even more important for cases that involve huge amounts of data which can require several months to process and therefore delay further investigations. To help speed up such investigations, the National Software Reference Library, a database maintained by the National Institute of Standards and Technology (2003), regroups a collection of the fingerprints of popular files such as operating systems and widely used software. This is of great help to reduce the amount of relevant data to analyze.

Hashing whole documents does not give any measure of the difference between two files since even a single bit flip completely changes a fingerprint. Nicholas Harbour proposed the concept of *piecewise hashing*, implemented in his software dcfldd (Harbour, 2002) to address this problem. Data are split into fixed-size segments hashed independently. Therefore a single file is represented by several fingerprints. When parts of it are corrupted, only the fingerprints of the corrupted segments are modified. However, any data deletion or insertion in a file modifies all the segments from that point and thus all the corresponding fingerprints.

Andrew Tridgell faced this issue in proposing a software called SpamSum in 2002 (Tridgell, 2002). The purpose was to detect spam mails, which usually contain similar but not identical text, in order to circumvent classic hashing detection. His method shares some similarities with Rabin's fingerprints (Rabin, 1981), developed in the context of string matching. The original idea has more recently been adapted to the forensics field (Kornblum, 2006) as "*Context Triggered Piecewise Hashing*" (CTPH). A document is split into segments whose boundaries are determined by local properties. A fingerprint is then computed for each segment. Roussev (2009) details this method, which he names "data fingerprinting". Computing several fingerprints for each document increases the probability of detecting unchanged segments. More importantly, since the boundaries only depend on local windows of data, the corresponding segments are still detected when they are relocated, in the same document or not. More generally, these techniques address the measure of containment and similarity, as defined by Broder (1997).

Notable applications and specialized hardware benefits

Hashing has been widely used in network forensics. The main challenges are usually either to prevent network incidents or to retrace what happened after an incident. Typical incidents are intrusions and unauthorized accesses (Zheng, 2010), malicious infections and the transmission of sensible content (Yoshihama et al., 2010). Since it is usually not possible to log all the traffic over long periods, fingerprinting can reduce the storage size while preserving the ability to retrace incidents. In their survey, Broder and Mitzenmacher (2004) cover a few application fields that use Bloom filters to summarize and reduce content size and eventually speed up communications in distributed systems. This includes packet tracing, payload identification and traffic optimization (Callado et al., 2009).

These approaches are off line and post-attack methods, but fingerprints can also be used for "live" analysis: malicious software protection (Fechner, 2010) and live network securing systems (Zheng, 2010), for instance. However, because of the increasing network bandwidths, monitoring devices need to handle a growing amount of data. General Purpose Processors (GPPs) are not powerful enough to handle such flows, making specialized hardware mandatory. FPGAs (Lin et al., 2009), GPUs (Tumeo et al., 2011; Jamshed et al., 2012) and more recently many-core processors (Jiang et al., 2013) are potential targets for such processing because of their parallel architecture. In addition to their ease of programming, GPUs are known to offer significant performance gains (Vasiliadis and Ioannidis, 2010) against a GPP implementation.

The max-hashing algorithm

The max-hashing algorithm proposed by David (2013) is designed for quickly detecting the presence of some known content when the user can only access its fragments, which is precisely the context of file transfers over the Internet. It is particularly powerful when the reference database is large since the computation time does not depend on its size.

Basically, the max-hashing algorithm computes the hash value of every fixed-size –small– window in the data but only keeps the one with the *maximum value*. This maximum value has the useful property of representing any fragment of the original data containing the related window. In other words, we can spot some known content inside an Ethernet stream as soon as the window that generates the maximum value passes through the packet analyzer with a single database access per packet.

Implementation steps

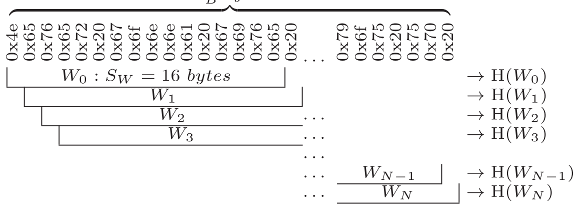
Either while referencing data or analyzing the live flow, the system processes the data in three steps: extract data blocks, compute fingerprints for these data blocks and finally access the fingerprint database to store the fingerprints or search for a match.

Split data into blocks

At referencing time, a set of high-entropy sub-blocks is extracted from the content to monitor. Ideally, each sub-block should be unique to the content so that a local match also means a global match. Using multiple blocks actually decreases the false positive probability. During live detection, the payload is extracted from the IP packets by removing the low level protocols (typically UDP and TCP).

Compute block fingerprints

A window size S_W and a hash function H must initially be chosen as constant parameters. Computing the fingerprint requires hashing all the overlapping windows with H. For a S_B-byte data block, every S_W-byte window is hashed, resulting in $S_B - S_W + 1$ fingerprints from which the algorithm will only keep the maximum one. Fig. 1 shows an example of every computed fingerprint from an input data flow. In the proposed implementation, the algorithm computes 64-bit fingerprints from 128-bit windows. The



$$Final fingerprint : F = \max_{0 \le n \le N}(H(W_n)) \quad with \quad N = S_B - S_W$$

Fig. 1. Fingerprints of each window in a data set.

implemented hash function is fully described in Section "Fingerprint computation".

Storing/searching fingerprints

For large reference data sets, DDR3 or GDDR5 memories are best exploited for burst accesses, which enable very efficient hash table implementations. We use a 2^p-row $\times$ c-column array of fingerprints. The hash function used to access the hash table is simply the p least significant bits of the fingerprint value. When the algorithm searches for a match, it keeps the p least significant bits and makes a burst read of the complete row. The fingerprint is further compared with each of the c cells in the row.

Possible issues

False negative

Considering a document with one fingerprint, splitting it into packets introduces a probability of cutting the window producing the maximum value fingerprint. In this case the document would not be detected. Let's consider a document of S_D bytes divided into S_P-byte packets. The fingerprints are computed from a S_W-byte window, and the probability of splitting right inside a given window is the number of possibly cut windows over the total number of windows, which can be calculated as follows:

$$p = \frac{\overbrace{(S_W - 1)}^{\text{Number of windows cut by a split}} \times \overbrace{\left(\left\lceil \frac{S_D}{S_P} \right\rceil - 1\right)}^{\text{Number of splits}}}{\underbrace{S_D - S_W + 1}_{\text{Total number of windows}}}$$

For instance, for a document of 15,000 B, divided into ten IP packets of 1500 B and using a 16-B window for fingerprint computation, we get about a 1% chance of not detecting the document at all.

This probability can be reduced by considering more than one fingerprint per document. The probability of not detecting n windows in different IP packets is $p_n = p^n$. In the previous example, the false negative probability with four fingerprints is less than one in a hundred million. However, this method requires more memory to store the fingerprints. For a given memory size, there is a compromise between the number of documents to detect and the probability of a false negative that can be tolerated.

False positive

The max-hashing algorithm relies on the fact that small windows are highly representative of the document. This is true for high-entropy content only because all the symbols (i.e., the complete small windows in our context) have the same probability of occurring and the probability of receiving a given symbol is 2^{-H}, where H is the entropy (in bits) of the small windows. Since we compute N-bit hash values, we need H to be greater or equal to N to ensure a maximum entropy for the signatures. For N large enough, the entropy becomes related to uniqueness. For instance, with $H = 64$, the probability of randomly receiving a given symbol is $2^{-64} = 5.4 \times 10^{-20}$.

The referencing step thus requires great care to select high-entropy regions in a given document, which is not always possible in the general case. However, large documents usually contain regions with compressed information, which intrinsically have high-entropy. These regions are good candidates for extracting the maximum value fingerprints. We present our method for the selection of high-entropy fingerprints from MP4 video files in section "Referencing original fingerprints from video files".

Optimizations

The best way to reduce the false detection rates (positive and negative) is to increase the number of fingerprints at reference time. This can be achieved in two ways: increasing the number of blocks in the content and/or using multiple hash functions. Since we do not have any control over the number of high-entropy blocks in the given content, it is relevant to use multiple hash functions. Nevertheless, we also desire to minimize the computation effort to maximize the processing rate.

In the proposed implementation of the max-hashing algorithm, we first use a common hash function. Then we derive four variants by just applying a XOR to the hash value with well-chosen constants C_i. In this way, the extra computation required to implement four hash functions instead of one is negligible. However, at detection time, four accesses to the database of reference fingerprints are now required instead of a single one.

Referencing MP4 files with max-hashing

The file referencing process can be improved by closely studying file formats. As described by Garfinkel et al. (2010), a dedicated approach must be adopted for each file format to be able to compute *unique* fingerprints. A quick overview of the standard video file format specification (ISO/IEC 14496-14 for MP4; ISO/IEC 14496-10 for H.264) and our own results presented in Section "Max-hashing the whole file" support the need for a specific fingerprinting method for MP4 video files using the H.264 encoder.

We propose to first identify high-entropy blocks in reference files based on their format. We focus on the *Macro-block* level of H.264 since data is compressed at such a level using *CABAC* or *CAVLC* algorithms. Furthermore, *Macro-blocks* describe the appearance of the video and are consequently unique to each file.

We modified the free and open-source software *Ffmpeg* and the included *libavcodec* library to extract the location of the high-entropy blocks in such video files. The modified application enables reading a video file and retrieving the position of any processed byte during the decoding stages of an MP4 (H.264) video file. At the end of the reading step, the application returns all the indexes and sizes of the high-entropy blocks.

Nevertheless, we cannot directly extract high-entropy blocks and apply the max-hashing algorithm to the blocks because of potential side effects. Actually, if a low-entropy block nearby a high-entropy block generates a maximum hash value higher than the one generated by the high-entropy block, there is a risk of a false negative if both blocks are present in the same IP packet. So, we apply the max-hashing algorithm to the whole file and we only keep the local maxima belonging to high-entropy blocks. This is an important contribution of the proposed method which leads to the impressive results presented in Section "Focusing on high-entropy blocks".

GPU computing

General-Purpose computation on Graphics Processing Units (GPGPU) consists of exploiting the high parallel computing power of GPUs to perform tasks in different contexts. Such applications have become so efficient that GPU manufacturers have modified the architecture of their processors to also best fit GPGPU. This section summarizes the main characteristics of a GPU as a basis for section "Max-hashing with GPU".

A GPU implements data level parallelism thanks to a SIMD (Single Instruction Multiple Data) architecture. The same instruction is performed on different data at the same time. Nowadays, these processors can launch several hundred processes in parallel. GPUs allow fast context switches to hide the latency of the external memory. Such an architecture becomes relevant as soon as an application applies the same sequence of operations on large data sets (as it is usually the case in graphics processing).

The vocabulary used in this section is the one proposed by NVIDIA in the description of the "CUDA" architecture (NVIDIA, 2013), the foundation of all their most recent processors.

Threads

Processing data on a GPU consists of running several occurrences of a function, called a "kernel", in parallel. A kernel usually takes an array of pointers as input and output arguments. Each instance of the launched function processes its own index in the array. In our work, we wrote the source code in CUDA C, which is developed by NVIDIA. Kernels can only access the onboard memory, except some special sharing manipulations available on the most recent processors. The interoperability between the GPU and the mainstream processor requires dedicated data transfers between their memories.

Software

Threads are organized in a three-level hierarchy, as illustrated in Fig. 2. The designer actually launches a "grid", which is a set of "blocks", themselves composed of

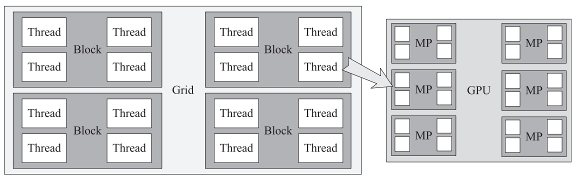


Fig. 2. Thread organization in NVIDIA CUDA architecture.

"threads" that all execute the same function. The processor is generally more efficient when there is no divergence between threads. Indeed, when processing a conditional branch instruction, the GPU splits the threads and runs them in two stages: first all the threads that branch run together, then the threads that do not branch. The advantages of the parallel architecture of GPUs are quickly lost in such cases because the numerous threads launched for parallel work are actually run serially.

Hardware

On chip, processing units are grouped in "multiprocessors". When a grid is launched, the scheduler maps the different blocks to the GPU's multiprocessors, so that each multiprocessor runs a block (Fig. 2).

Multiprocessors are independent from each other. Blocks running in different multiprocessors can come from different launches and belong to different grids. The NVIDIA K20c is composed of 13 multiprocessors, each running 32 threads at the same time. The group of currently running threads in a multiprocessor is called a "warp".

Recent NVIDIA GPUs can process several grids and several PCIe transfers at the same time. This ability can be used to create pipelined applications and reduce the transfer penalty.

Memory issues

GPUs usually embed their own GDDR memory, called the "global memory", which is a very high bandwidth dedicated memory coupled to L1 and L2 caches. This memory has a very high latency and is only efficient when the computing time is larger than this latency (200–400 cycles according to NVIDIA (2013)), which makes caching and data prefetching of the highest importance.

Moreover, cached global memory accesses are grouped over a whole warp and aligned on 128-byte segments (the size of the cache line). The GPU reads 128 bytes in the memory as soon as required data is not available in caches, and this is repeated for each thread in the warp. Thorough implementation is often required to keep data locality and anticipate further data requests.

Another memory, the "shared memory", is attached to each multiprocessor and can be accessed by all the threads within the mapped block. This memory is the same as the L1 cache and is very fast to access.

Small and temporary data can be stored in local registers. Nevertheless, their number is limited and split over a warp or a block (depending on the processor architecture). When too many registers are required, the compiler decreases the number of concurrent threads. In the worst cases, the registers are backed up in the "local memory", which is a part of the global memory mapped to each thread. Programmers have to limit their use of registers to run a maximum of threads at the same time in an efficient way.

GPU cards are connected to the motherboard through a bus, typically the PCIe on recent cards. This bus determines the bandwidth and the latency of all the transfers between the GPU and the host processor.

Max-hashing with GPU

An efficient GPU program involves a few adaptations of the original max-hashing algorithm to fit such specialized architecture. Section "Fingerprint computation" details how the algorithm has been adapted to a GPU and Section "Storing and searching in a database" addresses the database implementation. Results and performance for both parts are detailed in Section "Results".

Fingerprint computation

Computing all the hashes requires a significant amount of time and computational power. A rolling hash function is used to sequentially hash all the windows. It computes a fingerprint from the previous one in a small number of operations and keeps the maximum value "on the fly". In this way, the rolling hash mechanism just computes the contribution of the incoming byte and removes the contribution of the outgoing byte. More details about this mechanism are given in the work of David (2013). In the present implementation, the byte contributions are added and removed by the mean of a XOR function.

We compute 64-bit fingerprints from 128-bit windows. As current GPUs do not contain 64-bit integer ALU, computing two 32-bit hashes, h_{hi} and h_{lo}, instead of a single 64-bit one appears to be more efficient for these processors. In the following, "fingerprint" refers to the concatenation of these two 32-bit hashes. The byte contributions are computed through two functions, f_{hi} and f_{lo}. They use two 32-bit parameters, p_{hi} and p_{lo}, so that:

$$\mathrm{f}_{\mathrm{hi}}(x) = x \times p_{\mathrm{hi}}\ ,\ \ \mathrm{f}_{\mathrm{lo}}(x) = x \times p_{\mathrm{lo}}$$

The double rolling hash is performed according to the recurring relations:

$$h_{\mathrm{hi}}^{n+1} = \left(h_{\mathrm{hi}}^{n} \lll 4\right) \oplus \mathrm{f}_{\mathrm{hi}}(a_n) \oplus \mathrm{f}_{\mathrm{hi}}(a_{n+16})$$

$$h_{\mathrm{lo}}^{n+1} = \left(h_{\mathrm{lo}}^{n} \lll 4\right) \oplus \mathrm{f}_{\mathrm{lo}}(a_n) \oplus \mathrm{f}_{\mathrm{lo}}(a_{n+16})$$

These operations only require 32-bit rotates ($\lll$), multiplications and exclusive-or. The algorithm keeps the hash only when it is bigger than the current maximum (initially set to zero). To compute several hash functions H_i, we simply compute an exclusive-or between the hash values and predefined constants C_i ($1 \le i \le 4$). The results of this implementation are presented in Section "Fingerprint computation".

Storing and searching in a database

The database implementation is a straightforward adaptation of the hash table method described in Section "Storing/searching fingerprints" with a 2^p-row $\times$ c-column array of fingerprints. Two basic operations are required: storing and searching. They are both implemented as independent *kernels*.

Storing

Each *thread* is dedicated to a single fingerprint and sequentially reads the c_1 cells in the selected line and stops at the first zero value (empty cell). The value of the fingerprint is written there, as well as the reference location in the corresponding database. Read and replace accesses are executed as a single atomic operation to prevent conflicts between two different threads that try to access the same line at the same time. The performance is not measured here as we focus on the live analysis bandwidth.

Searching

The searching *kernel* reads the c_1 cells in parallel. We launch one *thread* per column and per input fingerprint. Searching for n_f fingerprints thus requires a total of $n_f \times c_1$ *threads*. This configuration results in a very simple *kernel* that only makes one comparison between a reference fingerprint and a value in the database. Memory accesses are automatically coalesced and aligned since we read contiguous 64-bit words in the database. The performance of the database search operations is presented in Section "Search in fingerprint database".

Detecting TCP/UDP flows

The system, as described above, can detect parts of known files through an Ethernet connection. Yet, it may be interesting to identify all the packets transferred between two given points that already triggered a hit (suspect flow). This is the purpose of the TCP/UDP flow detection part. A packet flow is determined by two IP addresses (source and destination), two port numbers and a transport layer protocol. In this paper, we focus on IPv4 addresses (32 bits) and TCP/UDP ports (16 bits). We adopted a similar approach as for the fingerprint database: the use of a hash table.

The packet header is first parsed: IP addresses and TCP/UDP ports are extracted and stored. A 16-bit index, for the hash table, is also computed from both IP addresses. We chose a commutative function ($\mathrm{f}(@_1,@_2) = \mathrm{f}(@_2,@_1)$) so that the direction of the packet has no impact.

The hash table is an array of 2^{16}-row $\times$ $c_2 = 8$ columns. Each cell contains two IP addresses and two boundaries for each port ($2 \times 32 + 4 \times 16 = 128$ *bits*). The whole table requires 8 MB for half a million cells. For each packet, the previously computed index indicates the row, and every non-empty cell in this row is tested. If the IP addresses are identical and the port numbers fit their boundaries, the CPU is alerted. The performance of the flow detection is presented in Section "TCP/UDP flow detection".

Results

The test bench is a Linux server composed of two Intel Xeon E5-2609 CPUs and an NVIDIA Tesla K20c GPU card connected to a PCI Express v2.0 16x interface. The GPU includes 13 multiprocessors with 32-thread warps and embeds 5 GB of GDDR5 memory. Fig. 3 presents an overview of the proposed detection system. The data are first copied into the main memory by the operating system. Then, they are transferred to the GPGPU memory where the computation can start, as already detailed.

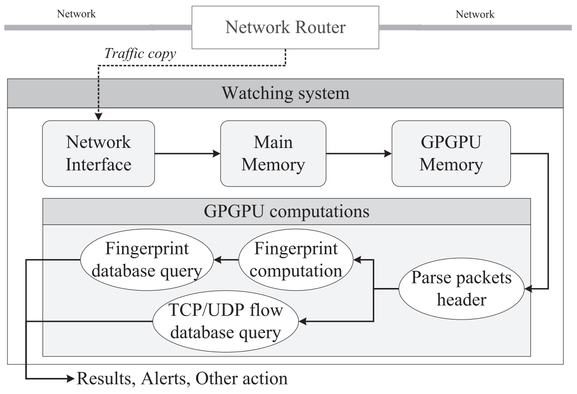


Fig. 3. Detection system Overview.

Test scenarios and results are summarized in Tables 1 and 2. We first report the performance of the proposed referencing algorithm. Then the bandwidth of memory transfers between the host and the GPU are measured because they will set the upper limit for the detection system. Finally, the performance of each kernel is measured independently to later be able to select and build an optimal configuration for the complete system.

Referencing original fingerprints from video files

We apply the max-hashing algorithm on 8073 MP4 video files (56.5 GB, 182.4 h) from the CCV database (Jiang et al., 2011). To measure the impact of our high-entropy block extractor on the detection results, we consider two distinct test cases: the basic case where the max-hashing algorithm is applied on whole files and the case where we focus on high-entropy regions as presented in Section "Referencing MP4 files with max-hashing". In both cases, the files are split into 128 blocks and 4 fingerprints are computed for each block. The results are shown in Table 1.

Max-hashing the whole file

Among the $8073 \times 128 \times 4 = 41{,}33{,}376$ computed fingerprints, we noticed 371 fingerprints that are common to *at least* two files (243 files generated at least one redundant fingerprint). Most of them were computed on files that share the same audio track (music sample) and involve only 7 files. But there are also 14 highly redundant fingerprints that are common to up to 143 visually unique files. We identified that these highly redundant fingerprints were computed on MP4 media information headers. Results show that a basic approach for referencing files generates redundant fingerprints and will consequently increase the probability of wrongfully detecting a file (false positive).

Table 1
Reference fingerprints computation.

	Whole file	High-entr. blocks
Total finger.	4,133,146	4,119,933
Redundant finger.	371	0
Files with redun.	243	0

Table 2
Test scenarios.

Tests and results	Configurations
Memory transfers Fig. 4	Variable buffer size DMA to pinned memory Host to device copy
Compute fingerprints 119.9 Gbps	768 MB of random input data 1536 B blocks 4 fingerprints per block No data transfer
Search in fingerprints database Fig. 5	2.09×10^6 random fingerprints $2^{p}{}_1$-row $\times$ c_1-columns database $p_1 = 21$ No data transfer
Parse packet headers Fig. 6	Variable number of packets Packets stored in 1536 B blocks No data transfer
Search in TCP/UDP flows database Fig. 7	524,288 random flows $2^{p}{}_2$-row $\times$ c_2-columns Database $p_2 = 16$ No data transfer
Chained kernels Fig. 8	Variable buffer size 1536 B blocks 4 fingerprints per block $c_1 = 16$ and $c_2 = 8$ No data transfer

Focusing on high-entropy blocks

This new approach does not generate any new redundant fingerprints and, as it filters both audio information and MP4 headers, the redundancies computed in the previous section are avoided. The computed fingerprints are all unique to each file in our database. The probability of a mismatch in the database is therefore nullified.

Memory transfers

Before being accessed and processed by the GPU, data must be copied into the dedicated memory. The theoretical data bandwidth of our PCI-Express 2.0 bus is 64 Gbps (16 lanes). We use Direct Memory Access (DMA) to copy a block of pinned memory to the GPU memory as this is the fastest way to transfer data because they cannot be swapped. Results show an actual sustained transfer rate of up to 50 Gbps, dependent on the transmitted block size, as presented in Fig. 4. The bandwidth is close to saturation for buffers as small as 2 MB. The application should therefore use buffer sizes greater than or equal to 2 MB.

Fingerprint computation

We measured the performance of this kernel by hashing 768 MB of random data. The kernel has to produce four fingerprints per 1536 B blocks (524,288 blocks). This test shows a processing rate of 119.9 Gbps, allowing the kernel to produce 40 million fingerprints per second.

Search in fingerprint database

The implemented fingerprint search function was launched on 2 million random fingerprints to search in a $2^{p}{}_1$-row $\times$ c_1-columns database (p_1 is set to 21, c_1 is variable). Opposite configurations were tested: first when no

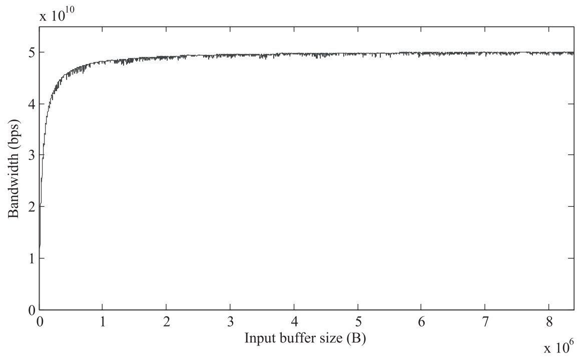


Fig. 4. Transfer performances.

hit occurs, then when hits occur for every searched hash. No relevant differences were observable. We can see in Fig. 5 that the hash searching rate varies linearly with $1/c_1$, an expected tendency as the total workload is linear with c_1.

TCP/UDP flow detection

These results refer to the bottom of Fig. 3, as described in Section "Detecting TCP/UDP flows".

Parsing headers

This kernel extracts, for each packet, the payload position, IP addresses and TCP/UDP ports and computes the index in the hash table from them. Results are stored in the GPU memory and are ready to be handled by further *kernels*. Fig. 6 shows that 400 Mpps (million-packets-per-second) can be treated. For 64 B packets, which is the minimal size for a standard Ethernet packet, it represents a total bandwidth of 204.8 Gbps.

Search in TCP/UDP flow database

The hash table is an array of 2^{p_2} lines and c_2 columns with $p_2 = 16$ and c_2 is variable. Each cell contains two IP addresses and two boundaries for each port ($2 \times 32 + 4 \times 16 = 128$ *bits*). For each packet, the previously computed index indicates the line, and every non-empty cell in this line is tested. If the IP addresses are identical and the port numbers fit their boundaries, the packet and the positions of the complying cells are reported back to the CPU. As with the fingerprint search *kernel* (Section "Search in fingerprint database"), the search rate varies linearly with $1/c_2$ and reaches 518 Mpps when $c_2 = 1$, as illustrated in Fig. 7. The performance spikes are the effect of optimizations provided by NVIDIA's architecture.

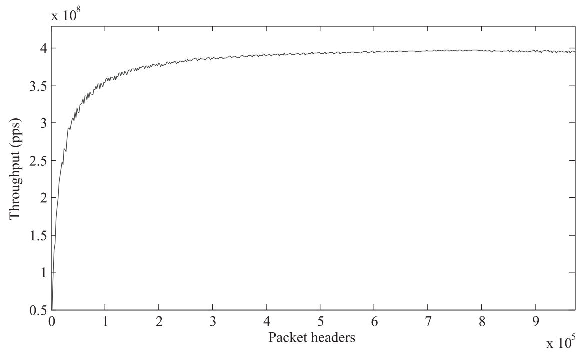


Fig. 6. Parsing headers.

Single-GPU complete system

A single-GPU configuration, where all the *kernels* run on the same GPU, is tested here. The performance of the resulting chain (illustrated in Fig. 3) is measured.

Chained kernels

In this configuration, *kernels* are launched sequentially except for the TCP/UDP flow search. This one can be launched after parsing the headers (in parallel with the fingerprint computation). The data are already stored in the GPU memory before the launch. They are real packets captured while transferring referenced files via FTP. Results are presented in Fig. 8: the solid blue line is the computing time of the chained *kernels* and the dashed red line is the memory transfer time at 50 Gbps (maximum PCIe 2.0 measured bandwidth). We achieved a maximum throughput of almost 63 Gbps.

Pipeline setup

An interesting feature of Tesla GPUs is that independent transfers and computations can overlap, so that the

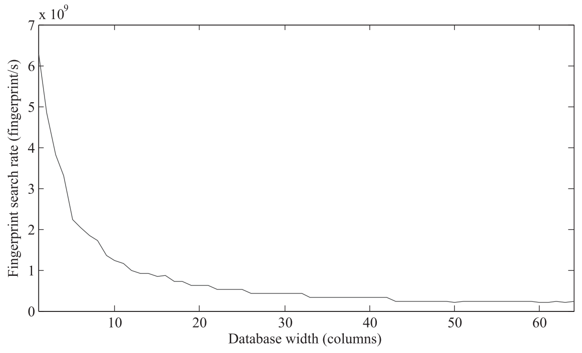


Fig. 5. Search in fingerprints database.

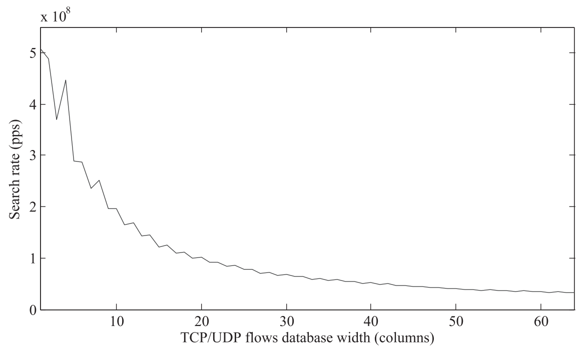


Fig. 7. Search in TCP/UDP flows database.

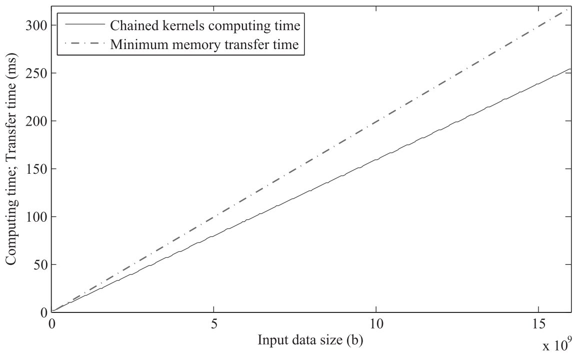


Fig. 8. Chained kernels computing time & minimum transfer time.

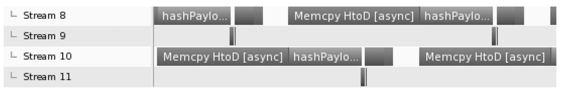


Fig. 9. Pipeline transfer and computing.

application can be pipelined: $packet_{n+1}$ is being transferred, $packet_n$ is being processed and the results from $packet_{n-1}$ are copied back at the same time. The bandwidth of the complete pipelined system is therefore limited by its slowest part, either transfer or computation. A physical limit already exists on the PCIe bus (measured in Section “Memory transfers”). These transfers represent the “slowest part” because the computations run faster than this limit as illustrated in Fig. 8. We therefore pipelined the whole application allowing it to fully exploit the use of the PCIe bus ($\approx$50 Gbps). Fig. 9 is a screenshot of the NVIDIA profiler illustrating the implemented pipeline: *streams* 8 and 10 are used to sequentially transfer data to the GPU, parse headers, hash payloads, search for known fingerprints and copy back the results to the CPU while *streams* 9 and 11 can search for TCP/UDP flows in parallel as soon as possible. We can visually identify the data transfer (*Memcpy HtoD*) to the GPU as the bottleneck of the system.

False positive rate measurement

At the beginning of this section, we measured the redundancy in the fingerprints when they are generated from the whole content or when we focus on the high-entropy blocks as proposed in Section “Referencing MP4 files with max-hashing”. To further measure the impact of the high-entropy block extraction, we have split our reference database into two sets. The first set is used to compute the fingerprints and the second set is used at detection time. A total of 28.25 GB of data are transferred via FTP and four fingerprints are computed for each IP packet, leading to the computation of about 500 M fingerprints. Ideally, we should not observe any match since no reference file is sent by FTP.

Basic approach, with redundant fingerprints
A total of 18K fingerprints were (wrongly) detected. The equivalent false positive rate is 3.8/100,000. The real rate is actually a little bit higher because when several *hits* are associated with the same redundant fingerprint, a single *hit* is reported.

With unique high-entropy fingerprints
Not a single *hit* has been observed in this configuration.

Discussion

False positive/negative rate reduction

Focusing on high-entropy regions greatly reduces the false positive rate. If required, taking the neighborhood of a fingerprint into account may further decrease the false positive rate as detailed by David (2013). Concerning the false negative rate induced by the packet segmentation (Section “False negative”), it can be lowered in the referencing process by increasing the number of fingerprints per file.

Capturing high bandwidth connection

A limiting factor we observed is the need of multiple data copies in the main memory. Actually, network card processors and GPUs both use DMA to copy their data to and from the main memory but they cannot share the same memory space. Multiple intermediate copies of the packets are performed from the network card to the GPU. The impact of these multiple copies is not noticeable for a bandwidth of 10 Gbps or less, but its effect appears at higher bandwidths. *PF_RING* and *PSIO*, for instance, improve the high-speed capture and enable a full 40 Gbps transfer as mentioned by Jamshed et al. (2012).

Comparison to other methods

Most content-based detection systems rely on feature extraction that requires access to the whole file. Such methods are more robust to small changes than the proposed one, which works at the binary level, but they cannot be applied if the monitoring application cannot access or reconstruct the whole file. Our approach has the advantage of only requiring access to individual IP packets. Its simplicity enables real-time processing at bandwidths as high as the ones used between Internet Service Providers at the cost of a few GPU cards.

Conclusion

We have proposed an implementation of the max-hashing algorithm on Graphics Processing Units (GPUs) and a method to compute high-entropy fingerprints from MP4 video files. Known files are first referenced in a fingerprint database, which is stored in the GPU card's embedded memory. The proposed method to extract high-entropy fingerprints led to very good results. In our experiment, 28.25 GB of data have been transferred by FTP and no false positives have been observed. Concerning the false negative rate, it can be lowered as much as desired by increasing the number of fingerprints computed per file.

Our results show a 63 Gbps processing rate for a single GPU card with the ability to store more than 32 million fingerprints and to track half a million TCP/UDP flows. Nevertheless, the PCI Express 2.0 limits the transfer rate to about 50 Gbps. The proposed implementation can thus efficiently monitor a 40 Gbps Ethernet stream with a single GPU card. Because of the parallelism of the application, a single PCIe 3.0 GPU card or two PCIe 2.0 GPU cards should straightforwardly enable the detection at 100 Gbps.

The proposed method has been fully implemented and tested for MP4 files. Nevertheless, it can be easily extended to other formats provided that it is possible to identify high-entropy blocks. This is typically the case for audio, image and video files since most of them contain compressed data blocks. Thus, the proposed system may be used by forensics investigators at very large scales (up to Internet backbones) to reliably detect the transfer of very large sets of referenced audio, image and video files.

Acknowledgments

The authors are grateful to NetClean and the Government of Quebec who have funded this work under the grant number PSVT3-25624.

References

Bakhtiari S, Safavi-naini R, Pieprzyk J. Cryptographic hash functions: a survey. Tech. rep. 1995.

Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. Internet Math 2004;1(4):485–509.

Broder AZ. On the resemblance and containment of documents. In: Compression and complexity of sequences 1997. Proceedings; 1997. p. 21–9.

Callado A, Kamienski C, Szabo G, Gero B, Kelner J, Fernandes S, et al. A survey on internet traffic identification. Commun Surv Tutor IEEE 2009;11(3):37–52.

Chawathe SS. Effective whitelisting for filesystem forensics. In: Intelligence and security informatics, 2009. ISI '09. IEEE international conference on; 2009. p. 131–6.

Cisco. Vni forecast highlights. 2012 [accessed 20.03.14]. URL, http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html.

David JP. Max-hashing fragments for large data sets detection. In: Reconfigurable computing and FPGAs (ReConFig), 2013 international conference on; 2013. p. 1–6.

Fechner B. Gpu-based parallel signature scanning and hash generation. In: 23rd international conference on architecture of computing systems (ARCS), 2010; 2010. p. 1–6.

Garfinkel S, Nelson A, White D, Roussev V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. Digit Investig 2010;7:S13–23.

Harbour N. dcfldd. 2002 [accessed 21.03.14]. URL, http://dcfldd.sourceforge.net/.

Jamshed MA, Lee J, Moon S, Yun I, Kim D, Lee S, et al. Kargus: a highly-scalable software-based intrusion detection system. In: Proceedings of the 2012 ACM conference on computer and communications security. CCS '12. New York, NY, USA: ACM; 2012. p. 317–28. URL, http://doi.acm.org/10.1145/2382196.2382232.

Jiang H, Zhang G, Xie G, Salamatian K, Mathy L. Scalable high-performance parallel design for network intrusion detection systems on many-core processors. In: Proceedings of the ninth ACM/IEEE symposium on architectures for networking and communications systems. ANCS '13. Piscataway, NJ, USA: IEEE Press; 2013. p. 137–46. URL, http://dl.acm.org/citation.cfm?id=2537857.2537883.

Jiang Y-G, Ye G, Chang S-F, Ellis D, Loui AC. Consumer video understanding: a benchmark database and an evaluation of human and machine performance. In: Proceedings of ACM international conference on multimedia retrieval (ICMR), oral session; 2011.

Knott GD. Hashing functions. Comput J 1975;18(3).

Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digit Investig 2006;3:91–7.

Lin Y-D, Lin P-C, Lai Y-C, Liu T-Y. Hardware-software codesign for high-speed signature-based virus scanning. IEEE Micro 2009;29(5):56–65.

National Institute of Standards and Technology. National software reference library. Aug. 2003 [accessed 20.03.14]. URL, http://www.nsrl.nist.gov.

NVIDIA. Cuda c programming guide version 5.5. July 2013 [accessed 20.03.14]. URL, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

Rabin MO. Fingerprinting by random polynomials. Tech. rep. Center for Research in Computing Technology, Harvard University; 1981.

Roussev V. Hashing and data fingerprinting in digital forensics. Secur Priv IEEE 2009;7(2):49–55.

Roussev V. Data fingerprinting with similarity digests, vol. 337/2010. Springer; 2010. p. 207–26.

Roussev V. An evaluation of forensic similarity hashes. Digit Investig 2011;8(SUPPL.):S34–41.

Tridgell A. spamsum. 2002 [accessed 21.03.14]. URL, https://www.samba.org/ftp/unpacked/junkcode/spamsum/.

Tumeo A, Secchi S, Villa O. Experiences with string matching on the fermi architecture. 2011.

Vasiliadis G, Ioannidis S. GrAVity: a massively parallel antivirus engine - recent advances in intrusion detection. Vol. 6307 of lecture notes in computer science. Berlin/Heidelberg: Springer; 2010. p. 79–96.

Yoshihama S, Mishina T, Matsumoto T. Web-based data leakage prevention. In: IWSEC; 2010.

Yu X, Becchi M. Gpu acceleration of regular expression matching for large datasets: exploring the implementation space. In: Proceedings of the ACM international conference on computing frontiers. CF '13. New York, NY, USA: ACM; 2013. 18:1–18:10. URL, http://doi.acm.org/10.1145/2482767.2482791.

Zheng Q. An improved multiple patterns matching algorithm for intrusion detection. In: IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS), 2010, vol. 2; 2010. p. 124–7.