Reducing time cost in hashing operations

Frank Breitinger*, Kaloyan Petrov†,
*da/sec – Biometrics and Internet Security Research Group
Hochschule Darmstadt, Darmstadt, Germany
Email: frank.breitinger@cased.de
†Institute of Information and Communication Technologies
Bulgarian Academy of Sciences, Sofia, Bulgaria
Email: kpp@acad.bg

***Abstract*—During a forensic investigation, an investigator might be required to analyze the content of a personal computer. Due to huge amounts of data, it becomes necessary to recognize suspect files and automatically filter out non-relevant files. To achieve this goal, an investigator can resort to hashing algorithms in order to classify files into known-to-be-good, known-to-be-bad and unknown files. The working steps are quite simple: hash the file, compare the resulting hashes against a database and put it in one of the categories. Typically personal computers nowadays store several hundred thousand files on their hard disk and thus this operation becomes time consuming.**

The paper at hand demonstrates a framework that speeds up this proceeding as it uses multiple threads for different tasks. Besides the typical multi-threading where the hashing algorithm is performed by multiple threads , we use a dedicated thread for reading files from the device, a prefetcher. Compared to single threading we improved the run time efficiency by nearly 40%.

***Keywords*-Digital forensics; hashing; cryptographic hash functions; performance; run time efficiency; file handling; prefetching.**

I. INTRODUCTION

Over the recent years the distribution and usage of electronic devices increased. Traditional books, photos, letters and LPs became ebooks, digital photos, email and mp3. This transformation also influences the capacity of todays storage media [22] that changed from a few megabytes to terabytes. Thus, the amount of data gathered within a computer forensic acquisition process is growing rapidly. The crucial task to solve this data overcharge is to distinguish relevant from non-relevant information which often resembles to look for a needle in a haystack.

To cope with this huge amount of data, investigators often use an automated preprocessing that groups files into three categories: known-to-be-good, known-to-be-bad and unknown files. For instance, system files of the operating system or binaries of a common application like a browser are said to be known-to-be-good and need not be inspected within an investigation. The working steps are quite simple: hash the file, compare the resulting fingerprint against a set of fingerprints and put it in one of the categories. The most common set/database of non-relevant files is the *National Software Reference Library* (NSRL, [13]) maintained by the US National Institute of Standards and Technology (NIST).

Due to the large amount of data the run time efficiency is important - *time is money*. Thus, one property of hashing algorithms is 'ease of computation' which is fulfilled by all traditional hashing algorithms, like SHA-1.

Additionally the performance of modern hardware increases rapidly as said by Moore's Law [12]. Multiple cores and powerful GPUs allow parallelizing the algorithms [1] and tasks. However, "the biggest bottleneck often involves loading data from files" [23]. One possibility to speed up the reading/writing is to use RAID systems[1] that combines multiple disk drive components into a logical unit. Thus instead of reading/writing from a single disk, we may use two or more. Another possibility are solid-state drives (SSD) which have higher throughputs than conventional hard disks. For instance, [4] says that SSDs can have a write/read speed up to 500MB/s whereas HDDs only up to 120MB/s. However, as RAID and SSD are not very widespread yet an intelligent file handling is important.

The paper at hand therefore introduces a new framework for file handling, that uses *prefetching*. Instead of having several threads / cores hashing independently we use one thread for reading and all remaining threads for hashing. The framework could be easily used for all hashing algorithms or different approaches.

The rest of the paper is organized as follows: In the Sec. II we discuss the state of the art and discuss relevant literature. Next, in Sec. III we explain different hash functions and their usage in digital forensics. Sec. IV is the core of our work an present the framework itself followed by some experimental results in Sec. V. At the end there is the conclusion.

II. RELATED WORK

Hash functions are very popular in computer science and deployed in many working fields. Probably the best-known areas are cryptography and databases [21, Sec. 9.6]. However, hash functions are also applied within computer forensics to identify files [2, p.56++]. According to [11] hash functions have two basic properties *compression* and *ease of computation*. Thus, as expressed by the latter one, run time efficiency with respect to computational complexity is important.

Besides traditional hash functions like FNV [14] or cryptographic hash functions like SHA-1 [20] or MD5 [15] there is

[1]http://en.wikipedia.org/wiki/RAID; last accessed 14.10.2012

also similarity preserving hashing[2] as proposed by Kornblum (`ssdeep` in 2006, [10]), Roussev (`sdhash` in 2010, [16]) or Breitinger et al. (`mrsh-v2` in 2012, [8]). Furthermore [8] showed that SPH algorithms are normally slower than the aforementioned traditional hash functions. In order to optimize the run time efficiency most authors focus on the algorithms themselves. For instance, improvements for `ssdeep` with respect to computational efficiency were presented in [6], [9], [18], [19].

III. Foundations

A. Hash functions in digital forensics

Nowadays the most popular use case of cryptographic hash functions within computer forensics is detecting known inputs. In order to detect files based on their fingerprints, the computer forensic investigator must have a database at hand, which comprises at least a referrer to the input file and its hash value. If he finds this hash value on a storage medium within an investigation, he is convinced that the referred file is present on the medium. In computer forensics hash values are typically computed over the payload of a file, i.e., hash functions are applied on the file level. Hence, known files can be identified very efficiently.

As it is a challenging task to generate a capacious database, often global databases are used. However, if a file is known-to-be-good or known-to-be-bad depends on regional law. Therefore hash databases have to be adapted according to the national boundary conditions. The most famous database is the *National Software Reference Library* (NSRL, [13]) with its Reference Data Set (RDS)[3].

While investigating a storage medium, the forensic software hashes the input, performs look-ups to the RDS and filters out each non-relevant file. Thus this reduces the amount of data the investigator has to look at by hand. Typically the software uses one thread to read and hash the file.

B. Similarity preserving hashing

In Sec. II we briefly introduced similarity preserving hashing (SPH) and stated that these algorithms are often slower. In order to test our framework we included three similarity preserving hashing algorithms `ssdeep`, `sdhash` and `mrsh-v2` to demonstrate its effectiveness. In the following we summarize the overall concept of the algorithms.

1) `ssdeep`: or also known as context triggered piecewise hashing (abbreviated CTPH) divides an input into approximately 64 pieces and hash each piece separately. Instead of dividing the input into blocks of a fixed length, it is divided based on the current context of 7 bytes. The final hash value is then a concatenation of all piecewise hashes where the CTPH only uses the least significant 6 bits of each piecewise hash. This results in a Base64 sequence of approximately 64 characters. A detailed description is given in [10].

2) `sdhash`: identifies "statistically-improbable features" using an entropy calculation. These characteristic features, a sequence of length 64 bytes, are then hashed using the cryptographic hash function SHA-1 and inserted into a Bloom filter [5]. Hence, files are similar if they share identical features.

3) `mrsh-v2`: is mainly based on `ssdeep` but includes some improvements which makes it more robust and efficient. For instance, the authors removed the restriction of 64 pieces which is a security issue (see [3]). Furthermore instead of using a Base64 fingerprint, `mrsh-v2` creates a sequence of Bloom filters as proposed in [16]. A detailed description is given in [8].

C. Run time efficiency of hash functions

This section compares the run time efficiency of different existing hashing algorithms which are all included in our framework. All tests are based on a 500MiB file from `/dev/urandom` and the times were measured using the `time` command and the algorithm CPU-time (`time` denotes this by `user`-time)

The results are shown in Table I. Row 1 shows that the cryptographic hash functions like MD5 and SHA-1 outperform every similarity preserving hashing algorithms. In row 2 we used SHA-1 as benchmark.

D. OpenMP

"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer."[4].

IV. A parallelized framework for file hashing

In the following we present a framework that optimizes the file handling for hashing. It is written in C++ and uses OpenMP 3.1 for multi-threading as explained in Sec. III-D.

The framework is divided into two branches - simple multi-threading (SMT) and multi-threading with prefetching (MTP) whereby our results focus on the latter approach. SMT was added to show the benefits of prefetching compared to MTP and thus this is only a side product.

We decided to name it *parallel framework for hashing* and use the abbreviation `pfh`. The framework can be downloaded on our web-page[5].

Remark. Additionally the framework supports OpenMP 2.0, but with a decreased run time efficiency due to 'capture

[2]This term might be a little bit confusing and *similarity digest* is more appropriate as hashing normally indicates a fixed size output. But as most of the similarity preserving (SPH) algorithms do not output a fixed sized hash value, we use similarity digest, fuzzy hash function and similarity preserving hash function as synonyms.

[3]http://www.nsrl.nist.gov; last accessed 13.01.2013

[4]http://openmp.org/wp/about-openmp/; last accessed 13.01.2013

[5]https://www.dasec.h-da.de/staff/breitinger-frank/#downloads; last accessed 13.01.2013

	SHA-1	MD5	MRSH-v2	ssdeep 2.9	sdhash 2.0
runtime	2.33s	1.35s	5.23s	6.48s	22.82s
$\frac{\text{algorithm}}{\text{SHA-1}}$	1.00	0.58	2.24	2.78	9.78

TABLE I
RUN TIME EFFICIENCY COMPARISON OF SIMILARITY PRESERVING HASH FUNCTIONS AND SHA-1.

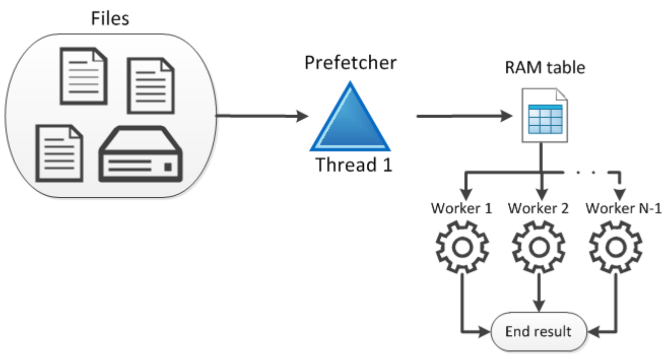

Fig. 1. Operations of the framework.

clause' which is only available since OpenMP 3.1. The capture clause allows to copy a global variable into a local one and increment the global variable, in an atomic operation. In OpenMP 2.0 capture clauses can be replaced with critical sections, but this reduces run time efficiency.

The rest of this section is structured as follows. Sec. IV-A presents the general idea of the algorithm followed by the command line in options and different modes in Sec. IV-B. In Sec. IV-C we describe the exact proceeding and working steps of our framework. Next are the implementation details in Sec. IV-D which discusses the main classes of the framework and give an example how to add new hashing algorithms. The last section presents some future work.

A. *Overview of multi-threading with prefetching*

In contrast to traditional approaches where hash functions request a file and process it, we implement a prefetching mechanism. A sketch of the overall approach is shown in Fig. 1.

The *prefetcher* handles the file reading and is responsible for the communication between hard disk an RAM. The general idea is that the critical resource - the hard disk - should 'work' all over the time. Thus, the prefetcher produces an ongoing file request.

All files are placed within the RAM wherefore we use a RAM table which limits the amount of storage. All remaining threads are *workers* and proceed the files from the RAM using the defined hashing algorithms. After hashing the file, there are several opportunities which is denoted by 'end result'.

Depending on the computational efficiency of the hashing algorithm there are two possibilities:

1) If the hashing algorithm is fast, the worker threads are faster than the prefetching process and thus the workers have to idle. However, the hard disk is at its limit and cannot process faster.
2) If the hashing algorithm is slow[6], the RAM table becomes full and cannot store any further files. Thus the prefetcher starts to idle. In this case the prefetcher thread could turn into a normal worker and help to proceed the files in RAM table[7].

One may say that distributed systems may further increase the performance. As will be demonstrated the limiting resource is the underlying device and not the computational power of the system. Thus there won't be any improvement using distributed systems for hashing inputs.

B. *Command line parameters and operation modes*

Before describing the details of our framework we briefly introduce the command line options that allow a rough configuration. Let N be the number of processor cores in the system and let the value of option `-p` be denoted by P where $P < N$.

- `c` - mode of framework operation [optional] (explained in the following paragraph).
- `d` - directory to be hashed or file with digests [is required].
- `r` - recursive mode for directory traversing [optional].
- `p` - number of prefetching threads [default is 1].
- `t` - number of all threads [default is N].
- `h` - hashing algorithm [default is MRSH-v2].
- `m` - size of used memory in megabytes [default is 20MB].

We have already motivated the default values for `-p` and `-t`. The motivation for a default memory size of 20MB is based on the average file size. [6] stated that the average file size of an operation system is approximately 255KB. This also somehow coincides with the t5-corpus where the average file size is at 418KB. Due to the largest file of $\approx$ 19MB we set the default to 20MB.

Currently there is one drawback: all files larger than the RAM table will be skipped. This issue will be fixed in an upcoming version.

In general the framework can operate in four different modes named in the following:

- `HASH` - All files are hashed using the specified algorithm and the results are printed on the standard output [default].
- `FULL` - The framework does an all-against-all comparison of all files in `directory`.
- `<DIGEST>` - All files within `directory` are hashed and compared against `DIGEST` which is a single fingerprint.

[6]Of course all hashing algorithms are supposed to be fast. However, some similarity preserving hashing algorithms like `sdhash` are multiple times slower than SHA-1 and thus we use the term slow.

[7]This functionality is future work. Currently the prefetcher never changes its role.

If value of parameter `-d` is a fingerprint file, the framework will compare `DIGEST` against all fingerprints with the file - skipping the hashing stage.

- `<FILENAME>` - `<FILENAME>` needs to be replaced by a path to a file containing a list of valid hash values. The framework hashes all files in `directory` and compares them against the list. If the signature is found within the list, it is a valid result[8]. This functionality is part of the framework, the implementation of the hash algorithm need not have an option for doing it.

Sample execution of the framework. The following command will execute the framework in the default mode, with a RAM table of size 256MB. The t5 directory will be traversed recursively and all hashes are sent to the standard output. Since we did not specify the `-t` and `-p` options, the program has $P = 1$ prefetching thread and $N - P$ hashing threads. Recall, N is the number of available processor cores in the system.

```
$ pfh -c hash -m 256 -d t5 -r
```

C. Proceeding

On starting there is an initializing part where the framework creates its 4 building blocks - options, hashing interface, ram table and mode of operation. Then the input parameters are parsed from the options class. In the following, the hashing interface is pointed to the chosen hashing algorithm and variables are set. RAM table is created at last, since it needs information from the hashing algorithm, to initialize its file filters.

The directory holding files is traversed and each file that passed the filter is added to files-to-be-hashed-list. Currently the filter system concerns about the file size and access rights. For instance, files larger than the RAM table cannot be handled. Furthermore some algorithms may need a minimum file size.

Next, we transfer the list structure to an array for easier thread processing. Knowing the amount of files that will be hashed, we initialize some of its internals to optimize its performance. Last part of the framework initialization sets the mode interface. Here we point comparison/result functions to the specified mode and set internal variables, if any.

The actual framework processing can be broken into three stages 1) reading/hashing files, 2) comparing hash values and 3) presenting results/scores whereby only the first and second stage are executed with multiple threads while the third stage is sequential. Threads are created before the first stage and finalized at end of second stage. This way no time is lost, for thread management (fork/join), during framework operation.

1) **Reading/hashing files.**
   - **SMT branch.** Each of the N threads put its file into the RAM table and hashes it. All threads continue until there are no more files in the queue.

     After being assigned to a role (reading or hashing), threads enter a 'work loop' for execution. Based on the return value, threads can change their role, e.g., if the RAM table is empty.
   - **MTP branch.** First, there is a thread assignment where every thread receives its role, i.e., we set P prefetchers. All $N - P$ threads are hashing threads. Currently within MTP all threads preserve their role over the whole runtime.
2) **Comparing hash values.** This stage is also executed in parallel using the OpenMP 'parallel for' clause, in which threads work on chunks of the global compare iterations. Scores from comparison are held in an array, because if threads print to screen, they have to synchronize and the speedup of parallelism is lost.
3) **Presenting results/scores.** At the end the file-path, hash value and score, if compare mode is used, are given to the standard output.

[8]This is equal to the `-m` option of `ssdeep`.

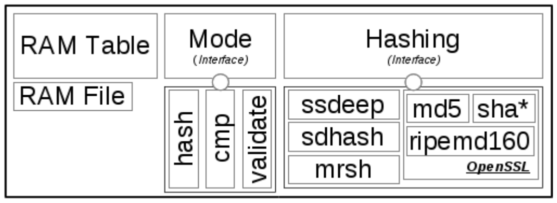

Fig. 2. Objects of the framework

D. Implementation details

Besides the two branches SMT/MTP and the operation modes, the framework mainly consists of two objects named RAM table and hashing interface as shown in Fig. 2 which will be explained in the following.

1) SMT and MTP: In the following we describe how to switch between the two branches of the framework - simple multi-threading (SMT) and multi-threading with prefetching (MTP). Although we could not find any case where SMT outperforms MTP, we describe how to use it for the sake of completeness.

In order to change the branch, there is a configuration file called `configure.ac`. This is a template which is used by the configuration script when automake is executed. There are three options:

`-without-prefetching` disables prefetching of files and thus sets the branch to SMT (default: no and thus MTP mode).

`-with-timing` enables timing (default: no). Supported times are total, compare, hashing, accumulated time for waiting for ram and file, reading from disk. It also provides throughput for hashing (MB/s) and comparing (items/s).

`-with-stats` enables statistics (default: no). Currently only two state variables are added, waiting for a file and waiting for space in ram table.

2) RAM Table: `RAM_table` is the class responsible for holding files and synchronizing threads. Files are presented with the `ram_file` class, which provides functionality for reading files from the hard disk and processing them using the hash algorithm interface. `ram_table` uses two semaphores[9] for realizing the producer/consumer model. One semaphore is used for waiting for free space in RAM table and the other for waiting for available/prefetched files in RAM table. POSIX and Windows semaphores are supported through macros expanded during compilation.

The processing of the files in the table is based on two indicies called fi and pi. fi is the amount of files within the table and set by the prefetcher, i.e., after every insertion into the table fi increases by one. pi is the index of the worker threads. Thus every time a worker thread fetches a new file from the table pi increases. As a consequence, if $pi \geq fi$, threads have to wait for data.

To avoid race hazards we use the OpenMP 3.1 capture clause. Thus a thread can take the current index and increase the global index, in a single atomic operation. This way threads work with RAM files without the need for locking or critical sections.

3) Interfaces: The framework accesses all hashing algorithms and modes through self-made interfaces and thus every developer can add own hashing algorithm with a few lines of code. Realizations of interfaces are written in their own `*.hpp` file and included in the interface implementation file.

The hashing interface `hash_alg.cpp` also provides two extensions one for hashing algorithms with character output and the other for byte output. The difference between these two are functions for printing and saving a digest buffer.

Member variables of the class are:

Type of output - could either be hex or string. For instance MD5 results in a buffer holding a byte array, which needs to be converted to string.

Length of hash digest - is used to print the hash value.

Minimum file size - is necessary as some hashing algorithms have a minimum file size requirement. For instance `ssdeep` needs 4096 bytes.

Listing 1 and Listing 2 show the necessary changes for adding the `ssdeep` algorithm.

1) All hashing algorithms are implemented in their own file with the name `hash_alg_NAME.hpp`.

```
class hash_alg_ssdeep: public
    hash_alg_char_output{
public:
  int hash(uchar *in, uint inlen, uchar **out){
    *out = get_out();
    return (NULL == fuzzy_hash_buf_r((const
        uchar*)in, inlen, *out))
         ? -1: FUZZY_MAX_RESULT;
  };

  int cmp(uchar *a, uchar *b, uint len){
    return fuzzy_compare_r(a,b);
  };

  hash_alg_ssdeep(): hash_alg_char_output(){
    type          = HA_SSDEEP;
    max_result_size =
    hash_digest_size = FUZZY_MAX_RESULT;
    min_file_size = SSDEEP_MIN_FILE_SIZE;
  };
};
```

Listing 1. Framework extension for ssdeep.

2) Add new case in interface initialization function for `ssdeep`.

```
if( 0 == htype.compare(0, 6, "ssdeep")){
   h = new hash_alg_ssdeep();
}
```

Listing 2. Initializing hashing interface for ssdeep.

Currently the framework includes several cryptographic hash functions and three similarity preserving hashing algorithms. We included MD5, SHA1, SHA2, SHA3 and RIPEMD160 from the OpenSSL library and added `ssdeep`, `sdhash` and `mrsh-v2` with source code.

The `mode.h` interface allows the framework to operate in different ways, after it's compiled. The interface itself consist of 3 virtual functions, that represent the 3 steps of the framework - hashing of files, comparing digests and printing results/digests.

4) Coding optimizations: The following optimizations reduce the number of buffer allocations during the execution and brings two advantages:

1) Pre-allocation of all digest buffers reduces execution time, as there are less calls of `new[]`.
2) The Framework memory footprint is also reduced, because all digest buffers are grouped into one linear buffer. For instance, the GNU C library uses a header (2 words - 8b/32bits and 16b/64bit systems) for each memory block. If we allocate a digest for MD5 (16b), we will have another 16b (on 64bit systems) of OS administrative data (header).[10]

Both optimizations are only available for hashing algorithms with static hash value length. In the case of `mrsh-v2` and `sdhash` which have a variable hash value length we can not allocate the linear digest buffer before hashing is done.

E. Future work

An improvement to the implementation will be the addition of a balancing function. Under balancing we mean changing the order of files, in which they are processed, to reduce waiting for free table space and fragmentation (empty space in table). A simple example is shown below.

```
A(8), B(4), C(3), D(2), E(4), F(5) #Files order
TBL(0/10) #Table of size 10 with 0 space used
------------------------------------
T1:PREF(A) -> TBL(8/10)
```

[9]http://en.wikipedia.org/wiki/Semaphore_(programming); last accessed 13.01.2013

[10]Example take from http://lwn.net/Articles/257209/ at the end of Sec. 7.3; last accessed 13.01.2013

```
T1:PREF(B) -> TBL(8/10) -> WAIT(4) #Wait because
    only space of 2 is available
T2:HASH(A) -> TBL(0/10)
----------------------------------
A(8), D(2), B(4), F(5), C(3), E(4) #Files order
    after balancing
```

Listing 3. RAM table balancing example.

V. Experimental results & assessment

To assess our framework every test uses the t5-corpus[11] [17, sec. 4.1] containing 4457 files of the file types given in Table II. The unzipped size of these files is 1.78GB which corresponds to an average file size of 418,91KB. All following tests are based on `ssdeep-2.9` and `sdhash-2.3`.

	jpg	gif	doc	xls	ppt	html	pdf	txt
Amount	362	67	533	250	368	1093	1073	711

TABLE II
Statistic of the t5-corpus.

All binaries were compiled using the same compiler and configuration options. To compiler flags we added `-g0` to disable debugging, `-O2` to enable second level of optimization and `-march=native` to allow usage of CPU specific instructions.

The test environment was a server having the following benchmark data:

```
CPU: 2xIntel(R) Xeon(R) E5430 2.66GHz x 4 cores
HDD: Seagate® ES™ Series 250GB(SATA2) 8MB Cache 7200RPM
RAM: 8x2GB DDR2 FB-DIMM 667 MHz
KERNEL: Linux 2.6.32-279.11.1.el6.x86_64
GCC: gcc-4.4.6-4.el6.x86_64
```

In general there are three different times[12]:

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

Since our framework improves the whole processing we are interested in the `user` and `sys` time. Due to the fact that we used a dedicated system we simply take the `real` time (we proofed that `user + sys = real`). The time was measured using the Linux `time` command.

A. Overall runtime efficiency

This section demonstrates the general improvement of using the framework wherefore we compared the original implementation against MTP. Both tests set `-t 2` which indicates one prefetching thread and one working thread.

Test one (T1) analyzes `ssdeep` in detail. The results are listed in Table III. Using SMT improves the basic algorithms by approximately 17.5% which is in contrast to our expectations that it halves the time. The MTP proceeding shows an improvement of nearly 40%. The lower speedup of SMT is due to the lack of data in RAM. Having two threads means there will be two times more request for file data, but still having the same disk throughput. This way threads are being underfed and they have to idle. In the case of MTP we have a linear system - one reader and one hasher - which reduces the idle times of each thread.

	Time	Difference	Terminal command
original	83.67s	100.00%	$ ssdeep -r t5
SMT	69.05s	82.52%	$ pfh -d t5 -t 2 -h ssdeep
MTP	52.19s	62.37%	$ pfh -d t5 -t 2 -h ssdeep

TABLE III
T1: Run time efficiency with ssdeep using standard output.

Test two shows the improvement for different algorithms. All output was sent to `/dev/null` to eliminate any execution time deviation caused by printing. Table IV shows the results.

The main result of T2 is that prefetching is very useful for SPH algorithms which are more computationally intensive than cryptographic hash functions. Using MTP we achieved similar run times for all algorithms expect `sdhash`. This shows that the limiting fact in this case is the underlying hardware.

	`MD5`	`SHA-1`	`mrsh-v2`	`ssdeep`	`sdhash`
original	51.65s	52.35s	75.61s	83.67s	145.38s
MTP	51.74s	51.64s	51.79s	52.19s	89.09s

TABLE IV
T2: Run time efficiency of different hashing algorithms.

Table V presents the results for T3 which runs in `FULL` mode. Thus besides the hash value generation there is also an all-against-all comparison.

In case of `ssdeep` which is described in rows 1 and 2 we obtained an improvement of nearly 45%. For `sdhash` (rows 3 and 4) the results are even better where we stopped the all-against-all comparison after 69min and MTP mode only needed 186s.

[11]http://roussev.net/t5/t5-corpus.zip; last accessed 27.01.2013

[12]http://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1; last accessed 19.12.2012

[13]We removed the `-h` option from both `pfh` commands for a better readability.

	Time	Difference	Terminal command[13]
ssdeep	119.21s	100.00%	$ ssdeep -d -r t5
MTP	68.34s	57.33%	$ pfh -c full -d t5 -t 8 -m 128
sdhash	>69m	-	$ sdhash -r -g -p 8 t5
MTP	186,56s	-	$ pfh -c full -d t5 -t 8 -m 128

TABLE V
T3: RUN TIME EFFICIENCY WITH OF FULL MODE [DEFAULT T=2].

B. Impact of multiple cores

In the following we discuss the influence of multiple cores. Thus we invoke the framework by

```
$ pfh -h ALG -c hash -d t5 -m 256 -t XX > /dev/null
```

where XX is the amount of cores/threads and ALG the used algorithm.

Our test includes two runs denoted by R1 and R2 which are shown in Table VI and Table VII, respectively. The peculiarity is that we perform both runs in immediate succession and thus the files where still cached in R2.

R1 demonstrates that multiple cores is especially important for slower algorithms like sdhash. For fast algorithms, e.g., mrsh-v2, it doesn't scale well as the underlying hardware is to slow. R2 simulates fast hardware as all files are cached. Due to the fact that all files are cached the prefetcher thread is dispensable and thus SMT is faster.

		t=2	t=4	t=8
mrsh	SMT	64.03s	66.39s	67.14s
	MTP	51.79s	51.81s	52.02s
sdhash	SMT	89.33s	72.03s	68.14s
	MTP	89.09s	51.90s	52.08s

TABLE VI
R1: RUN TIME EFFICIENCY HAVING DIFFERENT AMOUNT OF THREADS.

		t=2	t=4	t=8
mrsh	SMT	10.15s	5.30s	2.92s
	MTP	17.68s	6.23s	2.90s
sdhash	SMT	48.42s	24.98s	11.83s
	MTP	88.15s	31.12s	15.07s

TABLE VII
R2: RUN TIME EFFICIENCY HAVING AMOUNT OF THREADS AND CACHED DATA.

As a conclusion we can say that in the first case the underlying hardware is to slow. To be more precise, the hard disk is to slow, the prefetcher thread cannot fill the RAM table and thus the worker threads have to idle. Having a SSD device or RAID system it should scale better because of higher throughputs.

C. Impact of different memory sizes

This section shows the impact of different RAM table sizes wherefore we invoke the framework by

```
$ pfh -h mrsh -c hash -d t5 -m XX -t 4 > /dev/null
```

where XX is the amount of memory in megabytes for the RAM table.

	m=128	m=256	m=512
SMT	66.36s	66.39s	66.20s
MTP	51.62s	51.81s	51.72s

TABLE VIII
RUN TIME EFFICIENCY HAVING DIFFERENT MEMORY SIZES.

Table VIII shows that the size of the RAM table does not influence the run time efficiency which rely to the two facts given at the end of Sec. IV-A. In general there are two possibilities. On the one hand there is a 'slow' hashing algorithm and the prefetching thread is faster. Thus the RAM table is always full because as soon as the worker thread fetches a file, the prefetcher adds the next one. The limiting source is the algorithm run time efficiency.

On the other hand there is a fast hashing algorithm and the worker threads are faster. Thus the RAM table is always 'empty' because as soon as a new fill is added, one worker processes it. The limiting source is the underlying hardware.

D. Impact of multiple prefetchers

Although the number of prefetcher threads is adjustable, tests showed that the default setting of 1 is the best choice. Table IX verifies that having two prefetchers worsen the runtime by 15% due to more overhead.

Time	Difference	Terminal command
52.05s	100.00%	$ pfh -t 8 -c hash -d t5 -h md5
60.09s	115.44%	$ pfh -t 8 -c hash -d t5 -h md5 -p 2

TABLE IX
IMPACT OF TWO PREFETCHING THREADS

E. Impact to a forensic investigation

In the following we analyze the improvement with respect to real world scenarios. Therefore we took two own working stations as given in Table X and used the results from this section to do a projection.

	Files	Size	Avg. size
Mac OSX	322,531	100.92GB	328.08KB
Windows 7	139,303	36.55GB	275.13KB

TABLE X
SUMMARY OF TWO OWN WORKSTATIONS.

Based on the findings from Sec. V-A we can estimate the runtime. The results from Table XI.

	Size	stand-alone	SMT	MTP
Mac OSX	100.92GB	99min 51sec	73min 43sec	56min 43
Windows 7	36.55GB	36min 10sec	26min 42sec	20min 32

TABLE XI
INVESTIGATION TIME USING SSDEEP ON TWO OWN WORKSTATIONS.

F. Distinction to existing parallelization tools

There are a couple of tools that execute commands, scripts or programs in parallel. Thus this section is a comparison of our framework against *parallel*[14] and *Parallel Processing Shell Script*[15] (abbreviated ppss), as the handling is easy. Both tools work on local cores (not distributed) with multi-threading and distribute the workload automatically to different threads.

The main conclusion from the results in Table XII is that MTP outperforms existing tools/scripts. Both analyzed tools do a simple multi-threading and thus we expect approximately results than with SMT which is true for *parallel*. In case of *ppss* the performance really worse with is due to a lot of I/O operations - *ppss* saves its state to hard drive in text files.

	Time	Difference	Terminal command
original	83.67s	100.00%	$ ssdeep -r t5 > /dev/null
ppss	337.11s	402.90%	$ ppss -p 8 -d t5 -c 'ssdeep '
parallel	69.81s	83.43%	$ parallel ssdeep – data/t5/*
SMT	67.55s	80.73%	$ pfh -t 8 -c hash -d t5 -h ssdeep
MTP	52.04s	62.20%	$ pfh -t 8 -c hash -d t5 -h ssdeep

TABLE XII
COMPARISON OF WITH DIFFERENT PARALLELIZATION TOOLS USING `SSDEEP`.

We verified our assumption that the slowness of *ppss* is due to the I/O bound by hashing 16 large files each having 1.2GB. In this case there are only a few writing operations. The results are given in Table XIII. As we can the the performance increased. However, it is still slower than the original implementation which is due to hard disk reading operations. Each threads reads the file whereas using `ssdeep` as stand-alone reads sequentially.

	Time	Difference	Terminal command
ssdeep	264s	100%	$ ssdeep -r t5.bz
ppss	342s	129%	$ ppss -p 8 -d t5.gz -c 'ssdeep '

TABLE XIII
COMPARISSON OF PFH, PPSS AND STAND-ALONE TOOLS WITH 16XT5.GZ

VI. CONCLUSION

Currently hashing a whole file system could be very time consuming doing it single threaded. With this paper we took the challenge of improving current practice for hash value generation and comparison with forensic purposes and provide a reusable framework. The results show an improvement of over 40% for an all-against-all comparison compared to the standard `ssdeep` algorithm. The design of the framework allows to add any other generic algorithm. Additionally we showed that in a real world scenario the investigation time could be improved a lot, e.g., from 1h 39 min to 56min without acquiring any extra hardware.

At the moment the framework already contains several cryptographic and non-cryptographic hash functions. Besides traditional hash functions we also included `ssdeep`, `sdhash` and `mrsh-v2`. Within an upcoming version we like to improve the drawback that only files smaller than the RAM table can be hashed.

[14]http://www.gnu.org/software/parallel/; last accessed 14.01.2013
[15]http://code.google.com/p/ppss/; last accessed on 13.01.2013

VII. ACKNOWLEDGMENTS

The research leading to these results has received partial funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 257007.

We thank Nuno Brito from the Serco Services and European Space Agency (Darmstadt, Germany) for valuable discussions and good ideas.

REFERENCES

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," in *ACM Transactions on Graphics*, 2009.
[2] C. Altheide and H. Carvey, *Digital Forensics with Open Source Tools: Using Open Source Platform Tools for Performing Computer Forensics on Target Systems: Windows, Mac, Linux, Unix, etc.* Syngress Media, May 2011.
[3] H. Baier and F. Breitinger, "Security Aspects of Piecewise Hashing in Computer Forensics," *IT Security Incident Management & IT Forensics (IMF)*, pp. 21–36, May 2011.
[4] A. Baxter, "Ssd vs hdd," *http://www.storagereview.com/ssd_vs_hdd*, 2012.
[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.
[6] F. Breitinger and H. Baier, "Performance Issues about Context-Triggered Piecewise Hashing," in *3rd ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, vol. 3, October 2011.
[7] ——, "A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance," *ADFSL Conference on Digital Forensics, Security and Law*, May 2012.
[8] ——, "Similarity Preserving Hashing: Eligible Properties and a new Algorithm MRSH-v2," *4th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, October 2012.
[9] L. Chen and G. Wang, "An Efficient Piecewise Hashing Method for Computer Forensics," *Workshop on Knowledge Discovery and Data Mining*, pp. 635–638, 2008.
[10] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Forensic Research Workshop (DFRWS)*, vol. 3S, pp. 91–97, 2006.
[11] A. Menezes, P. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.
[12] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, p. 4, 1965.
[13] NIST, "National Software Reference Library," May 2012. [Online]. Available: http://www.nsrl.nist.gov
[14] L. C. Noll. (2001) Fowler / Noll / Vo (FNV) Hash. Last accessed on 2012-07-05. [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/index.html
[15] R. Rivest, "The MD5 Message-Digest Algorithm," 1992.
[16] V. Roussev, "Data fingerprinting with similarity digests," *Internation Federation for Information Processing*, vol. 337/2010, pp. 207–226, 2010.
[17] ——, "An evaluation of forensic similarity hashes," *Digital Forensic Research Workshop*, vol. 8, pp. 34–41, 2011.
[18] V. Roussev, G. G. Richard, and L. Marziale, "Multi-resolution similarity hashing," *Digital Forensic Research Workshop (DFRWS)*, pp. 105–113, 2007.
[19] K. Seo, K. Lim, J. Choi, K. Chang, and S. Lee, "Detecting Similar Files Based on Hash and Statistical Analysis for Digital Forensic Investigation," *Computer Science and its Applications (CSA '09)*, pp. 1–6, December 2009.
[20] SHS, "Secure Hash Standard," 1995.
[21] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*. Springer Berlin Heidelberg, February 2007, vol. 1.

[22] C. Walter, “Kryder’s law.” [Online]. Available: http://www.scientificamerican.com/article.cfm?id=kryders-law&ref=sciam

[23] S. Wörthmüller, “Multithreaded file i/o,” *http://www.drdobbs.com/parallel/multithreaded-file-io/220300055*, 28. September 2009.