Reducing the Time Required for Hashing Operations

Frank Breitinger, Kaloyan Petrov

To cite this version:

Frank Breitinger, Kaloyan Petrov. Reducing the Time Required for Hashing Operations. 9th International Conference on Digital Forensics (DF), Jan 2013, Orlando, FL, United States. pp.101-117, 10.1007/978-3-642-41148-9_7 . hal-01460624

HAL Id: hal-01460624
https://inria.hal.science/hal-01460624

Submitted on 7 Feb 2017

Chapter 7

REDUCING THE TIME REQUIRED FOR HASHING OPERATIONS

Frank Breitinger and Kaloyan Petrov

Abstract Due to the increasingly massive amounts of data that need to be analyzed in digital forensic investigations, it is necessary to automatically recognize suspect files and filter out non-relevant files. To achieve this goal, digital forensic practitioners employ hashing algorithms to classify files into known-good, known-bad and unknown files. However, a typical personal computer may store hundreds of thousands of files and the task becomes extremely time-consuming. This paper attempts to address the problem using a framework that speeds up processing by using multiple threads. Unlike a typical multithreading approach, where the hashing algorithm is performed by multiple threads, the proposed framework incorporates a dedicated prefetcher thread that reads files from a device. Experimental results demonstrate a runtime efficiency of nearly 40% over single threading.

Keywords: File hashing, runtime performance, file handling, prefetching

1. Introduction

The availability and use of electronic devices have increased massively. Traditional books, photos, letters and LPs have become ebooks, digital photos, email and music files. This transformation has also influenced the capacity of storage media, increasing from a few megabytes to terabytes. Thus, the amount of data gathered during digital forensic investigations has grown rapidly. To solve this data overload problem, it is necessary to employ automated techniques to distinguish relevant from non-relevant information.

To cope with the massive amount of data, digital forensic practitioners often use automated preprocessing that groups files into three categories: known-good, known-bad and unknown files. Operating system files and common application binaries are generally deemed to be known-good files and need not be inspected. The steps involved in classifying a file are to hash the file, compare the hash value against a hash database, and put the file into one of the three categories.

Because of the amount of data to be processed, runtime efficiency is an important issue. Thus, one property of hashing algorithms is the ease of computation, which is met by popular algorithms such as SHA-1 and MD5.

Meanwhile, the performance of modern hardware has increased considerably [11]. The addition of multiple cores and powerful GPUs may allow algorithm parallelization [1], but the principal bottleneck often involves loading data from files [19]. One approach for speeding up reading/writing is to use RAID systems that combine multiple disk drive components into a logical unit. Another approach is to use solid state drives (SSDs) that have higher throughputs than conventional hard disks [4]. However, because RAID and SSD technologies are not very widespread, a practical approach is to develop intelligent file handling solutions.

This paper presents a new framework for file handling during hashing. Instead of having several threads and/or cores that perform hashing independently, one thread is used for reading and the remaining threads are used for hashing. The framework can easily be used for other hashing algorithms or other tasks involving high data throughput.

2. Background

Hash functions have two basic properties, compression and ease of computation [10]. Hash functions are used in cryptography, databases [18] and for file identification [2]. In addition to traditional hash functions such as FNV [14], cryptographic hash functions such as SHA-1 [12] and MD5 [15], there are also similarity preserving hash methods such as `ssdeep` [9], `sdhash` [16] and `mrsh-v2` [7]. Similarity preserving hash algorithms are typically slower than traditional hash algorithms [7]. In order to optimize their runtime efficiency, most researchers have focused on algorithmic improvements [6, 8].

The most popular use case of cryptographic hash functions in digital forensics is known file detection. To detect files based on their hashes, it is necessary to rely on a database that includes a reference for each input file and its hash value. The most well-known database is the Reference Data Set (RDS) from the National Software Reference Library (NSRL) [13]. If the hash value of a file from a storage medium matches the hash value in the database, one can assume with some confidence that the file is the known file.

When a storage medium is examined, forensic software is used to hash the input, perform look-ups in the RDS and filter the non-relevant files. This reduces the amount of data that the investigator has to examine manually. Typically, the forensic software uses one thread to read and hash the file.

2.1 Similarity Preserving Hashing

Three similarity preserving hashing algorithms are used to demonstrate the effectiveness of the proposed framework:

- **ssdeep:** This algorithm [9], also called context triggered piecewise hashing, divides an input into approximately 64 pieces and hashes each piece separately. Instead of dividing the input into blocks of fixed length, it is divided based on the current context of seven bytes. The final hash value is the concatenation of all of the piecewise hashes where the context triggered piecewise hashing only uses the six least significant bits of each piecewise hash. This results in a Base64 sequence of approximately 64 characters.

- **sdhash:** This algorithm identifies "statistically-improbable features" using an entropy calculation [16]. The characteristic features, corresponding to a sequence of 64 bytes, are then hashed using SHA-1 and inserted into a Bloom filter [5]. Files are similar if they share identical features.

- **mrsh-v2:** This algorithm is based on `ssdeep`. The principal improvement is the removal of the restriction of 64 pieces, which creates a security issue [3]. Furthermore, instead of using a Base64 hash, `mrsh-v2` creates a sequence of Bloom filters [7].

2.2 Hash Function Run Time Efficiency

This section compares the runtime efficiency of the hashing algorithms included in the framework. All the tests used a 500 MiB file from `/dev/urandom` and the times were measured using the `time` command and the algorithm CPU-time (user time).

Table 1 presents the runtime efficiency comparison of the traditional cryptographic hash functions SHA-1 and MD5 versus the three similarity preserving hashing algorithms considered in this work. The results show that the traditional cryptographic hash functions outperform the similarity preserving hashing algorithms.

Table 1. Runtime efficiency comparison of hash functions.

	SHA-1	**MD5**	**mrsh-v2**	**ssdeep 2.9**	**sdhash 2.0**
Time	2.33 s	1.35 s	5.23 s	6.48 s	22.82 s
$\frac{\text{Algorithm}}{\text{SHA-1}}$	1.00	0.58	2.24	2.78	9.78

3. Parallelized File Hashing Framework

Our parallel framework for hashing (`pfh`) optimizes file handling during hashing. It is written in C++ and uses OpenMP 3.1 for multithreading; it is available at `www.dasec.h-da.de/staff/breitinger-frank`.

The framework is divided into two branches – simple multithreading (SMT) and multithreading with prefetching (MTP). SMT is used for comparison purposes and shows the benefits of the prefetcher.

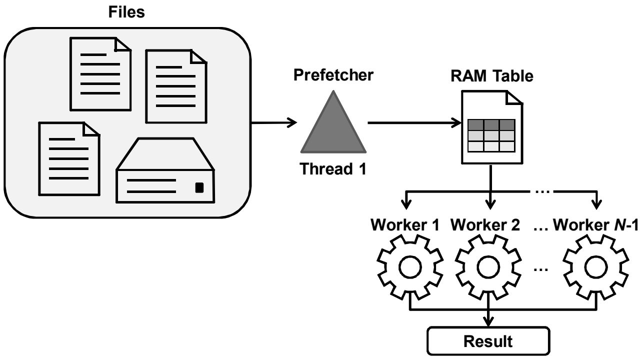

Figure 1. Operations of the framework.

Unlike traditional approaches where hash functions request a file and process it, the framework shown in Figure 1 includes a prefetching mechanism. The prefetcher handles file reading and is responsible for communications between the hard disk and RAM. The idea is that the critical resource bottleneck, the hard disk, should "work" all the time. Thus, the prefetcher produces an ongoing file request.

All the files are placed in RAM, which uses a RAM table to track the available storage. All remaining threads are "workers" and process the files from RAM using the hashing algorithms. After hashing the files, the outputs are denoted by result.

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

- If the hashing algorithm is fast, the worker threads are faster than the prefetching process and the workers must idle. However, the hard disk is at its limit and cannot process any faster.

- If the hashing algorithm is slow, the RAM table becomes full and cannot store any more files. This causes the prefetcher to idle. In this case, the prefetcher thread could turn into a normal worker and help process the files in the RAM table. Implementing this functionality is a part of our future work.

An alternative solution is to use a distributed system to further increase the performance. As will be demonstrated later, the limiting resource is the hard disk access time and not the computational power of the system. Thus, there is no improvement using a distributed system to hash inputs.

3.1 Command Line Parameters

Before describing the details of `pfh`, we introduce the command line options that allow rough configuration. Note that N denotes the number of processor cores in the system and the value of option `-p` is denoted by P where $P < N$.

- **c:** Mode of framework operation [optional].

- **d:** Directory to be hashed or file with digests [required].

- **r:** Recursive mode for directory traversal [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 memory in MB [default is 20 MB].

A drawback is that all files larger than the RAM table are skipped. This problem will be corrected in a future version of the framework.

The framework operates in four modes:

- **HASH:** All the files are hashed using the specified algorithm and the results are printed to the standard output [default].

- **FULL:** The framework does an all-against-all comparison of all the files in `directory`.

- **<DIGEST>:** All the files in `directory` are hashed and compared against `DIGEST`, which is a single fingerprint.

  If parameter `-d` is a fingerprint file, then the framework compares `DIGEST` against all fingerprints in the file and skips hashing.

- **<FILENAME>:** `<FILENAME>` is replaced by a path to a file containing a list of valid hash values. The framework hashes all the files in `directory` and compares them against the list. This is similar to the `-m` option of `ssdeep`. If the signature is found in the list, then it is a valid match.

The following command executes the framework in the default mode using a RAM table of size 256 MB.

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

The `t5` directory is traversed recursively and all the hashes are sent to the standard output. If the `-t` and `-p` options are not specified, the program has $P = 1$ prefetching thread and $N - P$ hashing threads where N is the number of available processor cores.

3.2 Processing

Upon initialization, the framework creates the four building blocks: (i) options parsed; (ii) hashing interface created; (iii) RAM table created; and (iv) mode of operation. The directory containing the files is traversed and each file that passes the file size and access rights filter is added to the files-to-be-hashed-list.

The framework processing involves three stages: (i) reading/hashing files; (ii) comparing hash values; and (iii) presenting results whereby only the first and second stages are executed with multiple threads while the third stage is performed sequentially. Threads are created before the first stage and are finalized at the end of second stage. Thus, no time is lost for thread management (fork/join) during execution.

1. **Reading/Hashing Files:**

   - **SMT Branch:** Each of the N threads places its file in the RAM table and hashes it. All the threads continue until there are no more files in the queue.
     After being assigned to a role (reading or hashing), the 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:** Each thread receives a role assignment and begins execution (e.g., P prefetchers and $N - P$ hashing threads).

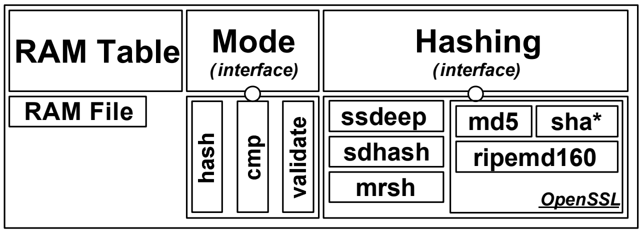

Figure 2. Objects in the framework.

2. **Comparing Hash Values:** This stage executes in parallel using the OpenMP “parallel for” clause, in which threads work on chunks of the global compare iterations.

   Scores from the comparison are held in an array, because if the threads print to screen, they have to synchronize and the speedup of parallelism is lost.

3. **Presenting Results:** The file path, hash value and the results are sent to the standard output.

3.3 Implementation Details

In addition to the two branches SMT/MTP and the operation modes, the framework comprises two objects, the RAM table and the hashing interface, which are shown in Figure 2.

SMT and MTP. Although SMT does not outperform MTP, we describe its implementation for completeness.

A configuration file called `configure.ac` is used to change the branch. This template is used by the configuration script when automake is executed. There are three options:

- **without-prefetching:** This option disables file prefetching and sets the branch to SMT (default: no, i.e., MTP mode).
- **with-timing:** This option enables timing (default: no). The supported times are total, compare, hashing, accumulated time for waiting for RAM and file, and reading from the disk. The option also provides throughput for hashing (MB/s) and comparisons (items/s).
- **with-stats:** This option enables statistics (default: no). Currently, only two state variables are added, waiting for a file and waiting for space in the RAM table.

RAM Table. The RAM table is the class responsible for holding files and synchronizing threads. Files are sent to 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 to implement the producer-consumer model. One semaphore is used to wait for free space in the RAM table and the other is used to wait for available/prefetched files in the RAM table.

The processing of the files in the table is based on two indices, fi and pi. The index fi denotes the number of files in the table and is set by the prefetcher; it increases by one after every insertion into the table. The index pi is for the worker threads; it increases by one every time a worker thread fetches a new file from the table. Thus, if $pi \geq fi$, threads have to wait for data.

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

Interfaces. The framework accesses all hashing algorithms and modes via interfaces, enabling developers to add their own hashing algorithms. Realizations of the interfaces are written in their own *.hpp files and are included in the interface implementation file. Currently, the framework includes MD5, SHA1, SHA2, SHA3 and RIPEMD160 from the OpenSSL library and the ssdeep, sdhash and mrsh-v2 similarity hashes.

The hashing interface hash_alg.cpp provides two extensions, one for hashing algorithms with character outputs and the other for byte outputs. The two extensions differ in the functions used to print and save a digest buffer. Member variables of the class are:

- **Output Type:** This could be a hex value or a string. For instance, MD5 yields a buffer holding a byte array, which has to be converted to a string.
- **Hash Digest Length:** This is required to print the hash value.
- **Minimum File Size:** This is required because some hashing algorithms have a minimum file size requirement (e.g., ssdeep requires 4,096 bytes).

Each hashing algorithm is implemented in its own file with the name hash_alg_NAME.hpp. Figures 3 and 4 show the changes needed to the ssdeep algorithm.

The mode.h interface allows the framework to operate in different ways after it is compiled. The interface itself consists of three virtual

```
class hash_al_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_compar_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;
  };
};
```

Figure 3. Framework extension for `ssdeep`.

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

Figure 4. Initializing the hashing interface for `ssdeep`.

functions that represent the three steps of the framework: hashing files, comparing digests and printing results/digests.

Code Optimizations. Two code optimizations reduce the number of buffer allocations during execution. The first optimization pre-allocates all digest buffers; this reduces execution time because there are fewer calls of `new[]`. The second optimization reduces the memory footprint by grouping all the digest buffers into a single linear buffer. For instance, the GNU C library uses a header (two words for 8 bytes/32-bit and 16 bytes/64-bit systems) for each memory block. If a digest is allocated for MD5 (16 bytes), there is another 16 bytes (in 64-bit systems) of operating system administrative data (header).

Both optimizations are only available for hashing algorithms with a static hash value length. In the case of `mrsh-v2` and `sdhash`, which have a variable hash value length, the linear digest buffer cannot be allocated before hashing.

```
01: A(8), B(4), C(3), D(2), E(4), F(5) #Files order
02: TBL(0/10) #Table of size 10 with 0 space used
03: ------------------------------------
04: T1:PREF(A) -> TBL(8/10)
05: T1:PREF(B) -> TBL(8/10) -> WAIT(4) #Wait because only space of 2 is
                                        available
06: T2:HASH(A) -> TBL(0/10)
07: ------------------------------------
08: A(8), D(2), B(4), F(5), C(3), E(4) #Files order after balancing
```

Figure 5. RAM table balancing example.

3.4 Future Work

A future enhancement to the implementation will be the addition of a load balancing function. Load balancing would change the processing order of files, reducing both the waiting time for free table space and the fragmentation (empty space in table). A simple example is shown in Figure 5.

Table 2. Statistics of the t5 corpus.

	jpg	**gif**	**doc**	**xls**	**ppt**	**html**	**pdf**	**txt**
Amount	362	67	533	250	368	1,093	1,073	711

4. Experimental Results

The experimental evaluation of `pfh` used the t5 corpus [17], which contains 4,457 files of the types shown in Table 2. The unzipped size of files was 1.78 GB, corresponding to an average file size of 418.91 KB. The following tests are based on `ssdeep-2.9` and `sdhash-2.3`.

All the binaries were compiled using the same compiler and configuration options. The compiler flags included `-g0` to disable debugging, `-O2` to enable second level of optimization and `-march=native` to allow the use of CPU-specific instructions. The test environment was a server with the following components:

- **CPU:** Two Intel Xeon E5430 2.66 GHz × 4 cores
- **Hard Drive:** Seagate ES Series 250 GB (SATA 2) 8 MB Cache 7,200 RPM
- **RAM:** Eight 2 GB DDR2 FB-DIMM 667 MHz
- **Kernel:** Linux 2.6.32-279.11.1.el6.x86_64

Table 3. Runtime efficiency with `ssdeep` using standard output.

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

- **GCC:** gcc-4.4.6-4.el6.x86_64

Three execution times were recorded:

- **Real Time:** This is the wall clock time from start to finish of the call. It corresponds to the elapsed time, including the time slices used by other processes and the time that the process spends in the blocked state.

- **User Time:** This is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This coresponds to the actual CPU time used to execute the process. The time consumed by other processes and the time that the process spends in the blocked state do not count towards the user time.

- **Sys Time:** This is the amount of CPU time spent in the kernel within the process. This corresponds to the execution time spent in system calls within the kernel, as opposed to library code, which is still running in user space. Like the user time, this only includes the CPU time used by the process.

Since the framework improves the entirety of processing, only the real time is reported.

4.1 Runtime Efficiency

This section demonstrates the runtime improvement of the framework compared with the original implementation. The tests used the `-t 2` option, corresponding to one prefetching thread and one working thread.

Table 3 presents the results for `ssdeep`. Using SMT improves the basic hash algorithm runtime by approximately 17.5%. The MTP process 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 are twice as many requests for file data, but with the same disk throughput. The threads are underfed and are forced to idle. In the case of MTP, it is a linear system – one reader and one hasher – which reduces the idle time for each thread.

Table 4. Runtime efficiency of the hash algorithms.

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

Table 4 shows the runtime improvements for the hash algorithms. All the outputs were sent to `/dev/null` to eliminate any execution time deviation caused by printing. The main result is that prefetching has more impact on the similarity preserving hash algorithms, which are more computationally intensive than the cryptographic hash functions. MTP achieves similar runtimes for all the algorithms except `sdhash`. This shows that the limiting factor is the underlying hardware.

Table 5. Runtime efficiency with `FULL` mode [default `t` = 2].

	Time	**Difference**	**Terminal Command**
`ssdeep`	119.21 s	100.00%	`$ ssdeep -d -r t5`
MTP	68.34 s	57.33%	`$ pfh -h -c full -d t5 -t 8 -m 128`
`sdhash`	> 69 min	–	`$ sdhash -r -g -p 8 t5`
MTP	186.56 s	–	`$ pfh -h -c full -d t5 -t 8 -m 128`

Table 5 presents the results obtained using the `FULL` mode. This mode involves an all-against-all comparison in addition to hash value generation. In the case of `ssdeep` (rows 1 and 2), an improvement of nearly 45% was obtained. For `sdhash` (rows 3 and 4), the results are even better when the all-against-all comparison was stopped after 69 min and MTP only required 186 s.

4.2 Impact of Multiple Cores

This section explores the influence of multiple cores. The framework command line operation is:

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

where `XX` is the number of cores/threads and `ALG` is the hash algorithm.

The test includes two runs, denoted by R1 and R2, shown in Tables 6 and 7. In R2, all of the files were pre-cached from R1.

R1 demonstrates that using multiple cores is important for slower algorithms like `sdhash`. For the faster traditional hashing algorithms, it does not scale well because the underlying hard disk is too slow and the prefetcher thread cannot fill the RAM table. R2 simulates fast hardware

Table 6. R1: Runtime efficiency with different numbers of threads.

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

Table 7. R2: Runtime efficiency with different numbers of threads and cached data.

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

because all the files are cached. As a result, the prefetcher thread is dispensable and SMT completes in less time.

Table 8. Runtime efficiency with different RAM table sizes.

	m=128	**m=256**	**m=51**
SMT	66.36s	66.39s	66.20s
MTP	51.62s	51.81s	51.72s

4.3 Impact of Memory Size

Table 8 shows that the size of the RAM table does not influence runtime efficiency. In general, there are two possibilities. The first is that 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 thus the runtime efficiency of the algorithm.

The second possibility is that 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 file is added, one worker processes the file. The limiting source is thus the underlying hardware.

Table 9. Impact of two prefetching threads.

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

4.4 Impact of Multiple Prefetchers

Although the number of prefetcher threads is adjustable, tests showed that the default setting of one is the best choice. Table 9 verifies that having two prefetchers worsens the runtime by 15% due to the additional overhead.

Table 10. Summary of workstations used.

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

4.5 Impact on a Forensic Investigation

Next, we examine the improvement obtained using two workstations. The workstations listed in Table 10 were used. The results and projections are shown in Table 11.

Table 11. Investigation time using `ssdeep` on two workstations.

	Size	Standalone	SMT	MTP
Mac OSX	100.92 GB	99 min 51 s	73 min 43 s	56 min 43 s
Windows 7	36.55 GB	36 min 10 s	26 min 42 s	20 min 32 s

4.6 Parallelization Tool Comparison

This section compares `pfh` against Parallel and Parallel Processing Shell Script (`ppss`). Both Parallel and `ppss` execute commands, scripts or programs in parallel on local cores with multithreading and distribute the workload automatically to different threads.

Table 12 presents the results of using different parallelization tools. The main result is that MTP outperforms existing tools/scripts. Parallel and `ppss` both perform simple multithreading and should approximate

Table 12. Comparison of with different parallelization tools using `ssdeep`.

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

the SMT results. This holds true for Parallel, but in the case of `ppss`, the performance is worse due to an increase in I/O operations because `ppss` saves its state to the hard drive in the form of text files.

5. Conclusions

Current hashing techniques applied to entire filesystems are very time-consuming when implemented as single-threaded processes. The framework presented in this paper significantly speeds up hash value generation by including a separate prefetcher component. Experimental results demonstrate that an improvement of more than 40% is obtained for an all-against-all comparison compared with the standard `ssdeep` algorithm. In a real-world scenario, this results in a reduction in processing time from 1 hour and 39 minutes to 56 minutes without using any extra hardware.

The current implementation of the framework incorporates several cryptographic and similarity hash functions. Our future research will attempt to eliminate the limitation that only files smaller than the RAM table can be hashed.

Acknowledgements

This research was partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) under Grant No. 257007. We also thank Nuno Brito with Serco Services and the European Space Agency (Darmstadt, Germany) for valuable ideas and discussions.

References

[1] D. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. Owens and N. Amenta, Real-time parallel hashing on the GPU, *ACM Transactions on Graphics*, vol. 28(5), article no. 154, 2009.

[2] C. Altheide and H. Carvey, *Digital Forensics with Open Source Tools*, Syngress, Waltham, Massachusetts, 2011.

[3] H. Baier and F. Breitinger, Security aspects of piecewise hashing in computer forensics, *Proceedings of the Sixth International Conference on IT Security Incident Management and IT Forensics*, pp. 21–36, 2011.

[4] A. Baxter, SSD vs. HDD (`www.storagereview.com/ssd_vs_hdd`), 2012.

[5] B. Bloom, Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM*, vol. 13(7), pp. 422–426, 1970.

[6] F. Breitinger and H. Baier, Performance issues about context-triggered piecewise hashing, *Proceedings of the Third International ICST Conference on Digital Forensics and Cyber Crime*, pp. 141–155, 2011.

[7] F. Breitinger and H. Baier, Similarity preserving hashing: Eligible properties and a new algorithm `mrsh-v2`, *Proceedings of the Fourth International ICST Conference on Digital Forensics and Cyber Crime*, 2012.

[8] L. Chen and G. Wang, An efficient piecewise hashing method for computer forensics, *Proceedings of the First International Workshop on Knowledge Discovery and Data Mining*, pp. 635–638, 2008.

[9] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing, *Digital Investigation*, vol. 3(S), pp. S91–S97, 2006.

[10] A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Florida, 1997.

[11] G. Moore, Cramming more components onto integrated circuits, *Electronics Magazine*, pp. 114–117, April 19, 1965.

[12] National Institute of Standards and Technology, Secure Hash Standard, FIPS Publication 180-3, Gaithersburg, Maryland, 2008.

[13] National Institute of Standards and Technology, National Software Reference Library, Gaithersburg, Maryland (`www.nsrl.nist.gov`), 2012.

[14] L. Noll, FNV hash (`www.isthe.com/chongo/tech/comp/fnv/index.html`), 2012.

[15] R. Rivest, MD5 Message-Digest Algorithm, RFC 1321, 1992.

[16] V. Roussev, Data fingerprinting with similarity digests, in *Advances in Digital Forensics VI*, K. Chow and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 207–226, 2010.

[17] V. Roussev, An evaluation of forensic similarity hashes, *Digital Investigation*, vol. 8(S), pp. S34–S41, 2011.

[18] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*, Springer-Verlag, Berlin Heidelberg, Germany, 2010.

[19] S. Woerthmueller, Multithreaded file I/O, *Dr. Dobb's Journal*, September 28, 2009.