2015

An Empirical Comparison of Widely Adopted Hash Functions in Digital Forensics: Does the Programming Language and Operating System Make a Difference?

Satyendra Gurjar

Ibrahim Baggili
University of New Haven, ibaggili@newhaven.edu

Frank Breitinger
University of New Haven, fbreitinger@newhaven.edu

Alice E. Fischer
University of New Haven, AFischer@newhaven.edu

Publisher Citation

Gurjar, S., Baggili, I., Breitinger, F. and Fischer, A. (2015). An empirical comparison of widely adopted hash functions in digital forensics: does the programming language and operating system make a difference? Proceedings of the Conference on Digital Forensics, Security and Law, CDFSL 2015, Daytona Beach, Fla., May 19-21, 2015, pp. 57-68.

Comments

Dr. Ibrahim Baggili was appointed to the University of New Haven's Elder Family Endowed Chair in 2015.

AN EMPIRICAL COMPARISON OF WIDELY ADOPTED HASH FUNCTIONS IN DIGITAL FORENSICS: DOES THE PROGRAMMING LANGUAGE AND OPERATING SYSTEM MAKE A DIFFERENCE?

Satyendra Gurjar, Ibrahim Baggili, Frank Breitinger and Alice Fischer
Cyber Forensics Research and Education Group (UNHcFREG)
Tagliatela College of Engineering, ECECS Department
University of New Haven, West Haven CT, 06511
{agurj1, ibaggili, fbreitinger, afischer}@newhaven.edu

ABSTRACT

Hash functions are widespread in computer sciences and have a wide range of applications such as ensuring integrity in cryptographic protocols, structuring database entries (hash tables) or identifying known files in forensic investigations. Besides their cryptographic requirements, a fundamental property of hash functions is efficient and easy computation which is especially important in digital forensics due to the large amount of data that needs to be processed when working on cases. In this paper, we correlate the runtime efficiency of common hashing algorithms (MD5, SHA-family) and their implementation. Our empirical comparison focuses on C-OpenSSL, Python, Ruby, Java on Windows and Linux and C# and WinCrypto API on Windows. The purpose of this paper is to recommend appropriate programming languages and libraries for coding tools that include intensive hashing processes. In each programming language, we compute the MD5, SHA-1, SHA-256 and SHA-512 digest on datasets from 2 MB to 1 GB. For each language, algorithm and data, we perform multiple runs and compute the average elapsed time. In our experiment, we observed that OpenSSL and languages utilizing OpenSSL (Python and Ruby) perform better across all the hashing algorithms and data sizes on Windows and Linux. However, on Windows, performance of Java (Oracle JDK) and C WinCrypto is comparable to OpenSSL and better for SHA-512.

Keywords: Digital forensics, hashing, micro benchmarking, security, tool building.

1. INTRODUCTION

Cryptographic hash functions are critical to digital forensic science (DFS). Almost all tools written for forensic acquisition and analysis compute hash values throughout the digital forensic process to ensure the integrity of seized devices and data. For instance, to ensure the integrity of digital evidence in court, a forensic examiner traditionally computes the hash digest of the entire disk image that is then securely stored. When it becomes necessary to verify that a disk has remained intact without alteration after being acquired, a new hash digest is computed on the entire disk and compared against the stored hash digest. If both hashes coincide, we conclude that no alteration to the original drive took place during the acquisition process.

On the other hand, the availability and use of electronic devices has dramatically increased. Traditional books, photos, letters and records have become e-books, digital photos, e-mail and music files. This transformation has also influenced the capacity of storage media, increasing from a few megabytes to terabytes. According to the Federal Bureau of Investigation (FBI)'s Regional Computer Forensics Laboratory annual report in 2012 (*Regional Computer Forensics Laboratory. Annual report*, 2012), there was a 40% increase in amount of data analyzed in investigations. Due to the amount of data to be processed, runtime efficiency becomes an important and timely issue.To that end, automatic data filtration has become critical for speeding up investigations.

A common procedure known as file filtering is in use by today's digital forensic scientists and examiners, which requires hash functions. The procedure is quite simple:

1. compute the hashes for all files on a target device and
2. compare them to a reference database.

Depending on the underlying database, files are either filtered out (e.g., files of the operating system) or filtered in (e.g., known illicit content). A commonly used database for 'filtering out' data is the National Software Reference Library Reference Data Set (RDS) (*RDS Hashsets*, 2014) maintained by National Institute for Standards and Technologies (NIST).

Traditional hash functions can only match files exactly for every single bit. Forensic examiners frequently face the situation when they need to know if files are similar. For example, if two files are a different version of the same software package or system files or if the files partially match an image or video. Research has found utility for hash functions for finding similar files. Kornblum's Context Triggered Piecewise Hashing (CTPH) (Kornblum, 2006) and Roussev's similarity digest hashing (sdhash, (Roussev, 2010a)) have presented these ideas. These algorithms provide a probabilistic answer for similarities of two or more files. Although these algorithms are designed to detect similarity, they make use of traditional / cryptographic hash functions.

A common field of the application for hash functions is in digital forensics. Since this area has to deal with large amounts of data, the ease of computation (runtime efficiency) of hashing algorithms is very important.

In this paper we compare runtime efficiency of hashing algorithm implementations in multiple programming languages across different operating systems. Namely, we compare MD5, SHA-1, SHA-256 and SHA-512 in C, C#, Java, Python and Ruby on both Windows and Linux. While the DFS community has performed extensive research on hash function applications, little to no experimental work has been published with regards to the variance in the runtime efficiency of hash functions across different programming languages and libraries. This is of critical importance and may help scientists and practitioners alike when choosing a particular programming language if their forensic applications are hash function intensive.

2. RELATED WORK

Hash functions (e.g., SHA-1 (Gallagher & Director, 1995)) have a long tradition and are applied in various fields of computer science like cryptography (Menezes, van Oorschot, & Vanstone, 2001), databases (Sumathi & Esakkirajan, 2007, Sec. 9.6) or digital forensics (Altheide & Carvey, 2011, p.56ff). Garfinkel (Garfinkel, Nelson, White, & Roussev, 2010) also discussed small block forensics using cryptographic hash functions by calculating hashes on individual blocks of data rather than on entire files. Techniques described in his paper can be applied to data acquired from memory images as well. A hash based carving tool `frag_find` (*frag_find*, 2013) is used to find a MASTER file or fragments in a disk image using small block hashing. It computes hashes of small blocks (512 bytes) of MASTER files then compares it with disk image blocks in each sector.

In contrast to cryptographic hash functions, bytewise approximate matching do not have a long history and probably had its breakthrough in 2006 with an algorithm called context triggered piecewise hashing (CTPH). Kornblum (Kornblum, 2006) used this algorithm to identify similar files. The idea of CTPH is based on spamsum (*spamsum*, 2002-2009), a spam detection algorithm by Tridgell (Tridgell, 1999). The basic idea is behind it is simple: split an input into chunks, hash each chunk independently and concatenate the chunk hashes to a final similarity digest (a.k.a. fingerprint).

The `sdhash` tool[1] was introduced four years later (Roussev, 2010b) in an effort to address some of the shortcomings of `ssdeep`. Instead of dividing an input into chunks, the `sdhash` algorithm picks statistically improbable features to represent each object. A feature in this context is a byte sequence of 64 bytes, which is hashed using SHA-1 and inserted into a Bloom filter (Bloom, 1970). The similarity digest of the data object is a sequence of 256-byte Bloom filters, each of which represents approximately 10 KB of the original data.

Besides these two very prominent approaches, more tools published over the last decade `mrsh-v2` (Breitinger & Baier, 2012) seem to be promising since they use concepts from `sdhash` and `ssdeep`. In addition to the tools, Breitinger (Breitinger, Stivaktakis, & Baier, 2013) presented a testing framework entitled FRamework to test Algorithms of Similarity Hashing (FRASH) which is used to compare these algorithms – efficiency was one of the important metrics.

Saleem, Popov and Dahman (Saleem, Popov, & Dahman, 2011) presented a comparison of multiple security mechanisms including a hashing algorithm in accordance with Information Technology Security Evaluation Criteria (ITSEC). One of the criteria chosen in their analysis most relevant to this research is computational efficiency. In their experiments, they concluded that SHA-256 had the slowest average time. They also referred to a collision attack on MD5 (Wang & Yu, 2005) and SHA-1 (Wang, Yin, & Yu, 2005) and concluded SHA-256 and SHA-512 show more *Strength of Mechanism* compared to MD5 and SHA-1.

[1]`http://sdhash.org` last visited 2014-09-29.

3. METHODOLOGY

In this section, we first explain our experimental environment in Sec. 3.1 followed by an explanation of how we present our results in Sec. 3.2.

3.1 Experimental environment

In order to compute runtime, we generated files of multiple sizes from 2 MB to 1 GB using Python's `os.urandom` function. On UNIX-like systems, this Python function uses `/dev/urandom` and on Windows it uses CryptGenRandom to generate random binary data. In our experiments, the programs written in the respective languages take four command line arguments:

Warmup-count is the number of times to run a hash function before we start collecting elapsed time. This is to help programming languages like Java that have a Just-In-Time compiler to start, compile, and optimize code before we start collecting measurements.

repeat-count is the number of times we are going to run the hash function on the same data to collect the elapsed time. Elapsed time is collected for computing the digest only. The time to load the file from disk into the buffer is not included. For this experiment, we set $repeat - count = 10$ for each hashing algorithm and data file.

algorithm is the name of the hashing algorithm to be used for a run. We use MD5, SHA-1, SHA-256 and SHA-512 in our experiments.

data-file is the name of the data-file whose hash digest is to be computed. As we have data files of multiple sizes, each run computes the hash digest on every single data file.

Each program prints the elapsed time, repeat index and computed digest (for the verification of correctness of the program).

Table 1 shows the used hardware for our experiments where Table 2 describes the hashing algorithms. On Linux we tested the hashing algorithms using Java, Ruby, Python and C (with OpenSSL). For Windows we tested using Java, Ruby, Python, C# and C (with two libraries OpenSSL and WinCrypto). The source code of the experiment is available on github: `https://github.com/sgurjar/hashfun-benchmark`.

3.2 Data analysis and results

For each language, hashing algorithm and data size, we recorded the elapsed time of the $n = 10$ runs. Next, we computed the mean values for all runs. In addition, we wanted to identify the best curve/graph that represents this set of data points. More precisely, we wanted to identify the best coefficients of the linear equation $y = a+bx$ where we decided to use the Least Squares Mean[2] (LSM). According to LSM, we identified the coefficients as follows:

$$b = \frac{(n * \sum_{i=1}^{n} x_i * y_i) - ((\sum_{i=1}^{n} x_i) * (\sum_{i=1}^{n} y_i))}{(n * \sum_{i=1}^{n} x_i * x_i) - (\sum_{i=1}^{n} x_i)^2} \tag{1}$$

$$a = \frac{(\sum_{i=1}^{n} y_i) - (b * \sum_{i=1}^{n} x_i)}{n} \tag{2}$$

where x is an independent variable representing the size of the data we are computing the hash digest for, and y is the dependent variable representing the average elapsed time for a given language, algorithm and data size. b is called slope of the line and is a measurement of how well the implementation of an algorithm will scale in a programming language, i.e. the higher the slope the slower the implementation for large amounts of data.

4. ASSESSMENT AND EXPERIMENTAL RESULTS

We divided this section into five subsections. The first four subsections are named according to the tested algorithms MD5, SHA-1, SHA-256 and SHA-512 and present the detailed results of our experiments. The last section visually summarizes our results and discusses critical findings.

[2]`http://en.wikipedia.org/wiki/Least_squares` last visited 2014-09-29

Data	Avg. elapsed time in milli-Sec.			
in MB	C	Java	Python	Ruby
2	5	11	4	5
4	12	22	11	10
8	23	44	21	21
16	44	89	42	44
32	87	178	87	88
64	177	356	175	176
128	353	712	353	352
256	705	1420	704	706
512	1409	2848	1413	1407
640	1761	3523	1763	1765
768	2114	4252	2116	2119
896	2465	4978	2470	2470
1024	2820	5716	2823	2824

Table 3: Average elapsed time for MD5 on a Linux system.

To present our results, we decided to have three tables per algorithm:

1. The first table shows the average elapsed time in milliseconds for Linux dependent on the file size.

2. The second table shows the coefficients a and b using equations 1 and 2.

3. The third table shows the average elapsed time in milliseconds for Windows dependent on the file size.

The column header C indicates that C-OpenSSL is used while C (win) stands for the WinCrypto library.

4.1 MD5

The detailed results for the MD5 algorithms are shown in Table 3, 4 and 5.

As indicated by Table 3, languages using OpenSSL (C, Python and Ruby) showed similar performance on Linux, where Java was approximately half as fast. On Windows, Table 5, C-OpenSSL and Ruby were the fastest and have perform similar than on the Linux system. Python is faster than C#, C WinCrypto, and Java, but slower than when ran on Linux. C# and WinCrypto showed similar performance.

Part	Specification
RAM	4 GB
Processor	Intel® Core2™ Duo Processor E6400 (2M Cache, 2.13 GHz, 1066 MHz FSB)
Unix	Ubuntu 12.04.1 LTS, 32 bit
Win	Windows Server 2012 R2, 64-bit

Table 1: Test Environment

Language	Version	Module
Linux:		
Java	1.7.0_51 OpenJDK	java.security.MessageDigest
Ruby	2.1.0	openssl
Python	2.7.3	hashlib
C	GCC 4.6.3 -Ofast	OpenSSL 1.0.1
Windows:		
Java	1.7.0_51 SunJDK	java.security.MessageDigest
Ruby	2.0.0	openssl
Python	2.7.6	hashlib
C#	MS C# Compiler ver 12	System.Security.Cryptography
C	MSVC C/C++ Compiler ver 18	WinCrypto and openssl-1.0.1f

Table 2: Runtime Environment Windows

Language	a	b
Linux:		
C	0.209	2.752
Python	−0.575	2.758
Ruby	−0.488	2.758
Java	−1.642	5.558
Windows:		
C	−0.276	2.751
C (win)	0.523	3.736
C#	−0.367	3.669
Python	−0.422	3.374
Ruby	−0.206	2.647
Java	3.445	5.497

Table 4: MD5 a and b coefficients

Again, Java showed the slowest results; around 2 times slower as its counterparts (C-OpenSSL, Python and Ruby) and 1.5 times slower than C# and WinCrypto.

These findings also coincide with Table 4. Comparing b shows that C and Ruby have similar efficiency regardless of the operating system while Python is faster on the Linux system. As expected, Java is almost two times slower, evident by the value of $b = 5.558$ (Linux).

4.2 SHA-1

The detailed results for SHA-1 are shown in Table 6, 7 and 8 which shows that overall, SHA-1 is slower than MD5 with respect to all tested scenarios.

Again, OpenSSL (C, Python and Ruby) on Linux perform very well while we identified a slight drawback for Python. The Windows system shows a similar behavior – C has the fastest implementation followed by Ruby. Next are C (win), C# and Python with a small disadvantage for the latter one. Regardless the operating system, Java was almost three times slower than OpenSSL with a slope value of 9.303 and 8.345.

4.3 SHA-256

The detailed results for the SHA-256 are shown in Table 9, 10 and 11.

Data	Avg. elapsed time in milli-Sec.					
in MB	C#	C	C (win)	Java	Python	Ruby
2	7	6	8	11	6	6
4	16	9	16	24	14	9
8	29	22	31	45	27	21
16	58	44	61	86	54	42
32	117	88	120	177	108	86
64	234	177	239	356	215	169
128	470	352	478	709	431	339
256	938	705	956	1414	863	677
512	1877	1408	1922	2849	1727	1354
640	2346	1759	2381	3506	2159	1693
768	2817	2113	2863	4230	2590	2033
896	3285	2464	3358	4920	3023	2372
1024	3761	2817	3825	5628	3454	2711

Table 5: Average elapsed time for MD5 on a Windows system.

Data	Avg. elapsed time in milli-Sec.			
in MB	C	Java	Python	Ruby
2	6	19	6	6
4	12	38	12	12
8	24	75	24	24
16	49	149	48	48
32	97	297	98	96
64	193	595	195	192
128	387	1187	393	386
256	771	2380	785	771
512	1542	4748	1570	1548
640	1934	5974	1969	1933
768	2320	7134	2362	2320
896	2705	8308	2757	2700
1024	3093	9550	3149	3094

Table 6: Average elapsed time for SHA-1 on a Linux system.

Language	a	b
Linux:		
C	−0.572	3.019
Python	−0.357	3.023
Ruby	−1.217	3.076
Java	−1.353	9.303
Windows:		
C	0.342	3.017
C (win)	0.468	3.568
C#	0.167	3.576
Python	−0.346	3.774
Ruby	−0.781	3.346
Java	−0.668	8.345

Table 7: SHA-1 a and b coefficients

While our experiments showed constant results for the Linux system with OpenSSL outperforming Java, the tests on Windows vary. For SHA-256 Ruby was fastest on Windows followed by C and C (win). Again, Java remained the slowest. Compared to the previous tests, we uncovered an odd behavior of C# which performed well expect for 1 GB file. We hypothesize that C# had a larger memory footprint and 4 GB RAM was not sufficient when handling large data.

4.4 SHA-512

The detailed results for SHA-512 are shown in Table 12, 13 and 14.

The results for SHA-512 are similar to SHA-256. On the Linux system, Java is the slowest while all other results are almost identical. On Windows, Ruby was the fastest followed by Python. C, C#, C (win) and Java showed similar efficiency. However, on the 1 GB data file, C# again was slow, mostly due to what we hy-

Data	Avg. elapsed time in milli-Sec.					
in MB	C#	C	C (win)	Java	Python	Ruby
2	7	6	6	16	7	6
4	14	13	14	33	15	12
8	32	25	28	66	31	26
16	57	48	58	136	60	52
32	114	97	114	269	120	107
64	229	194	228	547	241	214
128	457	386	459	1062	482	427
256	916	773	913	2148	966	856
512	1831	1544	1836	4231	1931	1712
640	2288	1931	2280	5305	2417	2139
768	2745	2317	2738	6467	2897	2568
896	3204	2703	3205	7452	3382	3000
1024	3663	3091	3649	8567	3864	3424

Table 8: Average elapsed time for SHA-1 on a Windows system.

Data	Avg. elapsed time in milli-Sec.			
in MB	C	Java	Python	Ruby
2	18	29	17	17
4	36	59	36	35
8	73	117	72	71
16	144	236	142	143
32	291	470	287	287
64	577	942	573	575
128	1150	1897	1144	1146
256	2301	3769	2286	2336
512	4601	7545	4579	4568
640	5733	9395	5767	5730
768	6884	11417	6913	6875
896	8031	13226	7990	8034
1024	9163	15161	9135	9183

Table 9: Average elapsed time for SHA-256 on a Linux system.

Language	a	b
Linux:		
C	1.724	8.946
Python	3.486	8.956
Ruby	2.237	8.959
Java	−4.972	14.789
Windows:		
C	−5.176	9.037
C (win)	5.264	9.688
C#	−1044.546	17.53
Python	−0.503	10.189
Ruby	−0.180	8.038
Java	2.561	13.085

Table 10: SHA-256 *a* and *b* coefficients

pothesize is a memory footprint.

Overall, we note that on Windows, SHA-512 was faster than SHA-256 for all of the languages, especially for larger data sizes. On Linux, the speed for SHA-512 and SHA-256 were similar for all of the languages except for Java where SHA-512 was much slower than SHA-256.

4.5 Result summary

This section discusses and summarizes the main findings. A visual summary of all the experimental results is presented in Figures 1 and 2. While most graphs show an expected behavior, there are two striking results. On Linux, the Java implementation of SHA-512 shows an unexpected behavior while on Windows C# is particularly eye-catching.

More precisely, on the Linux system, programming languages using the OpenSSL showed similar high performance. Regarding

Data	Avg. elapsed time in milli-Sec.					
in MB	C#	C	C (win)	Java	Python	Ruby
2	19	17	19	27	20	17
4	38	36	37	53	40	33
8	78	74	78	103	81	64
16	157	142	156	209	163	129
32	313	286	309	422	326	256
64	631	572	622	847	651	514
128	1263	1145	1255	1683	1303	1029
256	2541	2291	2486	3359	2607	2057
512	5077	4650	4988	6695	5219	4115
640	6352	5720	6242	8361	6520	5143
768	8342	7005	7431	10028	7822	6174
896	9008	8009	8683	11793	9128	7201
1024	28859	9295	9903	13370	10433	8231

Table 11: Average elapsed time for SHA-256 on a Windows system.

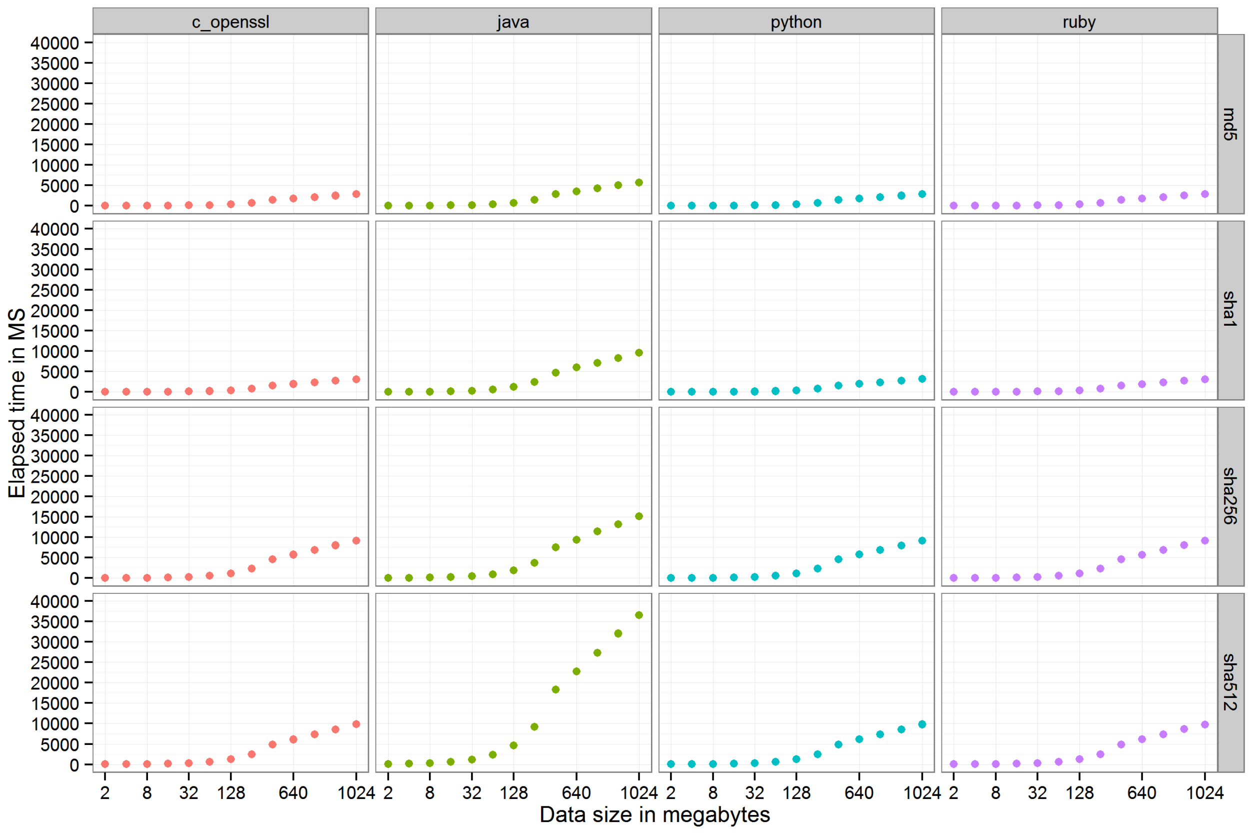

Figure 1: Overview of the measurements results for Linux.

Java, which was significantly slower than the OpenSSL library, we expected that for large data files that the efficiency will go up as Just-in-Time (JIT) compiler should have compiled and optimized byte code into native code. However, the slow performance of Java was related

Data	Avg. elapsed time in milli-Sec.					
in MB	C#	C	C (win)	Java	Python	Ruby
2	13	20	19	19	13	12
4	26	42	36	39	27	20
8	53	77	73	75	55	40
16	107	153	148	147	110	81
32	209	306	297	300	220	162
64	421	613	597	595	441	326
128	842	1231	1192	1180	883	651
256	1686	2452	2378	2356	1766	1302
512	3425	4894	4747	4706	3534	2606
640	4428	6114	5933	5895	4416	3256
768	5475	7342	7138	7073	5300	3908
896	5981	8567	8319	8238	6183	4557
1024	22381	9808	9492	9411	7067	5214

Table 14: Average elapsed time for SHA-512 on a Windows system.

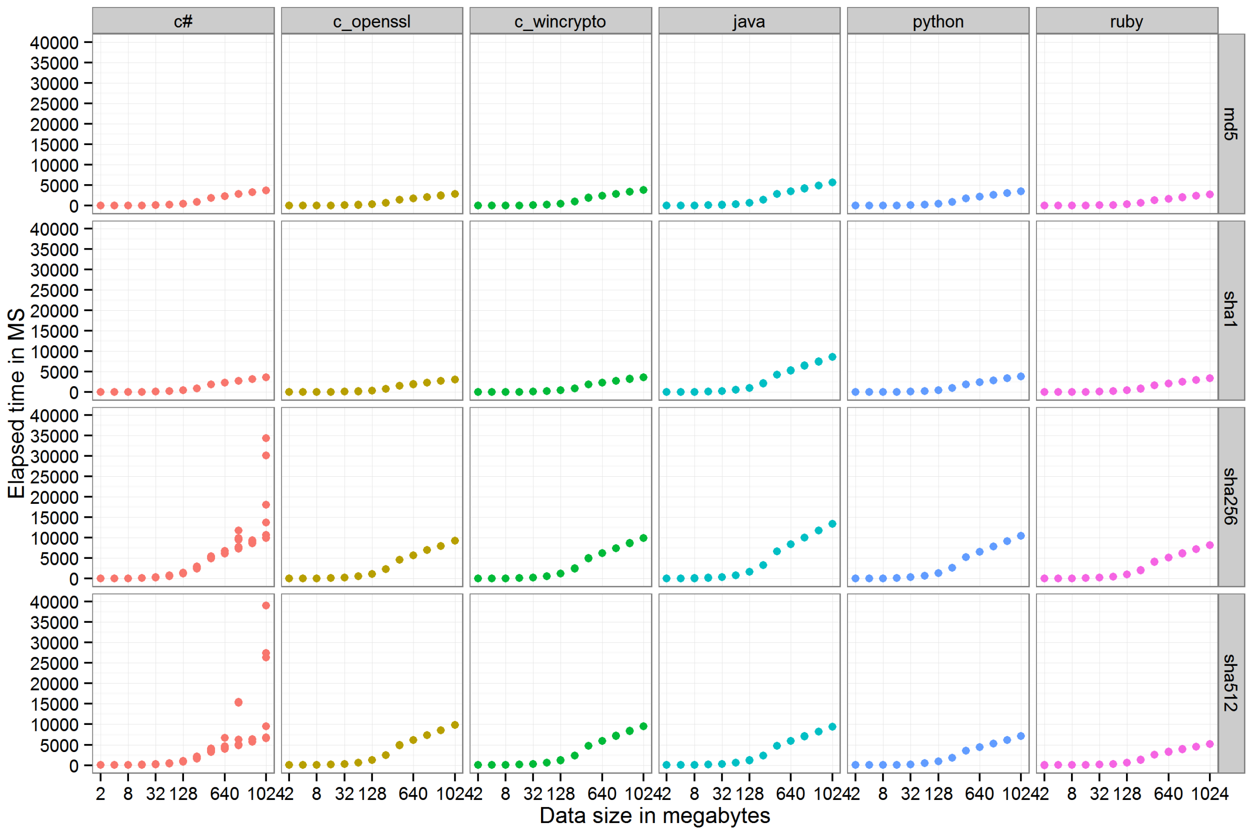

Figure 2: Overview of the measurements for Windows.

to the underlying cryptographic primitives, as noted by Garfinkel (Garfinkel et al., 2010). MD5 and SHA-1 were three times faster than SHA-256 and SHA-512.

On the Windows system, Java surprisingly was faster and outperformed C, Python and C#

Data	Avg. elapsed time in milli-Sec.			
in MB	C	Java	Python	Ruby
2	19	71	19	19
4	38	144	37	38
8	77	286	75	75
16	153	572	152	152
32	303	1144	304	303
64	611	2292	612	606
128	1221	4569	1224	1223
256	2431	9155	2448	2447
512	4870	18283	4864	4891
640	6079	22770	6123	6092
768	7335	27347	7335	7341
896	8565	31964	8572	8630
1024	9812	36527	9785	9714

Table 12: Average elapsed time for SHA-512 on a Linux system.

for SHA-512. We could not find any explanation why SHA-512 on Java has such high efficiency.

Again, programming languages using OpenSSL, such as Ruby and Python, steadily showed good and constant results on Windows. WinCrypto API showed good performance, and was better than OpenSSL for SHA-512. Overall Ruby showed the best times for SHA-256 and SHA-512
Main remarks:

- OpenSSL showed good performance across both platforms. This also applies to programming languages using OpenSSL as a library, such as Ruby and Python.

- SHA-256 and SHA-512 have a similar runtime. However, on Windows SHA-512 was faster while on Linux it was the other way round.

- On Windows Ruby was discovered to be faster than Python. On Linux the two languages were very similar.

- OracleJDK showed a higher performance on Windows than OpenJDK did on Linux. OracleJDK was specially good for SHA-512 on large data sizes.

Language	a	b
Linux:		
C	−0.009	9.547
Python	−1.803	9.557
Ruby	−5.256	9.559
Java	4.422	35.648
Windows:		
C	0.952	9.565
C (win)	0.906	9.277
C#	−874.141	13.348
Python	−0.548	6.902
Ruby	−0.102	5.089
Java	3.091	9.194

Table 13: SHA-512 *a* and *b* coefficients

- C# started showing sudden spikes on elapsed time for SHA-256 and SHA-512 when the data size reached 1 GB. This may be attributed to a lack of available RAM on the system used.

4.6 Impact on the real world

In this section we discuss the impact of our findings on a real world scenario. We assume that an investigator receives one hard drive of 512 GB, a smart phone having 32 GB memory, an SD-card of 8 GB and an external backup device of 160 GB. Furthermore, the user has 10 GB of cloud storage. We argue that this is a realistic scenario and that $(512 + 32 + 8 + 160 + 10 =)$ 722 GB can be easily found in a household nowadays especially when storing multimedia files such as videos and images.

Table 15 shows the upscaled results. For upscaling we used the times of processing 1024 MB = 1 GB, multiplied it by 722 and divided it by 1000 – except for star-marked numbers. Since, C# had problems with the 1024 MB file, we upscaled using the 512 MB file. Thus, the table shows the estimated time in seconds.

To conclude, there might be time differences of over 83 minutes for SHA-256 or even 322 minutes on Linux systems when using SHA-512.

5. CONCLUSION

Although most results were as expected, our experiments uncovered some strange behav-

Language	MD5	SHA-1	SHA-256	SHA-512
Linux:				
C	2036	2233	6615	7084
Python	2038	2273	6595	7064
Ruby	2039	2233	6630	7013
Java	4126	6895	10946	26372
Windows:				
C	2033	2231	6711	7081
C (win)	2761	2634	7149	6853
C#	2715	2644	7331*	4945*
Python	2493	2789	7532	5102
Ruby	1957	2472	5942	3764
Java	4063	6185	9653	6794

Table 15: Estimated time for processing 722 GB.

ior. The results on Linux are pretty solid and predictable – MD5 is the fastest while SHA-512 is the slowest both others are in between. Since most programming languages access the OpenSSL library, the times are quite constant. The slowest implementation was OpenJDK Java and therefore it is not recommended for hashing large amounts of data. We did not test OracleJDK on Linux.

Regarding Windows, the results are different and show unexpected behavior. The results for C (independent from the library) are reasonable and mainly coincide with the Linux results. C# showed strange behavior for SHA-256 and SHA-512 for larger files. We hypothesize that this is due to a larger memory footprint. Results for Python and Ruby are similar to the Linux results except for SHA-512 where algorithms are way faster on Windows. We cannot explain this behavior as of right now, and further experimentation is needed to explain these results.

In conclusion, for writing a tool that needs to be portable across Unix-like and Windows platforms, C-OpenSSL is a good choice, however scripting languages such as Ruby and Python showed strong promise for quick prototyping.

REFERENCES

Altheide, C., & Carvey, H. (2011). *Digital forensics with open source tools: Using open source platform tools for performing computer forensics on target systems: Windows, mac, linux, unix, etc* (Vol. 1). Syngress Media.

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

Breitinger, F., & Baier, H. (2012, October). Similarity Preserving Hashing: Eligible Properties and a new Algorithm MRSH-v2. *4th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*.

Breitinger, F., Stivaktakis, G., & Baier, H. (2013). Frash: A framework to test algorithms of similarity hashing. *Digital Investigation*, *10*, S50–S58.

frag_find. (2013). `https://github.com/simsong/frag_find`. ([Online; accessed Sep-2014])

Gallagher, P., & Director, A. (1995). *Secure Hash Standard (SHS)* (Tech. Rep.). National Institute of Standards and Technologies, Federal Information Processing Standards Publication 180-1.

Garfinkel, S., Nelson, A., White, D., & Roussev, V. (2010). Using purpose-built functions and block hashes to enable small block and sub-file forensics. *digital investigation*, *7*, S13–S23.

Kornblum, J. D. (2006, August). Identifying almost identical files using context triggered piecewise hashing. In *Proceedings of the digital forensic workshop* (p. 91-97). Retrieved from `http://dfrws.org/2006/proceedings/12-Kornblum.pdf`

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (2001). *Handbook of applied cryptography* (Vol. 5). CRC Press.

RDS Hashsets. (2014). `http://www.nsrl.nist.gov/`. ([Online; accessed Sep-2014])

Regional Computer Forensics Laboratory. Annual report. (2012). `http://www.rcfl.gov/downloads/documents/RCFL_Nat_Annual12.pdf`.

([Online; accessed Sep-2014])

Roussev, V. (2010a). Data fingerprinting with similarity digests. In *Advances in digital forensics vi* (pp. 207–226). Springer.

Roussev, V. (2010b). Data fingerprinting with similarity digests. In K.-P. Chow & S. Shenoi (Eds.), *Advances in digital forensics vi* (Vol. 337, pp. 207–226). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-15506-2_15` doi: 10.1007/978-3-642-15506-2_15

Saleem, S., Popov, O., & Dahman, R. (2011). Evaluation of security methods for ensuring the integrity of digital evidence. In *Innovations in information technology (iit), 2011 international conference on* (pp. 220–225).

spamsum. (2002-2009). `http://www.samba.org/ftp/unpacked/junkcode/spamsum/`. ([Online; accessed Sep-2014])

Sumathi, S., & Esakkirajan, S. (2007). *Fundamentals of relational database management systems* (Vol. 1). Springer Berlin Heidelberg.

Tridgell, A. (1999). *Efficient algorithms for sorting and synchronization.* Australian National University Canberra.

Wang, X., Yin, Y. L., & Yu, H. (2005). Finding collisions in the full sha-1. In *Advances in cryptology–crypto 2005* (pp. 17–36).

Wang, X., & Yu, H. (2005). How to break md5 and other hash functions. In *Advances in cryptology–eurocrypt 2005* (pp. 19–35). Springer.