All your hashes are belong to us: An overview of malware hashing algorithms

gdatasoftware.com/blog/2021/09/an-overview-of-malware-hashing-algorithms

VirusTotal's "Basic Properties" tab alone lists eight different hashes and supports even more to use them for queries and hunt signatures. Hashes are important for malware analysis, as well as identification, description and detection. But why do so many of them exist and when should you use which hash function?

Cryptographic hashes: MD5, SHA-1, SHA-256

Cryptographic hashing algorithms are a mathematical function that produces an alphanumeric string that is unique for a specific file or data input, making it an unalterable identifier. Unlike encryption, cryptographic hashing is a one-way function and only works in one direction. It is designed to be practically infeasible to compute the original input based on the hash value alone. Even changing a single byte in the input will result in a different hash value. That way an adversary cannot see if their input sample is anywhere close to producing the desired hash value.

All of these hashes have a fixed length. For the standard implementation of MD5 it is 128 bits (16 bytes), for SHA-1 160 bits (20 bytes) and for SHA256 the length is in the name: 256 bits (32 bytes).

The main purposes of these hashes are identification and blocklisting of samples. Using them for blocklisting makes sense because an attacker will have difficulty to design a malware with the same hash value as a clean file. They are ideal for identification because cryptographically secure hashes are meant to make collisions unlikely.

MD5 and SHA-1 should not be used anymore because they have been broken [fisher20] [kashyap06]. E.g. for MD5 people can create hash collisions in a way that allows control over the content [kashyap06]. But both are still sometimes used in hash listings of malware articles and some detection technologies might still work with MD5 hashes because computing them is fast and the values don't need much storage space. Therefore it is an important and common search option for sample databases.

Fuzzy Hashes: dcfldd, ssdeep, TLSH, mvHash-B

Fuzzy hashes are also called Similarity Preserving Hash Functions (SPHF). Unlike cryptographic hashes their goal is to provide a comparison or similarity measure. Fuzzy hash functions are further categorized into four types [p.1, martinez14]:

- Block-Based Hashing (BHB), e.g., the program dcfldd by Harbour creates hash values via BHB
- Context-Triggered Piecewise Hashing (CTPH), of which the most popular example is ssdeep
- Statistically-Improbable Features (SIF), e.g., sdhash
- Block-Based Rebuilding (BBR). e.g., mvHash-B

BHB creates a hash for every fixed-sized block of the input data. The larger the input data, the longer the resulting hash value will be. A similarity is determined by counting all blocks with the same hash value. BHB is used in forensics (dcfldd is a forensics tool) but not so much for sample analysis. Maybe because the arbitrary and potentially large size of the hash value makes it impractical for signatures and storage.

CTPH uses trigger points instead of fixed-sized blocks. Everytime a specific trigger point hits, the algorithms calculates a hash value of the current chunk of data. The conditions for the trigger points are chosen in such a way that the final hash value doesn't grow arbitrarily in size with increased input data size. E.g., ssdeep has a desired number of 64 chunks per input file, so the trigger point is dependent on the size of input data. To compare two files, ssdeep uses an edit distance algorithm: The more steps it takes to transform one ssdeep hash value to the other, the less similar the files are.

The development of ssdeep was a milestone at the time. New hashing algorithms which improve certain aspects of ssdeep have been created since. E.g., SimFD has a better false positive rate and MRSH improved security aspects of ssdeep [breitinger13]. The author's

website states that ssdeep is still often preferred due to its speed (e.g., compared to TLSH) and it is the "de facto standard" for fuzzy hashing algorithms used for malware samples and their classification. Sample databases like VirusTotal and Malwarebazaar support it.

TLSH stands for Trend-Micro Locality Sensitive Hash, which was published in a paper in 2013 [oliver13]. According to their paper TLSH has better accuracy than ssdeep when classifying malware samples [p.12, oliver13]. Just like ssdeep it is a CTPH. TLSH is supported by VirusTotal.

The idea of **SIF** hashing is to find features of a file that are unlikely present by chance and compare those features to other files. Sdhash uses entropy calculation to pick the relevant features and then creates the hash value based on them. That also means sdhash cannot fully cover a file and modifications to a file may not influence the hash value at all if they are not part of a statistically-improbable feature. Sdhash shows better accuracy than ssdeep when classifying malware samples [p.12, oliver13][roussev11]. However, its strong suit is the detection of fragments and not comparison of files [p.8, breitinger12].

BBR uses auxiliary data to rebuild a file. mvHash-B for instance maps every byte of the input file to either 0xFF or 0x00 by comparing it to its neighbors via a majority voting. If most of the neighbors are 1, the byte becomes 0xFF, otherwise 0x00. Afterwards the byte sequences are compressed to form a hash value. Other examples are the algorithms discussed in the section Image similarity: aHash, pHash, dHash

Control Flow Graph hashing: Machoc and Machoke

Machoc creates a numeric representation of a sample's control flow graph (CFG). Suppose you have a CFG like in the image on the right. The numbered blocks are turned into the following string (example from Github page):

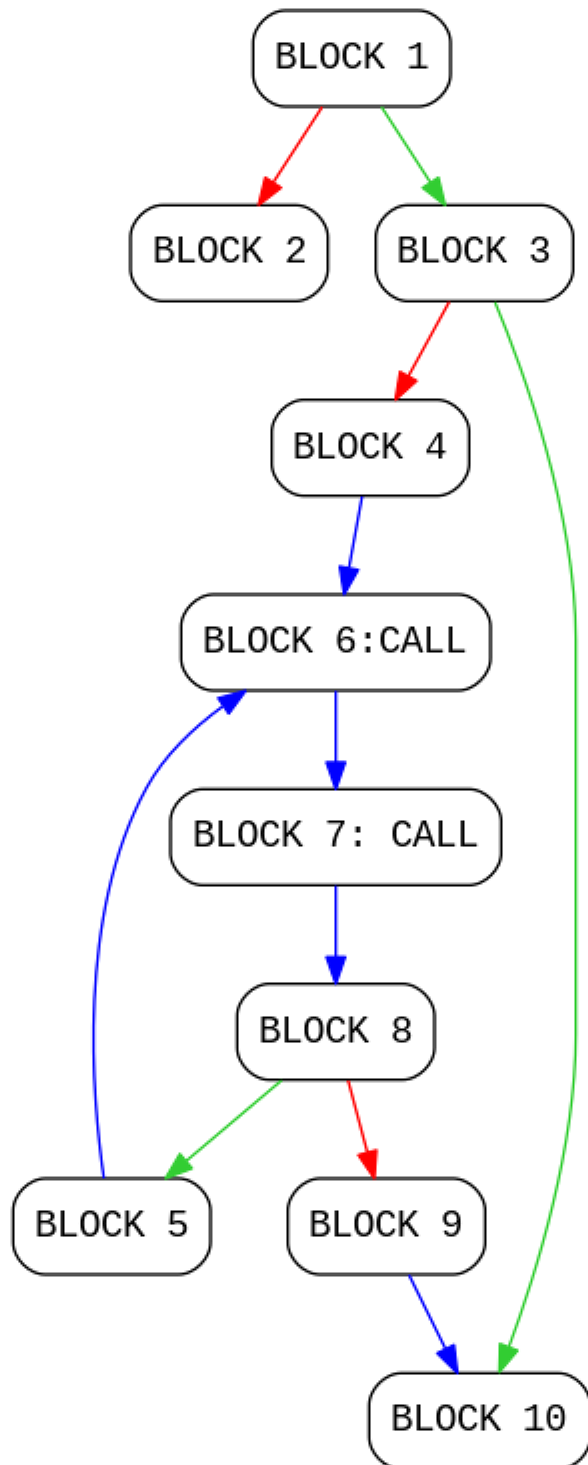
```
1:2,3;
2:;
3:4,10;
4:6;
5:6;
6:c,7;
7:c,8;
8:5,9;
9:10;
10:;
```

Machoc then applies Murmuhash3 to this string to create the final hash value. Therefore, samples with the same control flow graph will have the same hash value.

Machoke is the same algorithm, but a different implementation. Machoc bases their control flow graphs on IDAPython or miasm, whereas Machoke uses radare2 and r2pipe.

These hashing algorithms are limited to the executable types supported by their disassemblers and are vulnerable to control flow obfuscation.

Control flow graph hashes are not only useful for AV detection and sample clustering. They are also suitable to get a binary diff for samples, i.e., to identify similar and different functions in two samples. Binary diffing is a common technique for malware analysis to find differences between two versions of a malware family or identify re-used code in different malware families. Control flow hashing may also be applied to automatically rename known functions, thus, improve the readability of disassembled code for reversers.



Machoc applies block numbering to a control flow graph; image from https://github.com/ANSSI-FR/polichombr/blob/dev/docs/screenshots/cfg_numbered.png

Import hashing: ImpHash, TypeRefHash and ImpFuzzy

All of these hashing algorithms work with imported functions, types or modules. The idea is that the imports indicate behavioral capabilities of a malware, so a hash value will hopefully be the same for samples with similar capabilities.

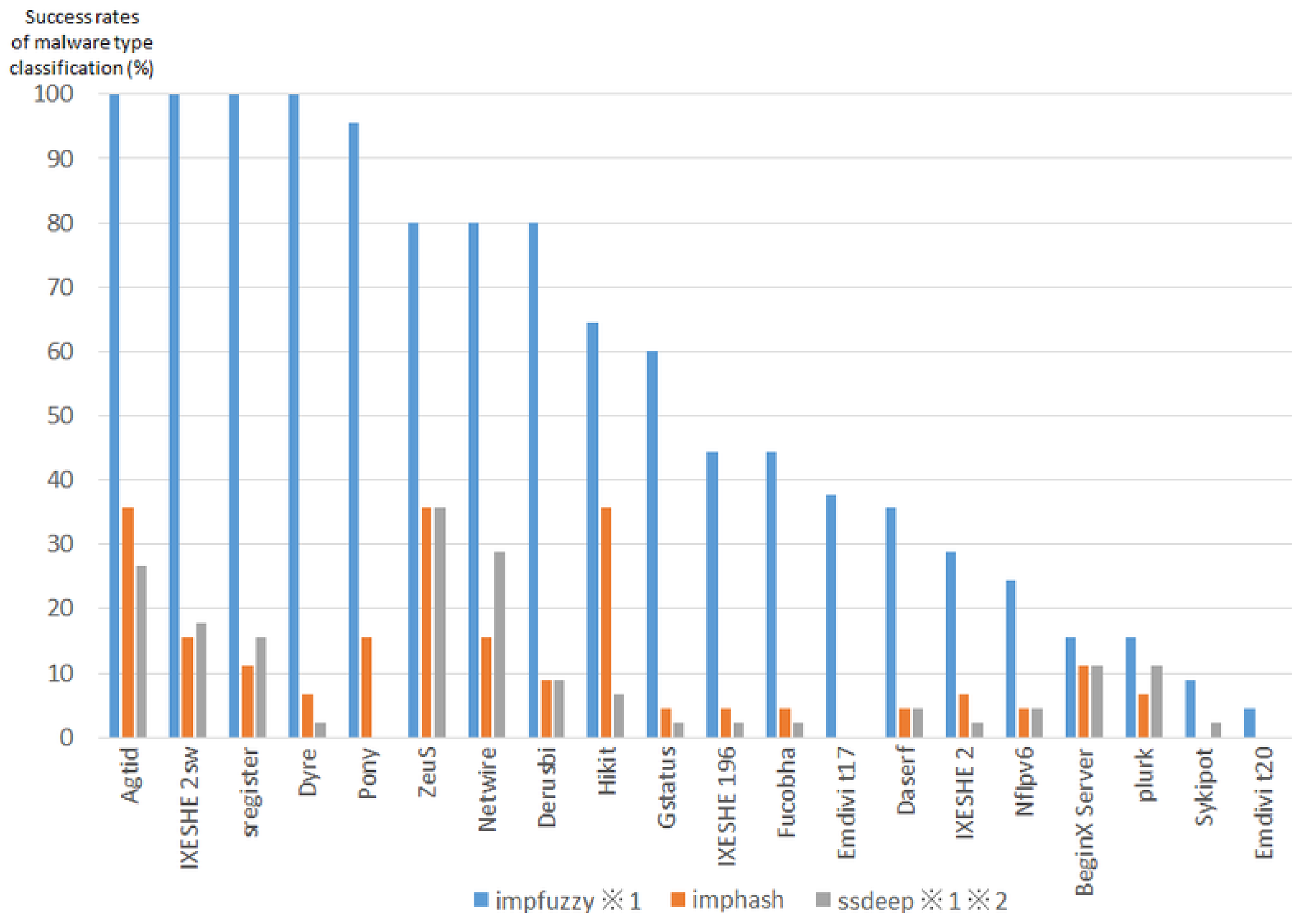
The ImpHash is used specifically for Portable Executable (PE) files and based on the PE import table contents. It concatenates the imported function names and module names, puts them to lowercase, then creates the MD5 value of the resulting string. That MD5 value is the ImpHash.

Explanation video of Imphash history and algorithm

Windows samples that are based on the .NET framework are also PE files. However, usually they only have one PE import, which is the .NET runtime. So the ImpHash is rather useless for .NET based PE files. Everything that is imported by the user-defined .NET code resides in .NET specific metadata tables. The TypeRef table is the .NET counterpart to the PE import table. It contains namespaces and types used by the sample. The TypeRefHash algorithm orders and concatenates TypeNamespaces and TypeNames, then creates the SHA-256 hash value of the resulting string.

The choice of using a cryptographic hash as intermediate step for import hashing is not ideal when keeping in mind that the idea behind ImpHash was to cluster samples of similar behavioral capabilities. Algorithms like ImpHash and TypeRefHash only determine clusters of samples that have exactly the same imports. Fuzzy hash values look similar if the input was similar. That is is why algorithms like ImpFuzzy were created, which uses ssdeep instead of MD5. A recent study [naik20] shows better results in malware classification tasks for fuzzy import hashing methods that employ ssdeep, sdHash or mvHash-B compared to MD5 for the ImpHash.

The ImpFuzzy blog post evaluates malware family classification for 200 non-packed samples using either ssdeep for the whole file, ImpHash (MD5 on imports) or ImpFuzzy (ssdeep on imports). For this specific test setup, ImpFuzzy shows consistently better success rates than the other two hashing algorithms (see image below) but the author also states that this setting creates false positives.



Malware classification success rates of ImpFuzzy vs ImpHash vs ssdeep, image from https://blogs.jpcert.or.jp/en/2016/05/classifying-mal-a988.html

Human readable hash: Humanhash

The humanhash has one purpose: It should be rememberable and pronouncable by humans, so we can search for these hash values in databases. Example hashes are "happy-edward-three-xray", "johnny-triple-william-jig" or "virginia-quebec-march-london".

The hash value is created by converting the input size to 4 bytes, then mapping each byte to a wordlist. The author states its uniqueness is 1 in 4.3 billion. This hash is not robust against collisions, but it does not have to be.

The original author's Github page states humanhash was inspired by Chroma-Hash, which is a colorful representation of hashes, and the NATO phonetic alphabet.

In my personal opinion more sample sharing platforms should add humanhash to their list of hashes. E.g., it would be a great addition to VirusTotal. Malwarebazaar supports humanhash and seeing it among the other hash values (image below) makes apparent what this sample will be remembered by apart from the filename and AgentTesla tag.

SHA256 hash:	ee7c081fae091ac85f5183f07128c756f8d5cadd515279c3f261c67291b8d8aa
SHA3-384 hash:	a2009fafe22afa9c9f72bf418a8eafc5b64b64814ec621dc4ad11ae8fdb514355140fe55ec4797058d5f65dbecb745ac
SHA1 hash:	4d89dc84c1dd90d67963f7f4646f5748236206f6
MD5 hash:	62405d3ff4066f24f1d653fe593a9e77
humanhash:	virginia-quebec-march-london
File name:	DUBAI 34243D.exe
Download:	download sample
Signature	AgentTesla Alert
File size:	861'184 bytes
First seen:	2021-09-09 07:24:43 UTC
Last seen:	*Never*
File type:	exe
MIME type:	application/x-dosexec
imphash	f34d5f2d4577ed6d9ceec516c1f5a744 (21'418 x AgentTesla, 4'952 x Formbook, 2'761 x Loki)
ssdeep	12288:dLMO3+VUPObK1Cnf2VtYLrlz1+e+75jJ9Oh6ZjHTSY90VXrAUNDcB933Dx9LX1:FHwlclmkBZ0VUG8x9tX
TLSH	T1EC05192479EA401DFD73BF751EF438969A6FBE733A36A41D148023860A33B81DD9153A
dhash icon	0307612c88000000 (18 x AgentTesla)
Reporter	@GovCERT_CH
Tags:	AgentTesla exe

Hash listing on Malwarebazaar for the virginia-quebec-march-london sample

Image similarity: aHash, pHash, dHash

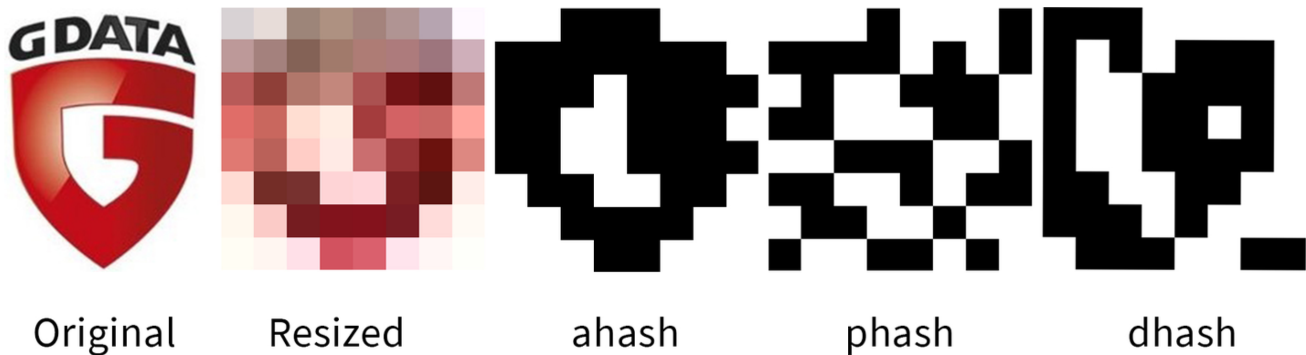
Checking icon similarity is especially useful if malware pretends to be a known application or office document. E.g., it is common for malware to try to appear as Word or PDF document by using icons for these applications in combination with double extensions like pdf.exe or file extension spoofing. Detecting such malware techniques with signatures or searching for them in databases is possible with similarity hashes that are specifically for comparing pictures, e.g., VirusTotal and Malwarebazaar support searches via dHash.

There are many hashing algorithms for image comparison, e.g., this blog article compares six of them. But the ones mentioned the most are pHash, aHash and dHash.

All three hashing algorithms first resize the picture to a fixed size and then convert it into grayscale. At this point aHash aka average hash compares every pixel value to the average grayscale pixel value of the image. If it is greater (= brighter) than the average, it sets the pixel to 1, otherwise to 0. pHash aka perceptual hash applies a Discrete Cosine Transform and compares pixels based on frequencies. dHash aka difference hash compares every pixel to their right neighbor (except the last one in each row). If the pixel value is increasing, it is set to 1; otherwise it is set to 0.

The consensus according to several articles [animeloop17][hackerfactor13] seems to be that dHash is the fastest of the three algorithms and also accurate, but it does not detect similarity in cropped images. pHash has the best accuracy but also the worst performance. aHash seems to be the least accurate of the three algorithms.

Image hashing algorithms are also used in sample clustering and applied to an image visualization of the malware file itself [bhaskara18].



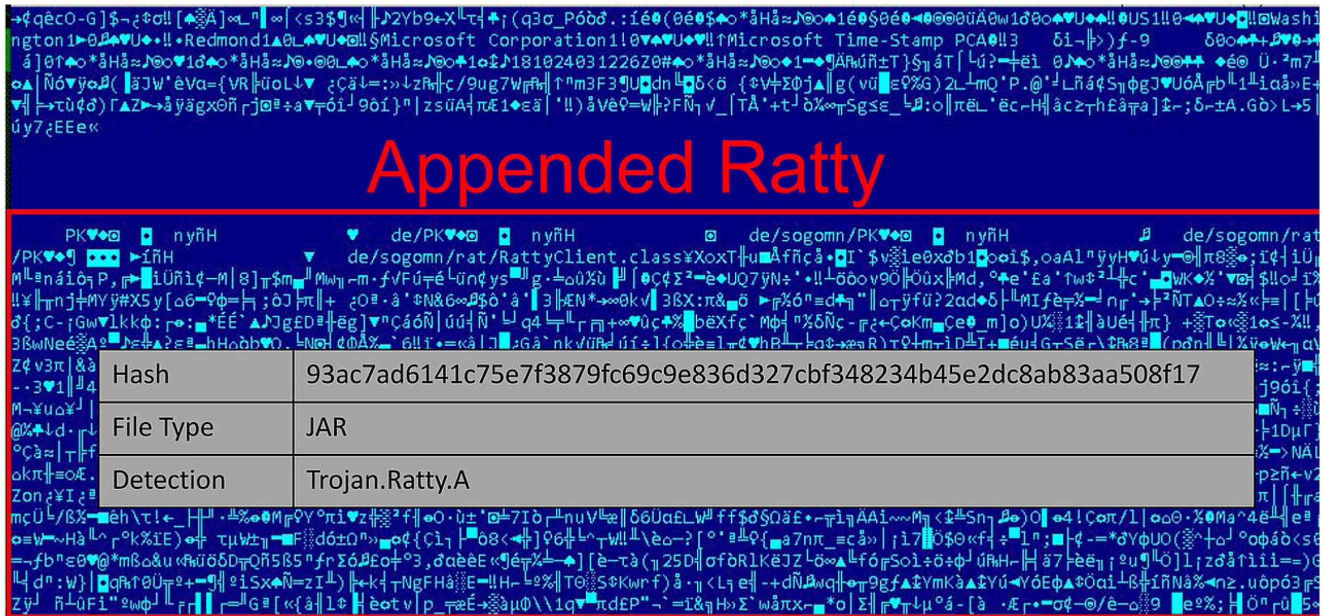
Result of different image hashing algorithms

Digital certificates: Authentihash

This cryptographic hash is computed on signed PE files and an important part of Microsoft's digital signature format Authenticode. Its purpose is to verify that a file has not been tampered with after it has been signed by a software publisher. File manipulation would result in a different hash value than the one listed in the file's digital signature. The Authentihash includes the PE image excluding certificate related data and overlay. That means appended data does not affect the hash value which has been abused by polyglot malware, that is malware that has several file types at once. More details about such malware is in the article "Code-Signing: How Malware gets a Free Pass"

Additionally appended data is sometimes also used to store settings of a file. E.g., as seen in MuddyWater campaign samples, the ScreenConnect clients have a valid certificate, but potentially risky settings are in the overlay and do not affect the signature.

For malware analysts Authentihash is useful to verify digital signers and to find such polyglot malware files or similar singed files with different appended data, e.g., with VirusTotal queries.



Appended malicious Java archive to a signed MSI file, resulting in a validly signed malware executable. Image from https://www.gdatasoftware.com/blog/how-malware-gets-a-free-pass

Rich PE Header hashes: Rich, RichPV

The Rich Header is part of Portable Exectuable files since Visual Studio 97 SP3. According to a study conducted in 2019 [p. 5, poslusny19], the Rich Header exists in 73.20 percent of all native PE files and represents a fingerprint of the development environment. That means it is useful for attribution, sample hunting, clustering, and as part of detection signatures.

The Rich Header hash or short **Rich** is calculated the following way [p. 8, dubyk19]: Part of the Rich Header is XOR encrypted. The Rich Header algorithm first searches for the decryption key, then decrypts the rich header data between the magic values "DanS", indicating the beginning of the plain text header, and "Rich", indicating the end of the plain text header. Finally the MD5 hash function is applied on the decrypted area. The resulting hash value is the Rich Header hash of the sample. The XOR decryption makes sure that the same Rich Header data contents yield the same hash value if the XOR key changes.

One modification of Rich is called **RichPV** and excludes the most volatile Rich Header field from the MD5 input data, the so called **pC** or **Product Count** field [p. 8, dubyk19]. "pC measures the number of source files referenced by the PE. As a result, the pC field has the potential to change across different PEs as the number of source files increase and decrease even if the products and their versions remain constant" [p. 8, dubyk19]. So generally, if we want to find samples that were compiled on the same system, from the same source code project, RichPV hash should be more suitable than Rich hash.

VirusTotal displays the Rich hash in the Details tab. RichPV might be a useful addition to that.

References

[animeloop17] "Animeloop: animation loop recognition", September 2017 https://blog.windisco.com/animeloop-paper-en/

[bhaskara18] Vineeth S. Bhaskara and Debanjan Bhattacharyya, "Emulating malware authors for proactive protectionusing GANs over a distributed image visualization ofdynamic file behavior", July 2018, https://arxiv.org/pdf/1807.07525.pdf

[breitinger12] F. Breitinger and H. Baier, "Properties of a similarity preserving hash function and their realization in sdhash," *2012 Information Security for South Africa*, 2012, pp. 1-8, doi: 10.1109/ISSA.2012.6320445.

[breitinger13] Breitinger F., Baier H. (2013) "Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2". In: Rogers M., Seigfried-Spellar K.C. (eds) Digital Forensics and Cyber Crime. ICDF2C 2012. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 114. Springer, Berlin, Heidelberg. doi.org/10.1007/978-3-642-39891-9_11

[dubyk19] Dubyk, Maksim. "Leveraging the PE Rich Header for Static Malware Detection and Linking." (2019). https://www.giac.org/paper/grem/6321/leveraging-pe-rich-header-static-alware-etection-linking/169729

[fisher20] Dennis Fisher, "SHA-1 ‘Fully and Practically Broken’ By New Collision", January 2020 https://duo.com/decipher/sha-1-fully-and-practically-broken-by-new-collision

[hackerfactor13] Neal Krawetz, "Kind of Like That", January 2013, http://www.hackerfactor.com/blog/?/archives/529-Kind-of-Like-That.html

[kashyap06] Kashyap N. A "Meaningful MD5 Hash Collision Attack" [Internet]. 2006. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.2659&rep=rep1&type=pdf

[kim20] Kim, Jun-Seob & Jung, Wookhyun & Kim, Sangwon & Lee, Shinho & Kim, Eui. (2020). Evaluation of Image Similarity Algorithms for Malware Fake-Icon Detection. 1638-1640. 10.1109/ICTC49870.2020.9289501.

[martinez14] Martínez, V., F. Álvarez and L. H. Encinas. “State of the Art in Similarity Preserving Hashing Functions.”, 2014, https://digital.csic.es/bitstream/10261/135120/1/Similarity_preserving_Hashing_functions.pdf

[naik20] N. Naik, P. Jenkins, N. Savage, L. Yang, T. Boongoen and N. Iam-On, "Fuzzy-Import Hashing: A Malware Analysis Approach," 2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), 2020, pp. 1-8, doi: 10.1109/FUZZ48607.2020.9177636.

[oliver13] Oliver, J., Cheng, C., Chen, Y.: "TLSH - A Locality Sensitive Hash. 4th Cybercrime and Trustworthy Computing Workshop", Sydney, November 2013
https://github.com/trendmicro/tlsh/blob/master/TLSH_CTC_final.pdf

[poslusny19] Poslušný, Michal & Kálnai, Peter. (2019). "Rich Headers: leveraging this mysterious artifact of the PE format for threat hunting."
https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Kalnai-Poslusny.pdf

[roussev11] Vassil Roussev, "An evaluation of forensic similarity hashes," Digital Investigation, vol. 8, Supplement, no. 0, pp. 34 – 41, 2011.