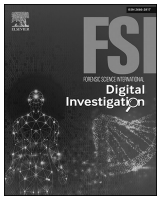
Forensic Science International: Digital Investigation 53 (2025) 301927

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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



DFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA

Bytewise approximate matching: Evaluating common scenarios for executable files

Carlo Jakobs *, Axel Mahr, Martin Lambertz, Mariia Rybalka, Daniel Plohmann

Fraunhofer FKIE, Zanderstr. 5, 53177, Bonn, Germany

A R T I C L E I N F O

Keywords:
Bytewise approximate matching
Executable files
Similarity hashing
Fuzzy hashing
Ssdeep
Sdhash
TLSH
MRSHv2

A B S T R A C T

This research explores the application of bytewise approximate matching algorithms on executable files, evaluating the effectiveness of ssdeep, sdhash, TLSH, and MRSHv2 across various scenarios, where approximate matching seems to be a natural tool to employ. Previous works already underlined that approximate matching is often used for tasks where the algorithms have not been thoroughly and systematically evaluated. Pagani et al. (2018), in particular, highlighted the shortcomings of previous research and tried to improve current knowledge about the applicability of approximate matching in the context of executable files by evaluating typical use cases. We extend their work by taking a closer look at further common scenarios that are not covered in their article. Specifically, we examine use cases such as different versions of the same software and comparisons between on-disk and in-memory representations of the same program, both for malicious and benign software.

Our findings reveal that the considered algorithms' performance across all evaluated scenarios was generally unsatisfactory. Notably, they struggle with size-related and localized modifications introduced during the loading stage. Furthermore, executables with no functional similarity may be mismatched due to shared byte-level similarity caused by embedded resources or inherent to certain programming languages or runtime environments. Consequently, these algorithms should be used cautiously and regarded as assisting tools rather than reliable methods for indicating similarity between executable files, as both false positives and false negatives can occur, and users should be aware of them.

Moreover, while some of the unfavored results stem from design decisions, we observed unexpected behavior in some experiments that we could trace back to issues in the reference implementations of the algorithms. After fixing the implementations, the strange effects in our results indeed disappeared. It is still an open question if and to what extent previous experiments and evaluations were affected by these issues.

1. Introduction

The ever-growing volume of digital data presents significant challenges in digital forensics, where efficiently analyzing vast amounts of information is crucial. Hence, the need for effective filtering, prioritization, and classification techniques becomes increasingly critical. A common approach to this challenge is using cryptographic hashing, which provides a simple and efficient solution for identifying and filtering exact matches within large data sets. However, cryptographic hashes fall short when identifying similar objects, as even a minor alteration in a file will produce a completely different hash. This limitation underlines the need for more sophisticated methods, such as approximate matching techniques, which can identify related files despite variations, highlighting files similar to known interesting files, or filtering out files similar to known uninteresting files, for instance.

Similarity can be measured at different abstraction layers. The lowest layer only considers the byte sequence that makes up a file, while the upper layers incorporate its syntactical structure or semantic content. Detecting similarity at the upper layers requires knowledge about the file type being processed and is typically more computationally expensive. Bytewise approximate matching, on the other hand, is agnostic to the file type and generally faster (Breitinger et al., 2014a), making it more suitable as a filtering technique than approaches operating at higher abstraction layers.

However, prior research has shown inconsistent results when assessing the applicability of bytewise approximate matching to

* Corresponding author.
E-mail addresses: carlo.jakobs@fkie.fraunhofer.de (C. Jakobs), axel.mahr@fkie.fraunhofer.de (A. Mahr), martin.lambertz@fkie.fraunhofer.de (M. Lambertz), mariia.rybalka@fkie.fraunhofer.de (M. Rybalka), daniel.plohmann@fkie.fraunhofer.de (D. Plohmann).

https://doi.org/10.1016/j.fsidi.2025.301927

compute the similarity between binary programs. Specifically, Pagani et al. (2018) provided a vital discussion of this issue, highlighting that approximate matching has often been used without fully understanding the implications of the selected algorithms. Furthermore, the authors took the first step in solving this issue by conducting three case studies representing typical use cases for approximate matching in binary program analysis.

In our paper, we build on the work of Pagani et al., complementing their results. Although we also investigate approximate matching in the context of executable files, we do not focus on binary analysis. Instead, we consider tasks more common in digital forensic investigations or triage scenarios. In particular, we look at three use cases where detecting similarity to known programs is particularly useful.

Detecting Updates: Detecting updated or otherwise modified versions of known software, whether malicious or benign, is a canonical example of a use case for approximate matching. Identifying updated versions of known legitimate software enables safelisting techniques to assign lower priority to the corresponding files during an analysis. For malware, computing the similarity between samples can aid in identifying malware families and variants, understanding their evolution, and detecting new threats based on known samples. Additionally, it can support threat intelligence by uncovering relationships between different malware strains.

Highlighting and Safelisting: Databases of known programs or operating system files, such as the National Software Reference Library (NSRL) managed by NIST (White, 2005), can be used to filter out benign files. Platforms like VirusTotal, on the other hand, can be used to identify malicious files. Both also often provide similarity digests to allow approximate matching. However, for reliable filtering, the two classes must be sufficiently discernible, which is an aspect that has not been scrutinized well enough.

Memory Forensics: Similarities between on-disk programs and their in-memory representation are relevant for memory forensics. An analyst may triage a list of acquired processes, filtering out well-known benign ones while directing their attention to more suspicious or high-priority processes. Likewise, they can connect information from an executable on disk to the related processes, providing further insights.

Our paper evaluates four prominent bytewise approximate matching algorithms in the above scenarios. In particular, we present the following evaluations:

- operating system files vs. malware
- all-against-all and intra-family comparison of malware samples
- different releases of benign third-party software and operating system file versions
- memory-mapped files vs. their on-disk counterparts

During our experiments, we discovered notable issues in the algorithms' reference implementations, one of which significantly impacted almost all our experiments. We solved the issues and provided the authors with corresponding information and patches. In this paper, we illustrate the impacts we observed in our evaluations.

Moreover, we illustrate why one of the algorithms computes unexpectedly high scores in specific scenarios.

In summary, our paper contributes by.

- evaluating four bytewise approximate matching algorithms in common forensics use cases;
- showing that the algorithms are unsuitable for most considered use cases and perform too unreliably;
- describing issues in the reference implementations and illustrating their impact; and
- providing background knowledge to better interpret the similarity scores of the algorithms.

We provide supplemental material in a GitHub repository[1], where we present further figures, lists of files in our corpora with their corresponding similarity digests, and evaluation scripts.

2. Scope

Much research has been done on bytewise approximate matching algorithms. This research covered various aspects in this area, ranging from studies of the performance and properties of the algorithms in general or for specific use cases to the development of new algorithms. Moreover, many works incorporate bytewise approximate matching algorithms as building blocks for their purposes.

This paper focuses on applying four of the most prominent bytewise approximate matching algorithms in the context of executable binary files. We are fully aware that the algorithms were not explicitly designed for executable files. Moreover, previous work has already pointed out weaknesses of the algorithms in this area (Li et al., 2015; Coffman et al., 2018; Pagani et al., 2018). Nevertheless, they are still widely employed in recent research and practical solutions (Ali et al., 2020; Bak et al., 2020; Nguyen et al., 2022; Namanya et al., 2020; Naik et al., 2021; Botacin et al., 2021; Kida and Olukoya, 2023; Magonia Research, 2023; Hutelmyer and Borre, 2024). Hence, we have the impression that the properties of using bytewise approximate matching in this context and the consequential implications are not understood comprehensively.

In Section 3, we present previous efforts to remedy these shortcomings by systematically evaluating bytewise approximate matching algorithms when applied to executable files. Our paper lines up with the evaluations presented in these works by investigating different use cases or assessing the currentness of their findings. We aim to evaluate how well bytewise approximate matching algorithms perform in our scenarios, contributing to a more comprehensive understanding of the algorithms. Given this focus, we explicitly exclude algorithms that are tailored to specific file types or use cases, such as mrsh-mem (Liebler and Breitinger, 2018), apx-bin (Liebler and Baier, 2019), Telfhash (Mercês, 2020) or approaches such as the ones presented by Li et al. (2015), Naik et al. (2021), or Fleming and Olukoya (2024). While they might yield better results in particular use cases, they commonly lack the more universal applicability bytewise approximate matching provides.

3. Related work

There has been considerable effort to evaluate the effectiveness of approximate matching (Roussev, 2011; Breitinger et al., 2014b; Oliver et al., 2014; Harichandran et al., 2016), which, to some extent, touches on the evaluations of our work. However, the experiments are worth systematic expansion and actualization. Moreover, the scope was broader and did not focus on binary executable files.

Regarding executable files, Pagani et al. (2018) conducted a detailed analysis of similarity hashing techniques, focusing on several practical scenarios: the identification of libraries in statically linked programs, a typical binary reverse engineering task; the comparison of applications compiled with different toolchains and compiler options, relevant to embedded systems and firmware analysis; and lastly, an examination of different versions of the same application, which involves tracking the evolution of malware or identifying related samples. Their research evaluated the performance of ssdeep, sdhash, and TLSH within these contexts to determine their effectiveness.

The study concluded that ssdeep performs poorly in most tasks, while sdhash and TLSH were more effective. Specifically, sdhash excelled at

[1] https://github.com/fkie-cad/paper-material-bytewise-approximate-matching-scenarios-for-binaries.

recognizing the same program compiled differently and TLSH reliably identified software variants with source code changes. The authors emphasized the importance of understanding the underlying mechanisms of these algorithms rather than applying them mindlessly.

Several more publications assessed the robustness of bytewise approximate matching algorithms concerning common modifications to executable files. Liebler and Baier (2019) evaluated the same scenarios as Pagani et al. (2018) including their framework apx-bin. Coffman et al. (2018) inspected the influence of different compilers and obfuscation and optimization techniques on the similarity scores of various algorithms. Oliver et al. (2014) assessed the effects of introducing source code changes such as reordering operands, functions, and statements, adding binary data, and inserting new and changing present variables.

Martín-Pérez et al. (2021) evaluated dcfldd, ssdeep, sdhash, and TLSH when applied to memory-mapped files for operating system files and benign programs. Their work highlights the implications of relocation in Windows processes, significantly affecting similarity scores.

Our work complements Pagani et al. (2018) by evaluating different scenarios for executable programs. Some of our experiments have already been conducted in related works. However, we extend them by including more algorithms and operating systems or using a larger data set, bridging the gap to a more systematized and thorough understanding of bytewise approximate matching for executable binary files.

4. Background

This section provides a brief overview of the bytewise approximate matching algorithms examined in our work. We selected four of the most prominent and widely used candidates in the field of bytewise similarity, which at the same time implement different algorithmic approaches.

ssdeep generates two signatures of at most 32 to 64 Base64 characters, with each character representing a block of data from the input file (Kornblum, 2006). The file is divided into segments based on a sliding 7-byte window, where a specific condition triggers new block boundaries, a method known as context-triggered piecewise hashing (CTPH). The algorithm's main weakness is the short hash length, which can lead to inaccuracies, especially when the segmentation process produces poorly sized segments for certain file types (Jakobs et al., 2022).

MRSHv2, another CTPH algorithm, generates variable-length hash values based on a specified average block size, typically 320 bytes (Breitinger and Baier, 2013). Smaller inputs yield shorter hashes, while larger inputs produce longer hashes. The algorithm uses Bloom filters to compress the data into a more manageable hash representation. The comparison process involves matching Bloom filters between hashes, where the best match is identified for each Bloom filter in the smaller hash. The similarity score is calculated as the sum of these best matches divided by the number of filters in the larger hash. Additionally, MRSHv2 includes a comparison mode for fragment detection, in which the sum of the best matches is divided by the number of filters in the smaller hash. Like ssdeep, MRSHv2 can produce irregular segment sizes, and using Bloom filters adds another factor affecting the accuracy.

Unlike CTPH algorithms, sdhash does not divide the file into segments (Roussev, 2010). Instead, it selects statistically improbable fixed-length segments based on their entropy and adds them into Bloom filters. Furthermore, sdhash's comparison function is more akin to a fragment detection. Similar to the comparison process in MRSHv2, sdhash performs an all-against-all comparison of the Bloom filters. In fact, the comparison function used in MRSHv2 was derived from the approach first introduced in sdhash.

TLSH generates a fixed-length hash representing every 5-gram in an input file (Oliver et al., 2013). Additionally, it includes a hash header, providing information about the file's overall size. When comparing TLSH hashes, the distance between the headers and the bodies of the hashes is calculated. Although the TLSH distance score can exceed 1000, a score above 300 typically indicates dissimilarity. To align TLSH scoring with other algorithms, we apply the normalization technique introduced by Upchurch and Zhou (2015), where the distance is normalized by subtracting the score from 300 and dividing by 3, thereby converting it to a 0 to 100 scale. This normalization is consistently applied to match the TLSH scoring with other similarity scores in our research.

5. Data sets

This section provides details on the data sets we created for our experiments.

5.1. Corpora

All experiments are based on three main corpora: a malware corpus, a third-party software corpus, and one containing operating system files. The following sections provide an overview of each corpus and the characteristics of the samples included.

Malware: We used the packed and unpacked malware samples from Malpedia (Plohmann et al., 2017), covering a diverse collection of representative samples targeting both Linux and Windows operating systems. Table 1 provides the numbers of packed and unpacked samples across different operating systems.

Third-Party Software: The third-party software corpus consists of widely used, benign applications with varied update cycles available for Windows and Linux. We selected Mozilla Firefox and Thunderbird, VLC, and GIMP, downloaded their installers from the official websites, and extracted the main executable. For the Linux variants of GIMP and VLC, we used the versions available in the package repositories of the Ubuntu versions 22.04, 23.10, and 24.04, as the maintainers do not provide precompiled versions on their download pages. Table 2 lists the number of executables per software.

Operating System Files: Another category of benign software includes operating system (OS) files. OS files are present on every system of that specific type, and their update cycles are often tied to the OS. They are usually deeply integrated into the system and cover a variety of program sizes and functionalities. Our Linux OS corpus consists of all standalone ELF files located in `/usr/bin/` and `/usr/sbin/` taken from the Ubuntu versions 22.04, 23.10, and 24.04, as well as Fedora 40. The Windows part of the corpus includes all .exe files taken from a new Windows 10 and 11 installation. This selection ensures a broad coverage of benign executables across multiple operating systems and environments.

5.2. Dumps generation

To compare memory-mapped files to their on-disk counterparts, we created a corpus using two virtual machines: a Windows 10 22H2 system to create the Windows PE dumps and an Ubuntu 22.04.4 system to create the ELF dumps. The respective operating system versions have been chosen for convenience, as they were available in our lab. Table 3 summarizes the statistics of all dumps created. Several dumps could not be created because of early process termination.

5.2.1. Windows PE dumps

We included all stand-alone executable files from Malpedia for the PE dumps, excluding .NET binaries due to their differing behavior, such as using an intermediate language and a runtime. For each file, we

Table 1
Number of samples and families in the malware corpus.

Malware Corpus	Samples	Families
Packed PE	2670	978
Unpacked PE	3531	1384
Packed ELF	161	98
Unpacked ELF	269	165

Table 2
Number of third-party software versions.

	Firefox	Thunderbird	GIMP	VLC
Linux	1748	725	14	24
Windows	1341	475	71	46

Table 3
Dump corpora statistics.

Operating System	On-Disk Versions	Successful Dumps	
		at entry point	after process attach at main
Windows 10 22H2	Packed (2670)	2584	1143
	Unpacked (1479)	1397	652
	OS files (859)	833	240
Ubuntu 22.04	Packed (161)	138	22
	Unpacked (178)	164	30
	OS files (902)	902	899

created dumps at two different points in time during the execution.

Dump at Entry Point: Here, we dump the process at a very early execution stage. We initiate WinDbg and load the PE file into memory. Once running, WinDbg halts execution at the entry point, where the file's sections have already been mapped into memory according to its PE headers. If the file has been relocated, code section addresses may differ from their original values on disk; further eventual modifications like thread local storage callbacks might also have altered the memory. Then, we extract the memory range of the mapped PE file utilizing the `ImageBase` and `SizeOfImage` fields from the Optional Header.

Dump after Process Attach: In this case, we attach WinDbg to the running process after a five second delay, pausing the execution. This dump allows us to analyze the in-memory state after a short execution period beyond the entry point, possibly comprising more modifications.

5.2.2. Ubuntu ELF dumps

Both packed and unpacked x86-64 ELF samples from Malpedia were included for ELF dumps. The dumps were created using GDB, the mapped segment address ranges were extracted from `/proc/<pid>/maps`, and the contents of these segments were concatenated to form the dump file. For OS files, the dumps were captured at the first instruction of the main function, rather than after five seconds, as many command-line utilities from `/usr/bin/` terminate too quickly. The following paragraphs provide an overview of the different dumping approaches.

Dump at Entry Point: We launch the ELF file using GDB, and capture the memory-mapped segments at the first instruction. At this time, the dynamic linker has loaded the shared libraries, resolved symbol addresses, and handled relocations. Control is handed over to the program, and the `LOAD` segments from the ELF header have been mapped into memory for execution.

Dump at Main: GDB launches the OS program, sets a breakpoint at the main function, and captures the memory-mapped ELF segments after reaching the breakpoint. At this time, the dynamic linker has loaded the necessary shared libraries, applied relocations, and executed any constructor functions.

Dump after Process Attach: We attach GDB to the running process five seconds after launching the ELF file. The dumping process follows the steps similar to previous scenarios, although we collected fewer dumps as the process often exited before GDB could attach.

6. Identified Implementation Issues

Implementation issues can significantly affect the reliability and integrity of experimental results, limiting the validity of conclusions based on the corresponding evaluations. This section discusses bugs we encountered in MRSHv2 and sdhash, analyzes their root causes, and evaluates their effects on the experimental outcomes.

6.1. MRSHv2

In MRSHv2, we found a bug in the logic comparing two hash values. The bug manifests in similarity scores underestimated or overestimated depending on the detection mode. In the regular mode, the score can occasionally exceed 100; in the fragment mode, the score can be halved.

MRSHv2 removes Bloom filters that contain only a small number of segments from the input file. However, this removal happens after determining which of the two hash values is smaller. This sequence can lead to inconsistencies in calculating the comparison score.

To illustrate this issue, consider a hypothetical scenario involving two hash values, h and h'. h and h' each comprise two Bloom filters, denoted as bf_1, bf_2 and bf'_1, bf'_2, respectively. Initially, h is identified as the smaller fingerprint due to the input order, as both h and h' contain the same number of Bloom filters. Subsequently, MRSHv2 removes Bloom filters with fewer segments than a predefined threshold. As part of this process, bf'_2 is removed from h' because it does not meet the minimum segment count requirement. Consequently, h' is effectively reduced to one Bloom filter, while h retains its two filters. Despite this change, the comparison score is calculated based on the initial determination that h is the smaller fingerprint. This behavior leads to incorrect results, as described above.

6.2. sdhash

The bug in sdhash was more significant in our experiments. Moreover, we assume that the bug caused the abnormal behavior reported in a GitHub issue (Vitale, 2022) and the paper of Fleming and Olukoya (2024). The issue arises during the hash generation, specifically when segments are inserted into a Bloom filter, causing it to exceed its configured capacity.

In version 4.0[2], an optional `-large` flag was introduced[3], which can be used to create larger, more fine-grained Bloom filters. However, even if the flag is not set, the larger Bloom filters are active and track inserted features—likely to ensure that identical features are not repeatedly inserted across multiple smaller Bloom filters.

When the insertion of a feature into the larger Bloom filter fails, indicating that the feature already exists, the implementation does not increment the counters for any of the filters. Hence, a feature may be inserted into a smaller Bloom filter without increasing its counter. This inconsistency causes an inflated number of features in this filter, even when those features are duplicates. As a result, the Bloom filter capacity can become critically overfilled, leading to inflated similarity scores because the filters are more likely to overlap during comparisons. We evaluated the severity of this issue by hashing more than 20 000 distinct inputs. Remarkably, this bug affected more than 50 % of the inputs. The issue was particularly critical in more than 3.5 % of the hashes, resulting in Bloom filters with more than half of their bits set, as illustrated in Fig. 1.

We resolved these issues in MRSHv2 and sdhash and incorporated the corrected and original versions into our experiments. Since MRSHv2 showed no significant differences in our experiments between the original and the fixed version or comparison modes, only the default version will be presented in this work. We provide the complete set of results in the supplemental material.

[2] A reviewer of this paper pointed out that version 3.4 is the latest stable version, which is also stated on the sdhash homepage. However, the latest release on GitHub is version 4.0, and several research papers have used this version in their evaluations. Hence, we opted to use this version.

[3] commit 0d2366f0ac1c627632774891e4ee20e9acfd702d.

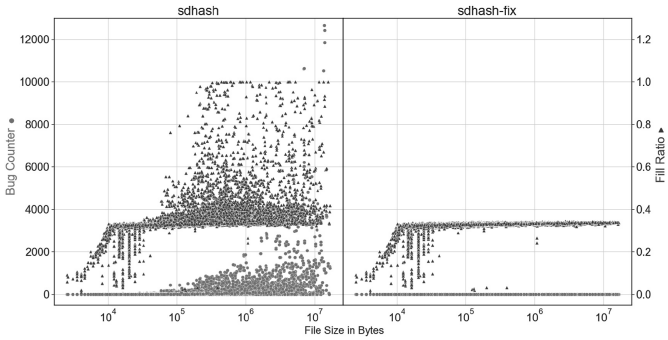


Fig. 1. Frequency of the sdhash error. The fill ratio is plotted for the most populated Bloom filter of a hash only. For smaller files, the bug occurs less frequently, as the Bloom filters are not filled beyond their capacity.

7. Assessing the TLSH comparison function

Our evaluations showed that TLSH often classified files as similar at low threshold values, even when highly dissimilar. We probed the cause of this and found that this behavior arises from the design of the comparison function. Here, the distance score is computed based on a body distance metric, in combination with header attributes, including an encoded data length.

The body distance metric is calculated by summing the pairwise distances between 128 buckets, where each bucket comparison contributes a value of 0, 1, 2, or 6, depending on the difference in their quartile encodings. In a randomized setup, where the buckets are filled uniformly at random, each bucket has a uniform probability of $\frac{1}{4}$ of falling into one of the quartile encodings (00, 01, 10, 11). Using this setup, one can calculate that the expected distance between two buckets is 1.625. With 128 buckets, as per default, this results in a total expected distance of 208. The header distance, on the other hand, is primarily influenced by the size difference between input files. Consequently, files with similar sizes tend to have a minimal header distance score. This behavior was experimentally confirmed through pseudo-random files and a comparison of packed PE files and Windows OS files, which are expected to show substantial differences at the binary level (cf. Appendix A).

As a result, the normalized TLSH score is likely to exceed 30 for files with similar sizes, even when the files are entirely dissimilar, and in extreme cases, it may even reach 50. Hence, we emphasize that even scores of 50 do not necessarily indicate any meaningful similarity between the inputs. The fact that two files have similar sizes may already contribute to a baseline similarity of 30. One should be aware of this behavior and apply similarity thresholds greater than these values to avoid misinterpretation.

8. Operating system files vs. Malware

This section presents the results of our experiments comparing operating system files with malware. We used the corpus described in Section 5.2, including the memory-mapped files and their on-disk counterparts. We compared the on-disk version of unpacked and packed malware samples with the on-disk version of the OS files from the same OS (i.e., Linux malware with Linux OS files and Windows malware with Windows 10 OS files). Moreover, we compared the malware dumps with the dumps of the OS files.

Fig. 2 displays the relationship between threshold values and the percentage of matched pairs of packed PE malware samples and OS files based on their similarity scores. The x-axis represents the threshold values, while the y-axis indicates the percentage of comparisons where the similarity score between a benign OS file and a packed malware sample exceeds the threshold.

For TLSH, a significant portion of malware samples and OS files were

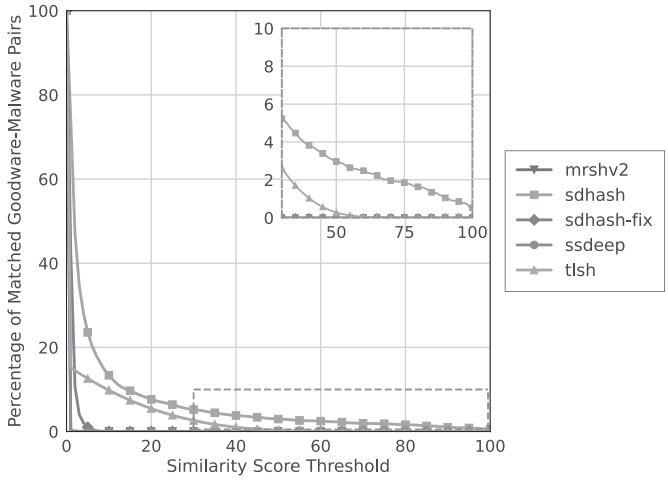


Fig. 2. Packed PE malware samples vs. Windows OS files. Pairs are considered as matched when the computed similarity score exceeds the given threshold.

classified as similar at threshold values up to 40. This result stems from the comparison function, in which files with similar sizes have a baseline similarity exceeding 30, as outlined in Section 7.

sdhash also classifies many malware and OS files as similar, even at higher thresholds. More than 0.5 % of the comparisons lead to a 100 % match, typically indicating exact duplication or full containment. This stems from the error previously described where Bloom filters are filled beyond their capacity. The fixed version exhibits considerably fewer similarities at higher thresholds, affirming that the issue is tied to the problem in the original implementation. However, compared to the CTPH approaches, sdhash-fix more frequently detects minor similarities between OS files and malware. The design of its comparison function, which operates like fragment detection, is mainly responsible for this behavior.

When evaluating unpacked malware vs. OS files, we observed an increasing number of OS and malware files being classified as similar, which was expected given the reduced obfuscation and closer resemblance to legitimate software. The dump comparisons showed similar trends to the on-disk results, with some in-place modifications done by malware that increased the similarity to OS files. This was eminently evident in an instance where we suspect that a malware sample mapped the legitimate `conhost.exe` binary into its memory-mapped region.

While exploring why some OS files were matched with malware, we discovered that certain malware and OS files share identical entries in their `.rsrc` section. When this section constitutes a large portion of the file, the approximate matching algorithms tended to output higher scores. In a particular case involving the OS file `AxInstUI.exe`, we discovered that the section was disproportionately large due to the inclusion of default icons. This effect was less pronounced in the CTPH algorithms.

Another finding is that comparing malware and OS files written in Go often leads to high scores, likely due to shared bundled libraries and runtime components. This was eminently evident in the Linux OS program `ipp-usb`, which matched various Go-written ELF malware samples.

Lastly, certain malware and OS file pairs showed higher TLSH scores, which we attribute to the presence of shared statically linked code. Using the code relationship toolkit MCRIT (Plohmann, 2024), we determined that this shared code predominantly corresponds to Visual Studio C runtime functions.

9. Software updates

The software update experiments are structured into three subcategories: malware samples, third-party software, and operating

system executables, taken from the corpora described in Section 5.1.

9.1. Malware

We conducted an all-against-all comparison between the malware samples for the same operating system. While several works have already evaluated approximate matching to classify malware (Shiel and O'Shaughnessy, 2019; Botacin et al., 2021; Fleming and Olukoya, 2024), we include this evaluation for completeness and with a systematized, publicly available, and widely used data set. Moreover, we compared every sample of each family with all other samples of the same family. Finally, as a case study, we did an intra- and inter-family comparison for Zeus-related samples. The goal of the experiments is to assess if samples belonging to the same family or sharing a codebase due to leaks, code reuse, or other factors are more similar to each other.

All-against-all Comparison: This section presents the results for comparing unpacked Windows PE samples. Naturally, the algorithms should perform better on unpacked samples, as the purpose of the packers is to introduce dissimilarities. This assumption was confirmed by our results comparing packed samples. These results and the results for ELF binaries, which are comparable to the PE file results, are shown on the website accompanying this paper.

To provide metrics on the algorithms' classification performance, we calculated the precision, recall, and F1 score, displayed in Fig. 3. A perfect precision score of 1 indicates that all detected matches belonged to the same family, while a recall score of 1 means that all possible matches were identified. The F1 score is the harmonic mean of precision and recall, where a score of 1 denotes an ideal precision and recall. We count a true positive when a sample is matched with another sample from the same family; a false positive occurs when a sample is matched with any sample from a different family. Malpedia is used as the ground truth for the family labels of the samples.

None of the algorithms achieved satisfactory scores, with ssdeep performing particularly poorly concerning the recall, failing to match most samples. MRSHv2 performed slightly better due to its increased block count, but the recall declined at thresholds above 20. The original sdhash exhibited a near-zero precision, which we attribute to the implementation bug. Once resolved, the algorithm outperformed the CTPH approaches, as the recall did not decline as rapidly. TLSH behaved differently, achieving high precision only at thresholds above 80. For lower thresholds, the algorithm tends to be overly optimistic, often matching samples from different families.

Although we did not account for actual similarities shared between malware families, the scores remain unsatisfactory. One could expect that with a sufficiently large threshold, even for families sharing a code base, an optimal process would be able to detect their dissimilarity and classify them appropriately.

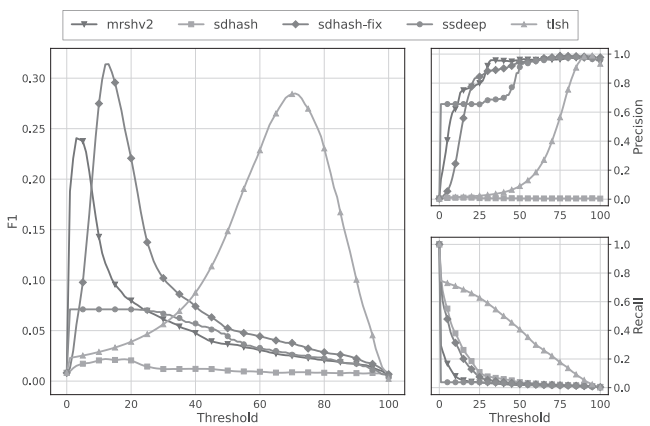


Fig. 3. F1 Score, Precision, and Recall of the malware family classification process for Windows PE samples.

Intra-Family: For the intra-family comparison of neighboring versions, we selected the ten families with the most versions available from our data set, shown in Fig. 4. The versions were sorted based on the version information provided by Malpedia, and those without such information were sorted exclusively by commit time.

Notably, as indicated by the recall, none of the algorithms consistently matched all samples to their respective families. CTPH algorithms primarily matched samples only to themselves. sdhash performed slightly better, albeit the noted implementation bug is clearly visible for certain Trickbot, Coldseal, and Cobra versions. The dark horizontal and vertical lines in the corresponding heatmaps of Fig. 4 indicate that those versions matched all other versions. In contrast, the other algorithms, especially the fixed sdhash version, did not yield such similarities.

Lastly, TLSH showed the highest similarity scores of all algorithms but failed to match all versions within the same malware family. Considering the small-scale assessment of Pagani et al. (2018), which suggests that TLSH is effective in handling artificial source code modifications in malware, it appears that TLSH can somewhat reliably match different real-world instances of the same malware family as well. However, based on the previous results from comparing OS files with malware, it should be considered that TLSH is optimistic in general. Hence, an appropriately high threshold is required. Then, TLSH might be moderately helpful in identifying similar versions within an already labeled set of files of the same family, reducing the number of versions needing detailed analysis.

Zeus-related: Finally, we present a case study comparing Zeus-related families. The leak of Zeus' source code led to several related malware families, and one could expect these families to exhibit some similarities. We obtained information from ZeusMuseum (Schwarz, 2024) to identify the families associated with Zeus. Fig. 5 compares Zeus variants ordered by their appearance using the TLSH algorithm. The other algorithms detected fewer similarities; their results are in the supplemental material.

TLSH detects similarities between earliest families, with Citadel, Kins, and VMZeus showing the highest similarity. However, Zeus OpenSSL only exhibits similarities with itself due to its static linking of an OpenSSL version. This statically linked library constitutes a significant portion of the binary and thus reduces its overall similarity with other samples. Similarly, Zloader does not show similarity with samples from other families. Despite sharing some functions with Zeus, it is primarily a loader for Zeus OpenSSL rather than a banking trojan (Fraunhofer FKIE, 2024).

Overall, the results of this case study turned out to be comparatively positive for TLSH. Most families exhibited high similarity scores with other Zeus-based families. Lower scores, on the other hand, are reasonable because of added or removed code. Still, it should be kept in mind that those detected bytewise similarities do not imply that TLSH is able to match malware with similar functionality. It is important to clearly distinguish between binary similarity and semantic similarity.

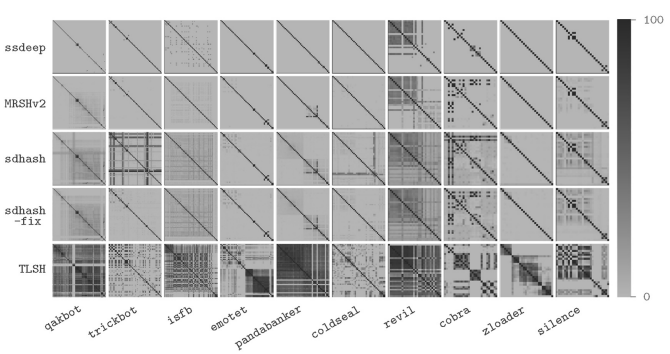


Fig. 4. Similarity scores for the intra-family comparison of the ten most popular families from our data set.

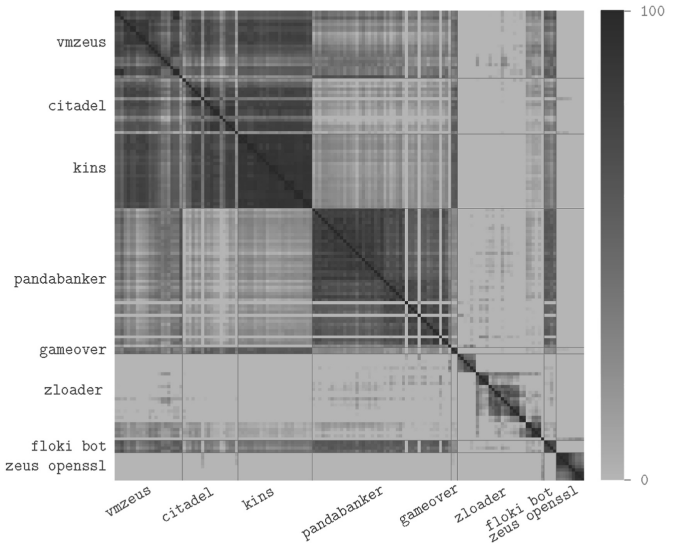


Fig. 5. Normalized TLSH scores for the all-against-all comparison of Zeus-related samples.

9.2. Third-party software

Here, we present a comparison between releases of third-party software. In an investigation, filtering versions of known programs is a common approach to direct the analysis to more relevant files.

All algorithms detected similarities between neighboring versions, with TLSH grouping the largest number of versions. Even the more conservative CTPH algorithms highlighted blocks in the heatmaps in Fig. 6, indicating updates with larger and smaller changes.

For Firefox on Windows, however, the sdhash bug was particularly severe, matching the majority of versions with scores above 90. While this might suggest high version similarity, the match rates stem from the bug rather than genuine bytewise similarity. Likely, sdhash would also erroneously match Firefox with unrelated software.

With TLSH, the identification of different versions was sometimes less clear than with the other algorithms. In some cases, such as with GIMP on Windows, TLSH matched later versions with very early ones. This is not necessarily due to code reuse but could also be attributed to the design of its comparison function.

In summary, the algorithms, particularly TLSH, seem to be able to match different software versions to some extent.

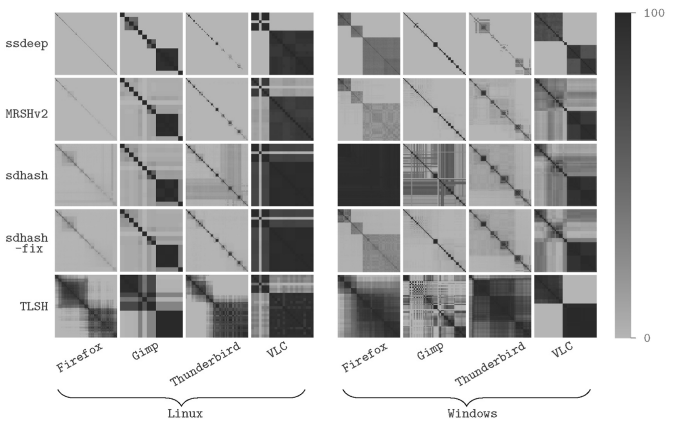


Fig. 6. Similarity scores for comparing releases of benign software products for Windows and Linux.

9.3. Operating system files

For this assessment, we compared files in identical paths with the same names to reflect version updates across different operating systems. Our main objective was to determine if approximate matching can effectively filter out operating system files across different OS releases.

Results: Fig. 7 includes comparisons between Windows 10 and 11 and different Linux distribution versions. Performance was weakest with ssdeep, as most files received a score of 0. MRSHv2, as a more resilient CTPH algorithm, showed a slight improvement, though most files scored low, with a few exceptions.

While the inaccuracies of ssdeep stem from its short hash length, MRSHv2 suffers from comparing multiple Bloom filters. If functions or code regions are reordered during compilation, they are likely to change their respective Bloom filters, too, leading to a decreased score. In contrast, sdhash produces higher scores due to its fragment detection-like design and finer granularity, with each Bloom filter representing around 10 kB of data compared to MRSHv2's 25 kB in the current version. TLSH is the only algorithm effectively detecting similarities across OS versions, producing more consistent and comprehensible scores. This is due to file sizes remaining similar across versions, and reordering code does not affect the resulting hash as significantly as with the other approaches.

10. On-disk vs. In-Memory

Here, we compared the on-disk file version of executables with their dumped in-memory representation. We showcase two scenarios representative of the overall evaluation. The first focuses on the influence of the dump time, the second on the differences between 32-bit and 64-bit and the impact of relocations. For the former, we refer to packed 64-bit PE samples that have not been relocated in memory; for the latter, we refer to unpacked PE samples.

Impact of Capture Time: Fig. 8 illustrates the differences in capture time, comparing dumps at the entry point with their on-disk counterpart on the left and dumps captured after five seconds of execution on the right. We picked packed PE samples to display the effect of in-place modifications. For unpacked malware and OS files, the difference between capture times was less significant, which is expected due to the absence of in-place unpacking or overlay inconsistencies in both, as well as the absence of obfuscation in OS files.

As anticipated, dumps at the entry point show consistently high scores, as only minimal changes are introduced to the memory before reaching the entry point. After five seconds of execution, the similarity scores decline due to in-place modifications, including unpacking and deobfuscation. Still, the similarity scores between on-disk and in-memory versions remain relatively high even after this time. This may be due to unpacking occurring in different memory regions or via process injection affecting other processes, leaving the original sections intact. Another possibility is that unpacking is delayed or prevented by

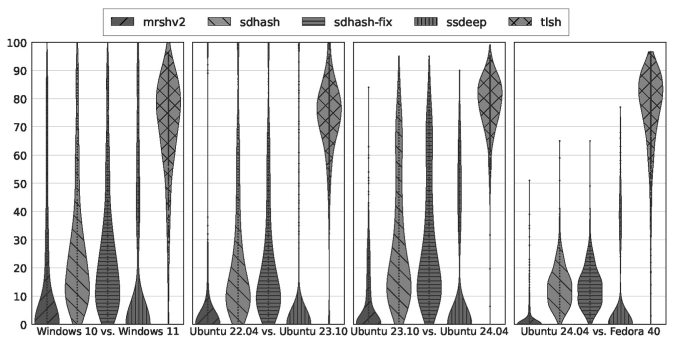


Fig. 7. Similarity score distribution for comparing the same files across different OS versions.

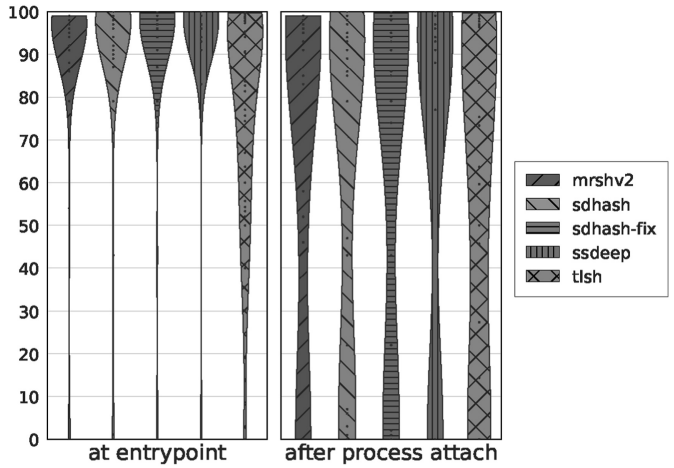


Fig. 8. Similarity score distribution for comparing packed 64-bit PE samples with their in-memory not-relocated version.

anti-analysis techniques.

TLSH stands out due to several comparisons yielding lower scores. We traced this back to large sections of null bytes, frequently occurring in files with substantial differences between raw and virtual section sizes, such as UPX-packed samples. In contrast, other approaches do not struggle with such areas of null bytes. CTPH algorithms do not trigger within those null byte sections, and sdhash filters out low-entropy regions during its feature selection process, effectively ignoring the null bytes.

For PE malware samples, another factor that can cause low similarity between on-disk and memory-mapped versions is the presence of an overlay—data appended after the last section of the file. The overlay may serve various purposes, such as holding embedded resources or any data the malware might process later. Since the overlay is located outside of the memory-mapped range, it is not loaded into memory upon execution. As a result, if a PE file contains a large overlay that constitutes a significant part of the on-disk version, ssdeep and TLSH yield low scores. In contrast, MRSHv2 and sdhash still produce higher scores because their comparison functions are designed for fragment identification. There, the missing overlay in the memory-mapped version has minimal impact as the comparison function still detects the memory-mapped version as a fragment of the on-disk file.

Impact of Relocation: Fig. 9 shows the results for unpacked PE samples dumped at the entry point to highlight the differences between relocated and non-relocated samples. We classified a dump as relocated if the Optional Header's `ImageBase` value of the on-disk and memory-mapped file version did not align and the file contained a `.reloc` section.

The influence of relocation on similarity scores is particularly pronounced for 32-bit PE samples. While 64-bit samples use RIP-relative addressing, reducing the need for relocations, 32-bit samples rely on fixed addressing, resulting in more frequent relocations. Since 32-bit relocation introduces localized modifications, most algorithms experience a drop in similarity scores. ssdeep is particularly affected due to its short hash length since many of the 32 to 64 segments will likely be modified during relocation. In contrast, TLSH is much less affected by relocations, a finding consistent with Pagani et al. (2018). This is because most 5-grams in the binary remain unchanged and the similarity is primarily determined by how closely the overall count of different 5-grams align.

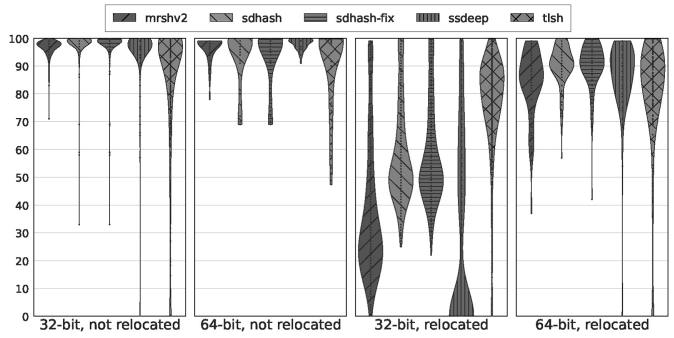


Fig. 9. Similarity score distribution for comparing unpacked PE samples with their in-memory version, dumped at the entry point.

11. Conclusion

During forensic investigations, the concept of similarity is not always straightforward. Its interpretation can vary depending on the concrete scenario, and the results of bytewise approximate matching algorithms may not always align with the analyst's understanding of similarity. Therefore, it is vital to understand the nuances, strengths, and limitations of these algorithms for different use cases. Our research has contributed to this for executable files, demonstrating that bytewise approximate matching should not be used as a standalone, all-encompassing solution for most scenarios we considered. It can merely serve as a component of more conceived approaches.

The following paragraphs summarize our key findings and give directions for future research. Although some of our results are not particularly surprising and confirm trends known to the community, we argue that our evaluations and especially our explanations of the reasons for observed effects, contribute to a better understanding of bytewise approximate matching.

Detecting Updates: Overall, the algorithms performed poorly in malware family classification. We suggest using them only as an assisting factor in more sophisticated classification models. However, an intra-family comparison revealed that TLSH may be moderately helpful in identifying similar versions within a labeled set, reducing the number of versions for detailed analysis.

For matching different versions of third-party software and OS files, TLSH emerged as the most suitable algorithm. This makes TLSH useful for detecting updated software versions, reducing the need to analyze previously examined files. In this context, TLSH can complement semantic and syntactic approaches. However, to maximize its effectiveness, a comprehensive hash database and sufficiently high similarity score thresholds are necessary.

Highlighting and Safelisting: The investigated algorithms are hardly suitable for distinguishing between benign and malicious software and thus only limitedly suitable for block- and safelisting. In some cases, we observed that OS files are matched to malware due to using language-specific runtime environments, statically linked libraries, or shared resources. These files indeed share a bytewise similarity, resulting in the misclassification of malware as benign software and vice versa. Therefore, it is advisable to use higher thresholds for safelisting or applying syntactic methods, approving their higher runtime and giving up file type independence. For instance, separately hashing PE sections could yield more accurate results, as already suggested in prior research (Shiel and O'Shaughnessy, 2019; Botacin et al., 2021). In general and as indicated above, block- and safelisting with these algorithms only seem to make some sense if a sufficiently large number of versions are present in the database.

Memory Forensics: Caution is required when assessing memory-mapped files with similarity hashes of on-disk files. Especially (packed) malware can introduce elaborate changes to memory-mapped files, resulting in notable differences from their on-disk counterparts.

The fixed sdhash version is the most suitable for analyzing memory-mapped files among the evaluated algorithms. This fitness arises from sdhash's ability to handle size-related variations and regions of null bytes. In contrast, TLSH struggles with large areas of null bytes, making

its use on memory-mapped files inadvisable without preprocessing. On the other hand, in scenarios where a 32-bit PE file has been relocated in memory, TLSH is the only viable algorithm, as all other algorithms fail to overcome the related localized modifications introduced.

None of the algorithms are fully reliable on their own, as each struggles in different areas: sdhash and CTPH approaches with relocation, and TLSH with null bytes and overlays, highlighting the need for a syntactic approach such as the methods described in Martín-Pérez et al. (2021), to address these limitations.

Implementation Issues and Design Specifics: We identified an implementation bug in sdhash that distorted almost all of our experiment results. Additionally, we found a minor issue in the MRSHv2 comparison function, in which the smaller hash is occasionally misidentified, resulting in faulty similarity scores. We fixed both errors and shared our findings with the corresponding authors. Furthermore, we unrolled why the comparison function of TLSH behaves overly optimistic, complicating the interpretation of its scores.

Future Work: Our work indicated that large portions of common statically linked code, such as runtime environments, increase the similarity between otherwise distinct programs. Since modern languages like Go or Rust exhibit this behavior, more systematic evaluations of such programs are required. Moreover, we are currently evaluating how to create an efficient and effective database to enable better version detection without requiring a hash of every program version. Finally, results of previous evaluations using the reference implementations of sdhash and MRSHv2 have to be reassessed to exclude that they were affected by the bugs we discovered.

Appendix A. Details on the TLSH Body Distance

Given two hash values h and h' and their respective bucket lists b and b' consisting of 128 buckets, the expected body distance $\mathbb{E}[X]$ can be calculated under the assumption that the buckets are uniformly filled at random. This calculation can be performed using the distance table A.4:

$$\mathbb{E}[X] = 128 \cdot \sum_{d \in \{0,1,2,6\}} P(X_i = d) \cdot d = 128 \cdot \left(0 \cdot \frac{1}{4} + 1 \cdot \frac{3}{8} + 2 \cdot \frac{1}{4} + 6 \cdot \frac{1}{8}\right) = 128 \cdot 1.625 = 208$$

Table A.4
Pairwise Distance Table for Bucket Comparisons

$\lvert b_i - b'_i \rvert$	b_i-b'_i	$P\ (X_i = d)$
0	00-00, 01-01, 10-10, 11-11	$\frac{4}{16} = \frac{1}{4}$
1	00-01, 01-00, 01–10, 10–11, 10-01, 11-10	$\frac{6}{16} = \frac{3}{8}$
2	00-10, 01–11, 10-00, 11-01	$\frac{4}{16} = \frac{1}{4}$
6	00-11, 11-00	$\frac{2}{16} = \frac{1}{8}$

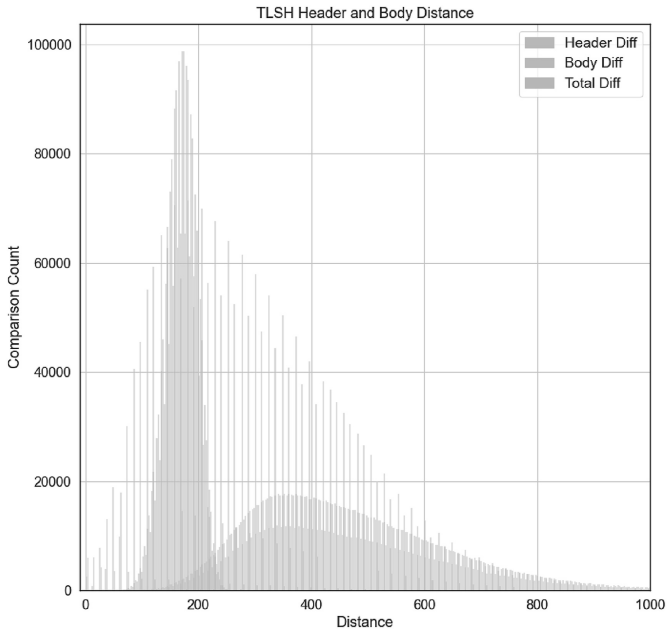


Fig. A.10. TLSH body distance and header distance calculation of packed PE and Windows OS file comparisons. The body distance is averaged around 170, which, for files of similar length, results in a normalized similarity score of above 40.

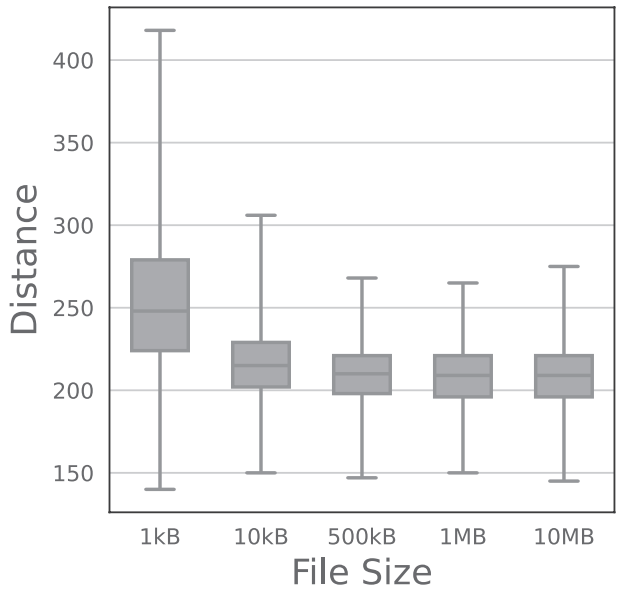


Fig. A.11. A comparison of differently seeded pseudo-random files using the TLSH algorithm. As the file size increases, the calculated distance approaches the expected body distance score of 208. The overall distance is slightly higher due to the header distance, which is influenced not only by the difference in lengths but also by quartile ratio differences. Variations in the results are due to chance, with distances lower than 150 — equivalent to a 50 % similarity — being possible.

References

Ali, M., Hagen, J., Oliver, J., 2020. Scalable malware clustering using multi-stage tree parallelization. In: IEEE International Conference on Intelligence and Security Informatics (ISI), pp. 1–6.

Bak, M., Papp, D., Tamás, C., Buttyán, L., 2020. Clustering IoT malware based on binary similarity. In: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, pp. 1–6.

Botacin, M., Moia, V.H.G., Ceschin, F., Henriques, M.A.A., Grégio, A., 2021. Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios. Forensic Sci. Int.: Digit. Invest. 38, 301220.

Breitinger, F., Baier, H., 2013. Similarity preserving hashing: eligible properties and a new algorithm MRSH-v2. In: Proceedings of the 4th International Conference on Digital Forensics and Cyber Crime (ICDF2C 2012), pp. 167–182.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014a. Approximate Matching: Definition and Terminology (NIST SP 800-168). Special Publication. National Institute of Standards and Technology.

Breitinger, F., Winter, C., Yannikos, Y., Fink, T., Seefried, M., 2014b. Using approximate matching to reduce the volume of digital data. In: Advances in Digital Forensics X: 10th IFIP WG 11.9 International Conference, Vienna, Austria, January 8-10, 2014, Revised Selected Papers 10. Springer, pp. 149–163.

Coffman, J., Chakravarty, A., Russo, J.A., Gearhart, A.S., 2018. Quantifying the effectiveness of software diversity using near-duplicate detection algorithms. In: Proceedings of the 5th ACM Workshop on Moving Target Defense, pp. 1–10.

Fleming, M., Olukoya, O., 2024. A temporal analysis and evaluation of fuzzy hashing algorithms for android malware analysis. Forensic Sci. Int.: Digit. Invest. 49, 301770.

Fraunhofer FKIE, 2024. Malpedia – Zloader (malware family). URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.zloader, 2024-09-20.

Harichandran, V.S., Breitinger, F., Baggili, I., 2016. Bytewise approximate matching: the good, the bad, and the unknown. J. Digit. Forens. Secur. Law 11, 4.

Hutelmyer, P., Borre, R., 2024. Implementing TLSH based detection to identify malware variants. URL: https://tech.target.com/blog/implementing_TLSH_based_detection.

Jakobs, C., Lambertz, M., Hilgert, J.N., 2022. ssdeeper: evaluating and improving ssdeep. Forensic Sci. Int.: Digit. Invest. 42, 301402.

Kida, M., Olukoya, O., 2023. Nation-state threat actor attribution using fuzzy hashing. IEEE Access 11, 1148–1165.

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digit. Invest. 3, 91–97.

Li, Y., Sundaramurthy, S.C., Bardas, A.G., Ou, X., Caragea, D., Hu, X., Jang, J., 2015. Experimental study of fuzzy hashing in malware clustering analysis. In: 8th Workshop on Cyber Security Experimentation and Test (CSET 15). USENIX Association.

Liebler, L., Baier, H., 2019. Towards exact and inexact approximate matching of executable binaries. Digit. Invest. 28, S12–S21.

Liebler, L., Breitinger, F., 2018. mrsh-mem: approximate matching on raw memory dumps. In: 2018 11th International Conference on IT Security Incident Management & IT Forensics (IMF). IEEE, pp. 47–64.

Magonia Research, 2023. CelesTLSH malware scanner. URL. https://www.magonia.io/products/celestlsh-malware-scanner.

Martín-Pérez, M., Rodríguez, R.J., Balzarotti, D., 2021. Pre-processing memory dumps to improve similarity score of Windows modules. Comput. Secur. 101, 102119.

Mercès, F., 2020. Telfhash: an algorithm that finds similar malicious ELF files used in Linux IoT malware. Technic. Rep. Trend Micro Res.

Naik, N., Jenkins, P., Savage, N., Yang, L., Boongoen, T., Iam-On, N., 2021. Fuzzy-import hashing: a static analysis technique for malware detection. Forensic Sci. Int.: Digit. Invest. 37, 301139.

Namanya, A.P., Awan, I.U., Disso, J.P., Younas, M., 2020. Similarity hash based scoring of portable executable files for efficient malware detection in IoT. Future Gener. Comput. Syst. 110, 824–832.

Nguyen, T., Feng, G., Pfadler, A., Poliakova, A., Fakeri-Tabrizi, A., Liu, H., Yuzifovich, Y., 2022. Detect emerging malware on cloud before virustotal can see it. J. Cybercr. Digit. Investigat. 7, 7–16.

Oliver, J., Cheng, C., Chen, Y., 2013. TLSH – a locality sensitive hash. In: Fourth Cybercrime and Trustworthy Computing Workshop. IEEE, pp. 7–13.

Oliver, J., Forman, S., Cheng, C., 2014. Using randomization to attack similarity digests. In: Applications and Techniques in Information Security: 5th International Conference, ATIS 2014, Melbourne, VIC, Australia, November 26-28, 2014. Proceedings 5. Springer, pp. 199–210.

Pagani, F., Dell'Amico, M., Balzarotti, D., 2018. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, pp. 354–365.

Plohmann, D., 2024. MinHash-based code relationship & investigation toolkit (MCRIT). URL: https://github.com/danielplohmann/mcrit, 2024-09-02.

Plohmann, D., Clauss, M., Enders, S., Padilla, E., 2017. Malpedia: a collaborative effort to inventorize the malware landscape. J. Cybercr. Digit. Investigat. 3, 1–19.

Roussev, V., 2010. Data fingerprinting with similarity digests. In: Advances in Digital Forensics VI: Sixth IFIP WG 11.9 International Conference on Digital Forensics, pp. 207–226.

Roussev, V., 2011. An evaluation of forensic similarity hashes. Digit. Invest. 8, S34–S41.

Schwarz, D., 2024. zeusmuseum. URL. https://zeusmuseum.com/, 2024-09-11.

Shiel, I., O'Shaughnessy, S., 2019. Improving file-level fuzzy hashes for malware variant classification. Digit. Invest. 28, S88–S94.

Upchurch, J., Zhou, X., 2015. Variant: a malware similarity testing framework. In: Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE). IEEE, pp. 31–39.

Vitale, A., 2022. Wrong result in malware samples comparison · Issue #17 · sdhash/sdhash (GitHub). URL: https://github.com/sdhash/sdhash/issues/17, 2024-09-03.

White, D., 2005. NIST national software reference library (NSRL). In: Mid-Atlantic Chapter HTCIA Meeting.