International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I12P118

Volume 72 Issue 12, 144-152, December 2024
© 2024 Seventh Sense Research Group®

Review Article

Advancements in File Similarity Techniques: Traditional and Modern Approaches for Malware Detection

Udbhav Prasad

Corresponding Author : udbhav523@gmail.com

Received: 04 November 2024 Revised: 30 November 2024 Accepted: 17 December 2024 Published: 31 December 2024

Abstract *- Threat hunting, malware analysis and digital forensic techniques often use signatures to identify malicious executables. While cryptographic hashes are helpful for identifying a particular file uniquely, attackers often tailor their malware to particular systems, releasing variants that target different platforms, operating systems, and even specific organizations or governments. As attacks become more sophisticated, security researchers have proposed "similarity" digests that attempt to overcome the limitations of cryptographic hashes and other traditional signatures by detecting variants of an executable. Modern enterprises manage tens of thousands of endpoints with billions of files, making the scalability of the proposed techniques more important than ever. This survey reviews traditional file similarity digests, such as ssdeep, sdhash, and TLSH, alongside emerging technologies like embeddings and vector databases. By classifying and comparing these techniques, the paper highlights their strengths, weaknesses, and practical applications in malware detection. Key contributions include a structured taxonomy of methods and insights into integrating traditional digests with modern vector database solutions for scalable, efficient detection. This work provides a roadmap for future research and development in this critical domain.*

Keywords *- Cybersecurity, Malware detection, File similarity, Fuzzy digests, Vector databases.*

1. Introduction

Enterprise IT organizations manage tens of thousands of endpoints on average [1]. As companies lean into more distributed workforces, the number of IoT-connected devices is projected to grow to 29 billion by 2030 [2]. Forensic examiners face major challenges in threat hunting and malware analysis [3] due to the sheer amount of data available.

Indicators of compromise (IoCs) are high-confidence signals of computer intrusion on a machine or network. Malware detection techniques often rely on IoCs to identify malicious files. These IoCs include cryptographic hashes (e.g., SHA-1, MD5), IP addresses, domain names, file paths, and digital certificates, which are extracted using static or dynamic analysis of executables. These indicators are compared against databases of trusted or malicious signatures, such as the National Software Reference Library (NSRL) [4], which catalogs digital signatures of known software. Similarly, platforms like MalwareBazaar [6] curate malware samples for use in blocklists, while tools like YARA [7] enable advanced static analysis through customizable rulesets for malware pattern detection.

However, static feeds of indicators are cumbersome to maintain and update. Additionally, cryptographic hashes drastically change with minor modifications to the executable, making it easy for malware authors to evade detection. These shortcomings have motivated the development of "similarity" digests, which can be compared with approximate matching functions. These digests are designed such that similar objects produce similar digests, and comparing two digests produces a measure of the similarity between the corresponding files.

The National Institute of Standards and Technology (NIST) defines similarity digests as a compressed representation of the original data object's feature set that is suitable for comparison with other similarity digests created by the same algorithm. Similarity digests can significantly improve file identification rates [8], but they come with computational and storage challenges. Traditional exact-match systems like key-value stores and relational databases are not well-suited for storing and retrieving similarity digests, which often require two steps: feature extraction and digest computation, followed by comparison against reference digests to identify matches. Brute-force comparison is expensive, and several data structures have been proposed to organize the reference digests for more efficient retrieval. The different techniques proposed in the literature encode information in different ways, making it difficult to design a common data structure across similar techniques.

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

Embeddings are vector representations of data objects and are generated using machine learning models. These embeddings enable highly scalable approximate similarity comparisons by evaluating the geometric distance between vectors in a shared feature space. To efficiently store and retrieve these embeddings at scale, vector databases such as Milvus [9], Pinecone [10], and Weaviate [11] have emerged as off-the-shelf solutions. These systems are specifically optimized for handling vector data, offering advanced indexing techniques like HNSW (Hierarchical Navigable Small World) [12] and ANNOY (Approximate Nearest Neighbors Oh Yeah) [13] to accelerate similarity searches.

The “Similarity Metrics” and “Similarity Digests” sections introduce the different algorithms used to measure similarity between files, providing an overview of their mechanisms, their ability to capture similarities, and the computational complexity involved. The “Similarity Search and Clustering” section examines the various search and clustering strategies tailored to these algorithms, discussing their performance characteristics, scalability, and the trade-offs they present. Following this, the paper explores the role of vector databases in addressing the limitations of traditional similarity digest techniques. Building on these insights, the subsequent section proposes an optimal approach that combines TLSH [14] (Trend Micro Locality Sensitive Hashing) and vector databases, bypassing the need for custom implementations such as HAC-T [15] (Hierarchical Agglomerative Clustering for TLSH) by leveraging the robustness of off-the-shelf solutions. The paper finally discusses future work in this area, suggesting stronger integration between language modes and malware analysis.

2. Methodology

This systematic review focuses on similarity techniques on binary or executable files. Specifically, text-based document similarity is not addressed by this review. Modern work on file similarity techniques is evaluated, although older work on similarity metrics is used to motivate the introduction of modern techniques.

The research addresses three key questions: the relative effectiveness of different similarity detection approaches, their computational and storage trade-offs, and the potential impact of modern technologies like vector databases on scalability.

Techniques are selected based on their practicality in real-world production environments. Security practitioners across the globe currently use the approaches reviewed by this paper for malware detection and threat analysis. Papers were also selected based on the novelty of their algorithms, the magnitude of improvement over existing techniques and comprehensive performance evaluations.

The evaluation and analysis framework evaluates five key factors: detection accuracy, computational efficiency, storage requirements, search performance, and implementation complexity. The methodology emphasizes empirical evidence and quantitative metrics where available, supplemented by qualitative analysis of implementation challenges and operational considerations. This balanced approach provides insights relevant to both theoretical advancement and practical deployment of file similarity techniques in cybersecurity applications.

3. Similarity Metrics

This section explores the evolution of file similarity techniques, starting from simple algorithms and identifying their shortcomings to motivate the development of more complex techniques.

3.1. Edit Distance

In general, edit distance is a measure of the dissimilarity of two strings. In the case of file similarity, this is a measure of dissimilarity between two-byte sequences. Several different types of edit distance have been proposed in the literature [16], and Hamming Distance, Levenshtein Distance and Longest-Common Subsequence are commonly used distance metrics that support different types of operations.

3.1.1. Hamming Distance

Hamming Distance compares two bit-streams or byte-streams by simply counting the number of positions where the two streams differ. The comparison of distance between two sequences of length N is $O(N)$, though it only applies to sequence pairs of equal length and is unable to detect insertions and deletions. Hamming distance is still employed widely owing to its simplicity, finding applications in detecting data corruption and errors in network transmissions.

3.1.2. Levenshtein Distance

Levenshtein distance measures the distance between two-byte sequences as the minimum number of insertions, deletions or substitutions to transform one sequence into the other. The distance between two sequences can be computed in $O(D \min(M, N))$ [17], where D is the edit distance, and M and N are the lengths of the two sequences. Because of its ability to handle insertions, updates and deletions, this algorithm is popular in spell checks and basic natural language processing applications like fuzzy string matching.

3.1.3. Longest Common Subsequence

Longest Common Subsequence measures the length of the longest pairing of characters that can be made between both strings so that the pairings respect the order of the letters. The distance is the number of unpaired characters

and can be computed in *O(MN),* where *M* and *N* are the lengths of the two sequences.

3.1.4. Search and Clustering Complexity

Given a target file of length *N*, when looking through a corpus of length *C*, the complexity of searching for the most similar files using linear-complexity edit distance algorithms is linear in the size of the corpus *O(CN)*. The runtime complexity of clustering is quadratic in the size of the corpus since every file must be compared against every other file to identify the appropriate clusters.

3.2. Jaccard Distance

The Jaccard Distance *J(A, B)* between two sets *A* and *B* is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Due to the small number of unique byte values (2^8 = 256), applying the Jaccard Distance to compare two sets of bytes is not effective. Most files are likely to have a large number of these byte values, and the Jaccard distance will be close to 1. In practice, Jaccard distance is applied to features extracted from files, like N-grams or the set of imports [18].

Unlike Edit Distance-based similarity detection, Jaccard distance computation between two files is linear in the size of the feature set, not the actual bit- or byte-sequences. Since extracted features are typically much smaller than the files themselves, this translates to much faster search and clustering algorithms in practice. However, the effectiveness of this approach depends on how well the extracted features encode the underlying file data. Moreover, Jaccard distance treats the sets of features as a “bag of words” and loses any ordering information.

3.2.1. MinHash

Even with the reduced feature set of Jaccard distance, computing set union and intersection can be computationally expensive for large files. MinHash [19] estimates the Jaccard distance of two sets, A and B, without explicitly computing their union and intersection. By applying multiple hash functions to each element of the feature set, MinHash constructs a signature for each file by generating a vector where each entry corresponds to the smallest hash value from a specific hash function. The *MinHash similarity* is computed as the fraction of hash functions for which the MinHash signatures of A and B match.

$$MinHash(A,B) = \frac{|Number\ of\ matching\ hash\ values|}{|Total\ number\ of\ hash\ functions|}$$

This metric is shown to be an unbiased estimator of the Jaccard distance. Although MinHash is not an improvement in terms of the accuracy of similarity computation, it offers significant efficiency gains in specific scenarios. It reduces the problem of comparing small fixed-size signatures, significantly reducing computational and storage costs, and hence scales better to larger datasets. The quality of MinHash’s approximation increases with the number of hash functions used.

4. Similarity Digests

Edit Distances and Jaccard Distance either operate on the full bytes of the input files or operate on a feature set, which requires complex feature engineering with unpredictable results and corpus sizes. Jaccard distances and MinHash are also more effective on text or structured metadata, which can be tokenized into meaningful chunks. These shortcomings in traditional similarity metrics motivated the engineering of similarity digests that worked natively on binary data.

4.1. SSDeep

ssdeep [20] was introduced as a technique to generate a feature set without needing to tokenize or transform the binary data into a set.

ssdeep uses Context-Triggered Piecewise Hashing (CTPH) to generate a fuzzy hash with the following algorithm

- Run a sliding window of fixed size over the byte stream, maintaining a rolling hash over the bytes in the sliding windows
- Emit variable-sized chunks based on the value of the rolling hash (boundary condition check)
- Apply a cryptographic hash function over all the generated chunks
- Base64-encode the least significant bits of the hash

Two ssdeep hashes can only be compared if their block sizes are equal or differ by a factor of 2x. Levenshtein distance is then used to measure the number of updates, insertions and deletions between the hashes, with different weights given to each type of operation. The resulting value is scaled to the range 0 – 100. The runtime complexity is linear in the size of the generated fuzzy hash. Although ssdeep is fast and accurate for hashes with compatible block sizes, many data are not comparable. Unlike Jaccard Distance, it is also sensitive to reordering, compression, padding and changes to encoding.

4.2. SDHash

The shortcomings of ssdeep and other fuzzy hashing techniques motivated the creation of sdhash [21]. While

ssdeep hashes all file chunks, sdhash chooses 64-byte sequences that are least likely to occur in other files by chance. Specifically, sdhash

1. Runs a sliding window of fixed size (64 bytes) over the byte stream to generate a feature set
2. Calculates the Shannon entropy of the extracted sub-sequences from the input sequence
3. Chooses a set with the lowest entropy
4. Hashes the features into a Bloom filter. If one Bloom filter fills up, another one is created.

The sequence of generated bloom filters is the generated hash. Two hashes are compared by averaging the maximum Hamming distance of each filter from the first digest to every filter in the second digest. Although sdhash outperforms ssdeep in terms of accuracy and robustness, it has some shortcomings. The final digest is between 2-3% of the input size, which is a significant proportion of the original bytes. Hence, searching across a corpus of files is still proportional to the size of the whole corpus.

4.3. TLSH and Telfhash

TrendMicro designed TLSH [14] as an attempt to address the shortcomings of ssdeep and sdhash, targeting robustness to reordering, predictable digest size, linear digest generation complexity, combinating evasiness, accounting for frequency of occurrence of features and generating an approximate metric from digest comparisons. TLSH samples the bytes of the input byte sequence to create a histogram of N-grams, finally generating a 35-byte digest. The steps in the algorithm are:

1. Run a sliding window of size 5 byte-by-byte over the input file
2. Extract unique 3-grams from each 5-byte window
3. Map each 3-gram to a bucket in a histogram and increment its count
4. Calculate *p25*, *p50*, *p75* quantiles over the array of bucket counts
5. Construct the 3-byte digest header as a combination of the input checksum, input length and the quantiles
6. Construct the 32-byte digest body as a function of the histogram values, emitting different bit pairs for different quantile thresholds

TLSH distance between two files is computed using two functions: one to calculate the distance between headers and the other, the distance between bodies. Distance == 0 means the two files are the same and increases as files become more dissimilar. TLSH has been shown to be more robust to changes in the input bytes than ssdeep or sdhash [23] and generates a fixed-size digest, but sdhash performs better at containment detection and in recognizing the same program when compiled in different ways [24].

The latter motivated the development of Telfhash, specifically the need to enhance similarity hashing specifically for ELF (Executable and Linkable Format) binaries, addressing the limitations of TLSH when applied to these files.

While TLSH is a robust general-purpose locality-sensitive hash, its reliance on global byte distribution statistics makes it less effective for binary executables, particularly those with structural changes like code injections or section rearrangements. telfhash improves upon TLSH by incorporating binary-specific features, such as focusing on code sections, symbols, and metadata that are more indicative of functional similarity in ELF binaries. This allows telfhash to better handle cases where binaries are modified for obfuscation, packed, or minimally altered.

4.4. Evaluation

Table 1, 2, and 3 show the results of evaluating the aforementioned techniques against several parameters: the hashing technique, output sizes in bytes, computational complexity of hashing and search, as well as their strengths and weaknesses.

Table 1. Summary of hashing techniques

k: Number of hash functions (where applicable), d: Digest size

Digest Type	Hashing Technique	Output
MinHash	Randomized hashing of sets	Fixed-size vector, depends on k: $O(k)$
ssdeep	Context-triggered piecewise hashing	Up to 128 characters (Base64 string)
sdhash	Statistical feature extraction	Binary digest, variable size (d)
TLSH	Locality-sensitive hashing with entropy	Fixed-size hash (35 bytes)

Table 2. Hashing complexity and practicality

k: Number of hash functions (where applicable), n: Input length

Digest Type	Hashing Complexity	Strengths	Weaknesses
MinHash	O(k·n)	Efficient; scalable	Approximation may lose fine details
ssdeep	O(n)	Fast and widely used; easy integration	Sensitive to reordering and padding
sdhash	O(n·log(n))	Robust to minor modifications	High storage and computational costs
TLSH	O(n)	Robust, predictable size, scalable	Less effective for binary-specific cases

Table 3. Evaluation of similarity techniques

d: Digest size

Digest Type	Similarity technique	Similarity complexity
MinHash	Jaccard similarity	O(d)
ssdeep	Fuzzy matching via edit distance	O(d^2)
sdhash	Hamming distance	O(d)
TLSH	Distance metric in TLSH space	O(d)

5. Similarity Search and Clustering

Practical applications of similarity digests usually search for the files most similar to a target file from a reference list of *N* files like NSRL or MalwareBazaar. The naive approach to solve this problem using the similarity digests from the previous sections would be to perform a brute-force search by comparing the target file digest to each of the *N* files' digests in the reference list. As expected, this approach is to time- and resource-intensive for large datasets.

Systems that are designed to search for exact matches, like against cryptographic hashes, build reverse indexes of the hashes and map them to files in the reference list using data structures like skip lists [25] or B-trees. When the data does not fit in memory, on-disk storage structures like log-structured merge [26] trees are used to store the reverse indexes. The search complexity over N files in these cases is *O(log(N))*.

Some of the proposed approaches to improve similarity digest search over a large corpus of files are discussed by presenting relevant in-memory and on-disk data structures and how they apply to the similarity digests previously discussed. This section also discusses the clustering properties and complexity of the proposed data structures.

Figure 1 shows the general flow of ingesting a corpus of N files, converting them into digests and storing them efficiently for search, while Figure 2 shows the flow of data when a user queries the same system for files similar to a target file. Table 4 evaluates the different similarity search and clustering approaches based on their runtime complexities

4.1. Fast Forensic Similarity Search (F2S2)

F2S2 [27] proposes reducing the search space for digest comparisons with an indexing structure based on the n-grams extracted from an input sequence. A reverse index is then constructed, mapping each n-gram segment to all the digests of the input bytes that contain it. When searching for similar digests across a corpus of N files, n-gram segments are extracted from the target file. For each n-gram, all matching digests are retrieved from the reverse index constructed earlier. This set of matching digests is the reduced search space against which similarity comparisons are executed.

The authors of F2S2 use ssdeep as the digest, but the general algorithm and indexing structure generalizes to any digest technique that operates on segments of its input files. MinHash uses hashed feature sets, making it a good candidate for F2S2's indexing technique. While TLSH also extracts n-grams from its input, it relies on histogram generation to account for entropy and reduce similarity complexity, and thus, the applicability of F2S2 to reduce TLSH search space is not obvious. F2S2 does not work well with Bloom filter-based similarity techniques like sdhash since they cannot be indexed.

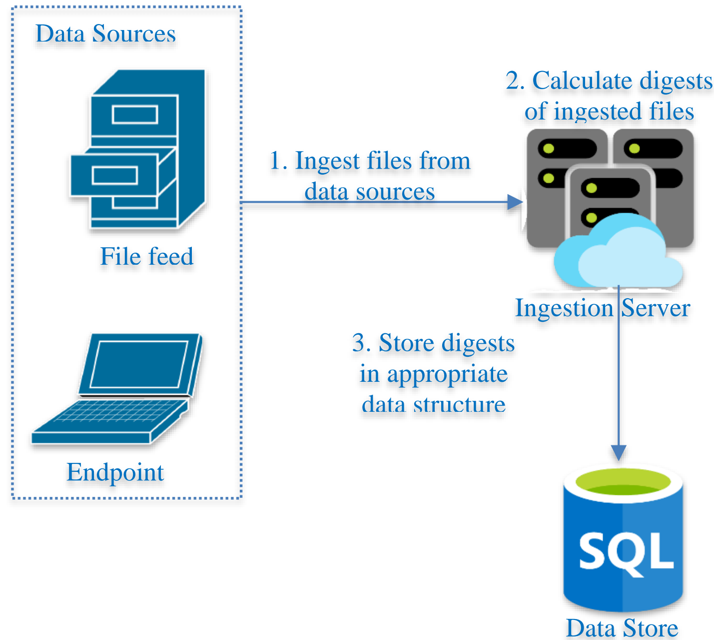

Fig. 1 General flow of ingesting a corpus of N files: (1) Files are ingested from feeds or endpoints (2) The ingestion application calculates the digests for the files and (3) Stores the digests in an efficient data structure for later retrieval

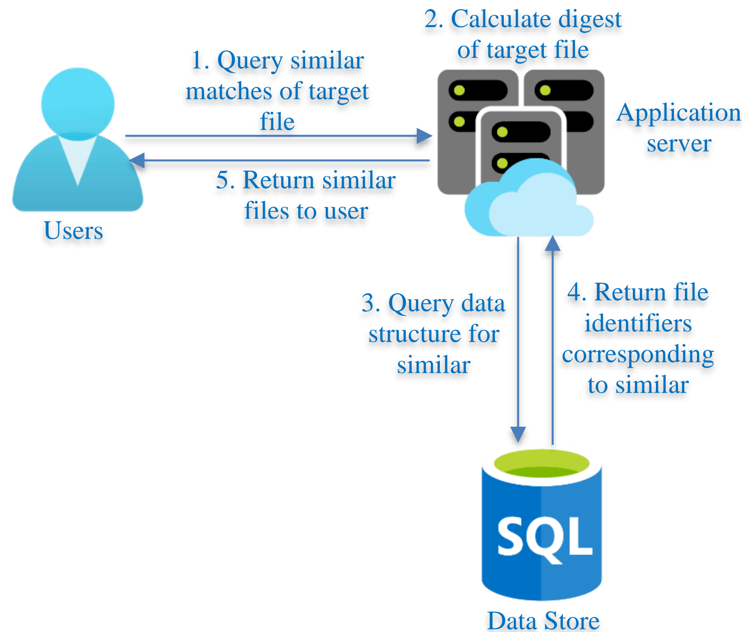

Fig. 2 General flow of querying for files similar to a target file: (1) User queries the corpus for files similar to a target file. (2) The application server calculates the digest of the target file. (3) The application server queries the efficient data structure for similar digests, (4) retrieves the files corresponding to those digests, and (5) returns them to the user

In general, the efficacy of F2S2 is only as good as the chosen digest technique, and the shortcomings of MinHash and ssdeep have been discussed earlier.

4.2. Bloom Filter Search

Breitinger et al. [28] propose a hierarchical bloom filter tree structure using a divide-and-conquer paradigm to reduce similarity digest search complexity to *O(logN)*. Bloom filters are organized into a tree structure, with higher-level nodes representing aggregated subsets of the corpus and lower-level nodes containing finer-grained representations. To conduct a similarity search, a query digest is first matched against the root node to determine potential matches, and the search then proceeds down the tree, narrowing the candidate set at each level. This recursive refinement reduces the number of comparisons

needed compared to brute force comparisons. This approach is well suited to sdhash, which generates similarity digests as sequences of Bloom filters. Researchers have also proposed extensions of this approach using Cuckoo filters [29].

4.3. Hierarchical Agglomerative Clustering

The HAC-T algorithm [15] employs a tree-based approach to enable TLSH similarity searches across a corpus of files efficiently. It constructs the tree by recursively splitting the dataset using a selected pivot – which is a random item from the dataset – and a threshold, which partitions the data into two subsets: items closer to the pivot and items farther away from the pivot. The threshold is chosen so that the two subsets are roughly equal in size, resulting in a balanced tree. The recursive splitting continues until a predefined stopping condition is met, such as a minimum subset size or an insignificant threshold value.

During a search, the tree is traversed based on the distance of the query to pivots, resulting in a search complexity of O(log N) on a corpus of N digests.

Table 4. Evaluation of similarity and clustering techniques

N: File corpus size

Data Structure	Compatible Digests	Search Complexity	Clustering Complexity
Reverse Index (F2S2)	MinHash, ssdeep	$O(N)$	$O(N^2)$
Bloom Filter	sdhash	$O(\log N)$	$O(N \log N)$
HAC-T	TLSH	$O(\log N)$	$O(N \log N)$

5. Future Work

From the discussions and analysis earlier in this paper, several insights emerge regarding the evolution and future of file similarity techniques. Below are key recommendations for advancing this field.

5.1. Multimodal Approach

The analysis shows that no single similarity digest technique is universally optimal. Each method has its strengths and weaknesses [30]. Consequently, security researchers should adopt a toolkit approach that combines multiple digest techniques to ensure robustness across diverse use cases.

5.2. Addressing Clustering Limitations

Clustering methods for similarity digests exhibit varying levels of performance and precision. Existing methods, such as HAC-T for TLSH or Bloom filter trees for sdhash, either struggle to generalize well across different datasets or are challenging to implement in practice. Future work should focus on developing more versatile clustering frameworks that strike a balance between scalability, ease of use, and precision. Hybrid clustering methods that incorporate aspects of density-based or centroid-based clustering with domain-specific optimizations could address these gaps.

5.3. Machine Learning for Digest Generation

Techniques based on deep neural networks, like Siamese networks [36], can be used to generate embeddings that capture similarities between two file feature sets. One of the challenges faced by these methods is the lack of a widely used benchmark dataset [37], forcing researchers to rely on custom datasets [35].

5.4. Leveraging Vector Databases for Scalability

Recent advancements in vector database technology offer a significant opportunity for scaling file similarity search and clustering by constructing indexes that allow for efficient approximate nearest neighbor (ANN) search. Systems like Milvus [31], Pinecone [32], and Weaviate [22] provide highly scalable and efficient infrastructures for managing high-dimensional vector data. These technologies have been proven to scale to billions of files in production machine learning environments, and show potential in being used for similarity digest search.

The embeddings generated by neural networks can also be seamlessly integrated with vector databases, offering compact and efficient storage solutions for search and clustering. For its clustering technique, HAC-T calculates the mean of multiple TLSH digests by preprocessing them into 70-dimensional Euclidean coordinates with ASCII values.

6. Conclusion

This paper provides a detailed discussion of popular file similarity techniques, discussing the advantages that fuzzy digests and locality-sensitive hashes provide over traditional cryptographic methods and static signatures. Additionally, the paper explored how modern advancements such as embeddings and vector databases can enhance the scalability and efficiency of these methods for malware detection and file similarity search. It also highlights the importance of a diverse toolkit in the security researcher's belt, emphasizing the combination of fuzzy digest techniques with cutting-edge machine learning and database technologies to achieve robust, real-time detection across massive datasets. The discussed file similarity techniques strengthen the cybersecurity field through early detection of malware variants based on behavioral similarities. The evaluation of search and clustering efficiency also allows organizations to leverage a layered detection approach while building a comprehensive threat detection database.

References

[1] Managing Risks & Costs at the Edge, Ponemon Institute, Report, pp. 1-49, 2022. [Online]. Available: https://adaptiva.com/hubfs/Reports/Adaptiva-Ponemon-Report-2022.pdf

[2] Cem Dilmegani, Endpoint Security Statistics in 2025, AI Multiple Research, 2024. [Online]. Available: https://research.aimultiple.com/endpoint-security-statistics

[3] Darren Quick, and Kim-Kwang Raymond Choo, "Impacts of the Increasing Volume of Digital Forensic Data: A Survey and Future Research Challenges," *Digital Investigation*, vol. 11, no. 4, pp. 273-294, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[4] National Software Reference Library, NIST. [Online]. Available: http://www.nsrl.nist.gov/

[5] National Vulnerability Database, NIST. [Online]. Available: http://nvd.nist.gov/

[6] Malware Bazaar. [Online]. Available: https://bazaar.abuse.ch/export/

[7] Reyadh Hazim Mahdi, and Hafedh Trabelsi, "Detection of Malware by Using YARA Rules," *2024 21st International Multi-Conference on Systems, Signals & Devices*, Erbil, Iraq, pp. 1-8, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[8] Frank Breitinger et al., "Using Approximate Matching to Reduce the Volume of Digital Data," *Advances in Digital Forensics X: 10th IFIP WG 11.9 International Conference*, Vienna, Austria, pp. 149-163, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[9] Jianguo Wang et al., "Milvus: A Purpose-Built Vector Data Management System," *Proceedings of the 2021 International Conference on Management of Data*, Virtual Event China, pp. 2614-2627, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[10] Build knowledgeable AI, Pinecone. [Online]. Available: https://www.pinecone.io

[11] The AI-Native Database for a New Generation of Software, Weaviate. [Online]. Available: https://weaviate.io

[12] Yu A. Malkov, and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824-836, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[13] Annoy. [Online]. Available: https://github.com/spotify/annoy

[14] Jonathan Oliver, Chun Cheng, and Yanggui Chen, "TLSH - A Locality Sensitive Hash," *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, Sydney, NSW, Australia, pp. 7-13,2013. [CrossRef] [Google Scholar] [Publisher Link]

[15] Jonathan Oliver, Muqeet Ali, and Josiah Hagen, "HAC-T and Fast Search for Similarity in Security," *2020 International Conference on Omni-layer Intelligent Systems*, Barcelona, Spain, pp. 1-7, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[16] Gonzalo Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001. https://doi.org/10.1145/375360.375365[CrossRef] [Google Scholar] [Publisher Link]

[17] Esko Ukkonen, "Algorithms for Approximate String Matching," *Information and Control*, vol. 64, no. 1-3, pp. 100-118, 1985. [CrossRef] [Google Scholar] [Publisher Link]

[18] Joshua Saxe, and Hillary Sanders, *Malware Data Science: Attack Detection and Attribution*, No Starch Press, pp. 1-272, 2018. [Google Scholar] [Publisher Link]

[19] A.Z. Broder, "On the Resemblance and Containment of Documents," *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, Salerno, Italy, pp. 21-29,1997. [CrossRef] [Google Scholar] [Publisher Link]

[20] Jesse Kornblum, "Identifying Almost Identical Files Using Context Triggered Piecewise Hashing," *Digital Investigation*, vol. 3, pp. 91-97, 2006. [CrossRef] [Google Scholar] [Publisher Link]

[21] Vassil Roussev, "Data Fingerprinting with Similarity Digests," *Advances in Digital Forensics VI: Sixth IFIP WG 11.9 International Conference on Digital Forensics*, Hong Kong, China, pp. 207-226, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[22] Dongkwan Kim et al., "Revisiting BCSA Using Interpretable Feature Engineering and Lessons Learned," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1661-1682, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[23] Jonathan Oliver, Scott Forman, and Chun Cheng, "Using Randomization to Attack Similarity Digests," *Applications and Techniques in Information Security: 5th International Conference*, Melbourne, Australia, pp. 199-210, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[24] Thomas Göbel et al., "FRASHER – A Framework for Automated Evaluation of Similarity Hashing," *Forensic Science International: Digital Investigation*, vol. 42, pp. 1-13, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[25] William Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668-676, 1990. [CrossRef] [Google Scholar] [Publisher Link]

[26] Patrick O'Neil et al., "The Log-structured Merge-Tree (LSM-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996. [CrossRef] [Google Scholar] [Publisher Link]

[27] Christian Winter, Markus Schneider, and York Yannikos, "F2S2: Fast Forensic Similarity Search Through Indexing Piecewise Hash Signatures," *Digital Investigation*, vol. 10, no. 4, pp. 361–371, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[28] Frank Breitinger, Christian Rathgeb, and Harald Baier, "An Efficient Similarity Digests Database Lookup - A Logarithmic Divide & Conquer Approach," *Journal of Digital Forensics, Security and Law*, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[29] Bin Fan et al., "Cuckoo Filter: Practically Better Than Bloom," *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*, Sydney Australia, pp. 75-88, 2014. [CrossRef] [Google Scholar] [Publisher Link]
[30] Irfan Ul Haq, and Juan Caballero, "A Survey of Binary Code Similarity," *ACM Computing Surveys*, vol. 54, no. 3, pp. 1-38, 2021. [CrossRef] [Google Scholar] [Publisher Link]
[31] Jiang Du et al., "A Review of Deep Learning-Based Binary Code Similarity Analysis," *Electronics*, vol. 12, no. 22, pp. 1-18, 2023. [CrossRef] [Google Scholar] [Publisher Link]
[32] Abhiraj Malhotra, "Single-Shot Image Recognition Using Siamese Neural Networks," *2023 3rd International Conference on Advance Computing and Innovative Technologies in Engineering*, Greater Noida, India, pp. 2550-2553, 2023. [CrossRef] [Google Scholar] [Publisher Link]