The Next Generation of Malware Defense using Deep Hashing and Neural Embeddings

Uma Kannan[1], Rajendran Swamidurai[1*]

[1]Department of Mathematics and Computer Science, Alabama State University, Montgomery, AL, USA

***Corresponding Author**

DOI: https://dx.doi.org/10.51244/IJRSI.2026.1308000003

Received: 14 August 2026; Accepted: 19 August 2026; Published: 25 August 2026

ABSTRACT

The proliferation of polymorphic and metamorphic malware has largely rendered traditional cryptographic signature-based detection ineffective, driving the adoption of similarity-based approaches. This survey systematically examines the evolution of binary similarity detection, tracing the trajectory from classical fuzzy hashing techniques—including ssdeep, sdhash, and TLSH—to contemporary deep representation learning architectures. We analyze state-of-the-art deep hashing methodologies, covering image-based representations via Convolutional Neural Networks (CNNs), structural control-flow graph modeling via Graph Neural Networks (GNNs), and assembly-level semantic analysis using Transformer architectures such as MalBERT and KEENHash. Furthermore, we critically assess the adversarial robustness of these embedding spaces across feature-space and problem-space threat models. By synthesizing recent theoretical developments and empirical benchmarks, this paper provides a unified taxonomy and outlines key challenges for resilient malware lineage tracking and zero-day threat detection.

Keywords: Malware Defense / Detection; Deep Hashing; Neural Embeddings; Binary Similarity Detection; Fuzzy Hashing; Convolutional Neural Networks (CNNs); Graph Neural Networks (GNNs); Control-Flow Graphs (CFGs); Transformers; Semantic Analysis; Adversarial Robustness; Polymorphic and Metamorphic Malware.

INTRODUCTION

The contemporary digital landscape is characterized by an escalating arms race between malware authors and defense mechanisms. Malicious software, or malware, has evolved from static, hand-coded executables into complex software ecosystems capable of autonomous mutation.[1] The sheer volume of unique malware samples emerging daily renders manual analysis impossible, necessitating automated systems capable of processing millions of files to identify threats. [2]

Historically, the cornerstone of malware detection was the cryptographic hash (e.g., MD5, SHA-256). These algorithms are designed with the avalanche effect as a primary property: a single bit flip in the input must result in a statistically independent, pseudo-random change in the output digest. [3] While this property is ideal for data integrity verification, it is catastrophic for malware detection. Adversaries exploit this fragility by applying trivial modifications—such as appending null bytes, changing timestamps, or reordering independent instructions—to generate a distinct cryptographic hash for every infection instance, a technique known as polymorphism. [2]

To counter this, the industry has adopted similarity detection, a methodology that seeks to quantify the resemblance between files rather than their equality. This field has matured through two distinct generations. The first generation, *Fuzzy Hashing*, relies on algorithmic heuristics to break files into chunks and compute digests that remain stable under minor modifications. [4] Tools like ssdeep and TLSH became industry standards for clustering malware families and identifying variants.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

However, as obfuscation techniques have advanced to include control flow flattening, dynamic API resolution, and cross-architecture compilation, the syntactic focus of fuzzy hashing has proven insufficient. The second and current generation, *Deep Hash Embedding*, leverages the representational power of Deep Neural Networks (DNNs) to map malware into continuous vector spaces. [5] By learning semantic features from the code—treating it as an image, a graph, or a language—these models can detect functional similarities even when the binary representation is drastically altered.

This report provides a detailed technical dissection of these technologies. We begin by establishing the theoretical and algorithmic foundations of fuzzy hashing in Section 2. Section 3 explores the visualization of malware and image-based deep hashing. Section 4 delves into graph-based representations using GNNs, while Section 5 analyzes the application of Transformer models to assembly code. Section 6 critically assesses the adversarial vulnerability of these systems, and Section 7 concludes with a discussion on scalability and future architectural trends.

FOUNDATIONS OF SIMILARITY HASHING

Fuzzy hashing, or Similarity Preserving Hashing (SPH), represents the first algorithmic attempt to solve the approximate matching problem in digital forensics. Unlike cryptographic hashes, SPH algorithms are designed to maximize collisions for similar inputs, mapping syntactically related files to proximal points in the digest space.

Context-Triggered Piecewise Hashing (CTPH)

One of the earliest and most widely deployed fuzzy hashing tools is *ssdeep*, which implements the Context-Triggered Piecewise Hashing (CTPH) algorithm. [4] Developed to identify spam emails, it was adapted for binary analysis to detect files that have been modified by appending or inserting data.

Algorithmic Architecture

The core innovation of ssdeep is its departure from fixed-size blocking. Traditional block-based hashing divides a file into chunks of N bytes (e.g., 4KB) and hashes each chunk. If a single byte is inserted at the beginning of the file, every subsequent block boundary shifts, changing every hash. ssdeep utilizes a *rolling hash* to determine block boundaries dynamically based on the file's content. [6]

- **Rolling State:** The algorithm maintains a hash state calculated over a sliding window of fixed size (typically 7 bytes). [7] As the window advances byte-by-byte, the hash is updated cheaply.
- **Trigger Points:** The algorithm monitors the rolling hash value. When this value satisfies a /specific condition (e.g., $H \bmod N = 0$), a *trigger* is fired. This marks the end of a chunk. Because the trigger depends on the *context* (the 7 bytes in the window) rather than the absolute offset, the block boundaries effectively resynchronize after an insertion, limiting the corruption of the signature to only the chunks containing the modification.[6]
- **Digest Generation:** Once a chunk is defined, its contents are hashed using a non-cryptographic algorithm, typically the Fowler-Noll-Vo (FNV) hash. This produces a 6-bit value, which is mapped to a Base64 character. [8]
- **Output Structure:** The final ssdeep signature consists of a block size, and two string sequences. The first sequence corresponds to the rolling hash at the chosen block size, and the second at double that block size. This dual-signature structure allows for comparison between files of significantly different lengths. [6]

Vulnerabilities and Limitations

Despite its speed and ubiquity, ssdeep suffers from inherent structural weaknesses that sophisticated malware authors exploit:

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

- **Padding Attacks:** The algorithm selects the initial block size based on the total file size. By appending a large volume of null bytes or random noise (padding) to a malware sample, an attacker can force the algorithm to select a larger block size. This alters the granularity of the chunking process, resulting in a completely different signature for the same malicious payload. [9]

- **Order Dependence:** The rolling hash is strictly sequential. If functions within the binary are reordered (a common technique in polymorphic engines), the sequence of triggers and the resulting string of characters changes completely, yielding a low similarity score.[10]

- **Minimum Commonality:** Research has shown that ssdeep requires a sequence of at least 7 consecutive common features to trigger a match. Attackers can defeat this by introducing small mutations at regular intervals (every few kilobytes), preventing any single chunk from matching. [11]

Statistically-Improbable Features (SIF)

To address the fragility of sequential hashing, *sdhash* introduces a feature-centric approach. Instead of hashing the entire file in chunks, sdhash attempts to identify and hash only the most unique or statistically improbable elements of the file.[12]

Entropy-Based Feature Selection

The sdhash algorithm relies on Shannon entropy as a selection heuristic. It scans the file and calculates the entropy for sequences of 64 bytes. It identifies features that have the lowest empirical probability of occurrence—essentially, the most distinctive parts of the code.[13] This is based on the insight that common code libraries or padding (high probability, low information) are less useful for identification than unique code logic (low probability, high information).

Bloom Filter Representation

Selected features are hashed using SHA-1. These hashes are then inserted into a *Bloom filter*, a space-efficient probabilistic data structure used to test set membership.[14]

- **Digest Structure:** The final sdhash digest is a sequence of 256-byte Bloom filters. Each filter stores approximately 160 features (or 192 in block mode).[14]

- **Comparison:** Similarity between two files is computed by estimating the intersection of their feature sets. This is done by comparing the bits set in their respective Bloom filters. A score of 0 to 100 is generated, representing the confidence that the files share common content.[14]

Comparative Strengths and Weaknesses

The primary advantage of sdhash over ssdeep is its robustness to reordering and alignment. Since features are selected based on their content (entropy) rather than their position, shuffling code blocks does not prevent feature selection.[13] Furthermore, sdhash excels at fragment detection—identifying if a piece of malware is embedded inside another file (e.g., a dropper).[15] However, the method is computationally expensive due to the entropy calculation and SHA-1 operations. It is also vulnerable to attacks that manipulate the entropy profile of the file (e.g., by interleaving high-entropy noise) to suppress feature selection, effectively blinding the algorithm.[11]

Locality Sensitive Hashing (LSH)

Trend Micro Locality Sensitive Hash (TLSH) represents a significant evolution in fuzzy hashing, explicitly designed to address the high false-positive rates and scalability issues of previous methods.[16]

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

N-Gram Analysis and Bucketization

TLSH processes the input file using a sliding window of 5 bytes. From this window, it generates triplet trigrams. To improve efficiency and distribution, it does not use all combinations; rather, it selects 6 specifically chosen triplets out of the possible 10.[16] These trigrams are mapped using a Pearson hash into an array of 128 buckets. Each bucket accumulates a count of how many times its corresponding trigrams appeared in the file.[17]

Quartile-Based Digest Construction

The defining characteristic of TLSH is its quantization method. Once the bucket counts are populated, the algorithm calculates the quartiles (Q_1, Q_2, Q_3) of the count distribution.

- **Bit Encoding:** The digest body is constructed by examining each bucket's value relative to these quartiles. For example, a bucket might be assigned a 2-bit code:
  - 00 if count $\leq Q_1$
  - 01 if $Q_1 <$ count $\leq Q_2$
  - 10 if $Q_2 <$ count $\leq Q_3$
  - 11 if count $> Q_3$
    This compresses the complex statistical distribution of byte patterns into a compact binary string.[17]
- **Header:** The digest also includes a header containing a checksum and the logarithm of the file length, aiding in rapid pre-filtering.[17]

Performance Metrics

Empirical benchmarks consistently rank TLSH as superior to ssdeep and sdhash for malware clustering tasks.[15]

- **Distance Metric:** TLSH provides a smooth distance score (typically 0-300+), where lower is more similar. This granularity allows analysts to set precise thresholds for same family vs. variant.[18]
- **Robustness:** The statistical aggregation over 128 buckets makes TLSH highly resistant to local modifications. An attacker must significantly alter the byte distribution of the entire file to change the quartile thresholds, which is far harder than breaking a rolling hash sequence.[16]

Set-Based Similarity

While ssdeep and TLSH operate on raw bytes, *MinHash* is a technique adapted from document clustering (e.g., detecting duplicate web pages) that treats malware as a set of abstract features (e.g., opcodes, strings, API imports).[19]

The Jaccard Index Estimation

The goal of MinHash is to estimate the Jaccard Similarity ($J(A,B) = \frac{|A \cap B|}{|A \cup B|}$) between two sets without storing the sets themselves.

- **Permutation:** The algorithm applies *K* different hash functions (or a single hash function with *K* different seeds/permutations) to every element in the set.
- **Min-Value Selection:** For each hash function, the minimum hash value observed across all elements is stored. This vector of *K* minimum values constitutes the MinHash signature.[n20]

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

- **Probability Property:** The fundamental theorem of MinHash states that the probability that the minimum hash value of two sets is the same is exactly equal to their Jaccard similarity.[21]

Application in Malware

MinHash is particularly powerful for large-scale databases like VirusTotal. By converting a malware sample into a set of 5-grams (sequences of 5 opcodes) and computing a MinHash signature, systems can perform nearest-neighbor search over millions of samples in sub-linear time.[21] It bridges the gap between byte-level hashing and semantic feature extraction.

Comparative Analysis of Fuzzy Hashing Algorithms

Table 1 summarizes the technical distinctions and operational capabilities of the discussed algorithms.

Table 1: Technical Comparison of Fuzzy Hashing Algorithms

Algorithm	Core Mechanism	Input Scope	Digest Structure	Strengths	Vulnerabilities
ssdeep [4]	Context-Triggered Piecewise Hashing (CTPH)	Sequential Rolling Window (7 bytes)	Variable length string (Base64)	Fast generation; excellent for identifying appended data; industry standard.	Highly sensitive to reordering; vulnerable to padding attacks manipulating block size [9]; order dependent.
sdhash [13]	Statistically-Improbable Features (SIF)	Entropy-based (64-byte chunks)	Sequence of Bloom Filters	Robust to block reordering; effective at detecting embedded fragments. [15]	Computationally expensive; vulnerable to entropy manipulation attacks [11]; complex comparison.
TLSH [10]	Locality Sensitive Hashing (LSH)	N-gram Statistics (Triplet Trigrams)	Fixed length (70 hex chars), Quartile-encoded	High accuracy; smooth distance metric; resistant to collision attacks; suitable for clustering.	Less effective on very small files or files with low byte variance; sensitive to compression.
MinHash [20]	Set Similarity (Jaccard Estimation)	Abstract Features (Opcodes, Strings)	Vector of Hash Integers	Extremely scalable search (LSH); mathematically grounded; feature-agnostic.	Requires preprocessing/feature extraction; ignores structure (bag-of-words approach).

VISUALIZING MALWARE

As malware obfuscation moved beyond simple byte manipulation to algorithmic transformations, researchers sought representations that could capture *texture* and *structure* simultaneously. This led to the visualization of binaries as images, enabling the application of Computer Vision techniques to malware analysis.

The Binary-to-Image Mapping

The transformation of a binary executable into an image is based on the observation that code, data, and resources exhibit distinct entropy patterns.[19]

- **Mapping:** The file is read as a vector of 8-bit unsigned integers. These integers are mapped directly to pixel intensity values (0-255).

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

- **Geometry:** The pixels are arranged in a 2D matrix. The width of the image is typically fixed (e.g., 256 or 1024 pixels) based on the file size, while the height varies.[22]

- **Visual Textures:**

    - **Code Sections (.text):** Exhibit fine-grained, structured patterns due to the repetitive nature of x86 opcodes.

    - **Data Sections (.data):** Vary widely; strings appear as smooth textures, while compressed data appears as high-entropy noise.

    - **Padding:** Appears as solid blocks of black (0x00) or white (0xFF).

Deep Hashing with Convolutional Neural Networks (CNNs)

Once the binary is an image, standard CNN architectures (e.g., ResNet-50, VGG-16) can be deployed to extract features. This process is often termed *Deep Hashing* when the objective is to learn a compact binary code rather than a classification label.[23]

Architecture

The pipeline typically involves:

1. **Feature Extraction:** The image is passed through convolutional layers to capture local patterns (edges, textures) and pooling layers to aggregate them into high-level semantic features.[19]

2. **Hash Layer:** A fully connected layer maps the high-dimensional features to a lower-dimensional latent space (e.g., 64 or 128 bits).

3. **Quantization:** A sign function or sigmoid activation is used to binarize the output, producing a Deep Hash.

4. **Optimization:** The network is trained using loss functions that preserve similarity, such as *Triplet Loss* or *Contrastive Loss*, ensuring that variants of the same malware family map to similar hashes.[23]

Self-Supervised Learning

A major bottleneck in deep learning for malware is the scarcity of labeled data. **MalSSL** (Malware Self-Supervised Learning) addresses this by training on unlabeled binaries using pretext tasks.[22]

- **Pretext Tasks:** The model might be asked to predict if an image section has been rotated, obscured, or shuffled. By solving these tasks, the model learns robust feature representations of the binary structure without human annotation.

- **Performance:** After pre-training, the model is fine-tuned on a smaller labeled dataset. MalSSL has demonstrated classification accuracies of 98.4% on the MalImg dataset, significantly outperforming supervised methods trained on limited data.[22]

GRAPH-BASED REPRESENTATIONS

While image-based methods capture texture, they ignore the control flow logic of the program. A visual representation might change if code blocks are moved, but the execution graph remains invariant. This motivates the use of *Control Flow Graphs (CFGs)* and *Graph Neural Networks (GNNs).*

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

The Graph Construction Pipeline

Constructing a graph representation involves reverse engineering the binary to recover its logic.[24]

1. **Disassembly:** Tools like IDA Pro or Ghidra disassemble the binary into assembly instructions.
2. **Basic Block Identification:** Instructions are grouped into basic blocks—sequences of code with a single entry and exit point (no internal jumps).
3. **Edge Formation:** Directed edges are created between blocks to represent control flow transfers (jumps, branches, calls, returns).
4. **Attribute Generation:** Each node (block) is assigned a feature vector. This can include statistical features (instruction counts) or learned embeddings (using Word2Vec on the instructions).[25]

Graph Embedding Architecture

Analyzing raw graphs is computationally prohibitive (the graph isomorphism problem is NP-intermediate). Therefore, we must embed the graph into a vector space.

Graph Neural Networks (GNNs)

GNNs generalize neural networks to graph structured data. They employ a *Message Passing* mechanism:

- **Aggregation:** For every node v, the network aggregates feature vectors from its neighbors $u \in N(v)$.
- **Update:** The node updates its own state based on this aggregated message and its previous state.
- **Readout:** After k layers (hops), a global readout function (e.g., SumPooling or Attention Pooling) combines all node states into a single graph embedding vector h_G.[26]

Gemini and GeminiNet

Gemini is a pioneering architecture in this domain. It utilizes a *Siamese Network* composed of two identical GNNs sharing weights.[27]

- **Input:** Two CFGs (one known malware, one suspect).
- **Process:** The GNNs independently compute embeddings for both graphs.
- **Objective:** The network calculates the cosine similarity between the embeddings. It is trained to minimize the distance for pairs from the same family and maximize it for different families.[28]

GeminiNet extends this by integrating *Function Call Graphs (FCGs)* and *Inter-procedural Control Graphs (PCGs)*. It employs an *adaptive gating mechanism* to fuse the embeddings from these different graph views, weighing them dynamically based on their relevance. It also incorporates joint node features, combining Local Degree Profiles with Shannon Entropy to enrich the node representation.[29]

SEMANTIC ANALYSIS

The analogy between assembly code and natural language is profound. Both consist of sequences of tokens (opcodes/words) that follow a grammar (syntax) to convey meaning (semantics). This has led to the adaptation of NLP models for malware analysis.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

Self-Attentive Function Embeddings

SAFE (Self-Attentive Function Embeddings) addresses the problem of function-level similarity. Malware authors frequently reuse functions from open-source libraries or previous malware versions. Identifying these cloned functions is critical for attribution.[30]

- **Architecture:** Unlike Recurrent Neural Networks (RNNs) which process data sequentially and struggle with long-range dependencies, SAFE utilizes a *Self-Attention Network.*
- **Process:**
    1. **i2v (Instruction-to-Vector):** A Bidirectional RNN (BiRNN) converts individual assembly instructions into vectors, capturing the syntax of the instruction (e.g., mov eax, ebx).[31]
    2. **Self-Attention:** The model computes attention weights for each instruction vector, identifying which instructions contribute most to the function's semantic meaning.
    3. **Embedding:** The weighted vectors are summed to produce a fixed-length embedding for the entire function.[32]
- **Impact:** SAFE enables efficient 1-to-N search, allowing analysts to query a function against a database of millions in milliseconds, robust to compiler optimizations and instruction reordering.[30]

Transformer-Based Model

The Transformer architecture, with its multi-head attention mechanism, has revolutionized NLP and is now reshaping malware analysis.

Contextual Understanding (MalBERT)

MalBERT treats the sequence of opcodes as a sentence and the function as a document.

- **Tokenization:** Standard NLP tokenizers (like WordPiece) are ill-suited for assembly. *MalBERTv2* introduces *code-aware tokenization* that respects the structure of assembly (separating prefixes, opcodes, and operands).[33]
- **Pre-training (MLM):** The model is pre-trained using *Masked Language Modeling /(MLM)*. Random opcodes are masked (e.g., PUSH), and the model must predict the missing token based on the context. This forces the model to learn the *grammar* of valid execution flows.[33]
- **Fine-Tuning:** For classification, a dense classification layer is added on top of the token embedding. The model is fine-tuned on labeled malware datasets.[33]
- **Efficiency:** Some implementations utilize *DistilBERT*, a distilled version of BERT that retains 95% of performance while being 40% smaller and 60% faster, making it feasible for real-time detection.[34]

Bridging LLMs and Hashing (KEENHash)

A cutting-edge development in 2025 is *KEENHash*, which seeks to combine the semantic depth of Large Language Models (LLMs) with the retrieval speed of hashing.[5]

- **Workflow:**
    1. **LLM Embedding:** Functions are fed into a pre-trained LLM to generate high-fidelity embeddings.
    2. **Compression:** These embeddings are compressed using *K-Means clustering* and *Feature Hashing*.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

3. **Hash Generation:** The continuous embeddings are discretized into a fixed-length *program-level* hash.

- **Results:** This approach allows for massive scalability in Binary Code Similarity Analysis (BCSA). KEENHash is reported to be *215 times faster* than traditional function-matching tools while outperforming them in detection accuracy by over 23%.[5]

Cross-Architecture Translation

A persistent challenge is detecting malware across different Instruction Set Architectures (ISAs). A model trained on x86 malware typically fails on ARM malware. *FlowMalTrans* addresses this using *Normalizing Flows (NFs)*.[35]

- **Mechanism:** It learns a mapping between the embedding spaces of different ISAs. It translates the latent representation of an x86 binary into the latent space of an ARM binary.

- **Unsupervised Training:** Crucially, it uses unsupervised objectives like *Denoising Auto-Encoding (DAE)* and *Back Translation*, allowing it to be trained without paired samples (i.e., it doesn't need the *same* malware compiled for both architectures).[35] This effectively allows defenders to leverage their massive x86 datasets to protect IoT ecosystems.

ADVERSARIAL ROBUSTNESS IN EMBEDDING SPACES

The transition to Deep Learning introduces a new vector of attack: *Adversarial Examples (AEs)*. Just as adding noise to an image can fool a classifier, modifying a binary can fool a malware detector.

Feature Space vs. Problem Space

Understanding adversarial malware requires distinguishing between two domains [36]:

- **Feature Space:** The mathematical vector representation of the malware. Attacking this space is mathematically simple (using gradient descent to find the optimal perturbation vector δ).

- **Problem Space:** The actual binary file on disk. Mapping a perturbation δ back to valid bytes that form a functional executable is the *Inverse Feature Mapping* problem. It is extremely difficult because binary formats (PE, ELF) are brittle; random byte changes usually crash the program.

Gradient-Based Attacks on Binaries

Despite these constraints, researchers have successfully adapted gradient methods for *Problem Space* attacks.

The Append Attack

This is the most common and effective attack against raw-byte models like MalConv.

1. **Gradient Calculation:** The attacker computes the gradient of the loss function with respect to the input bytes.

2. **Byte Selection:** The attacker identifies which byte values, if added to the end of the file (overlay), would maximize the classification error.

3. **Impact:** Modifying less than 1% of the file size via appended padding can reduce detection accuracy by over 50%.[37] The neural network, utilizing Global Max Pooling, picks up *benign* signals from the appended padding that override the *malicious* features in the body.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

Slack Space Injection

More advanced attacks inject adversarial bytes into the *slack space* (alignment gaps between sections) or unused headers (e.g., the DOS stub). Research indicates that *slack space injection* is often more effective than append attacks because the adversarial bytes are interleaved with the code, making them harder to filter out without breaking the binary.[38]

Strategic Defenses

Robust Hashing and Randomized Smoothing

One defense strategy is Randomized Smoothing. By adding noise to the input or the embedding during inference, the defender smooths the decision boundary, removing the sharp pockets of vulnerability that adversarial examples exploit.[39] Robust Hashing forces the model to learn discrete codes rather than continuous probabilities, reducing the gradient information available to the attacker.[39]

Contrastive Learning (SimCLR-GRU)

Contrastive Learning frameworks like SimCLR offer intrinsic robustness. By training the model to recognize that a binary and its perturbed version (e.g., with reordered sections) are the same, the model learns to be invariant to those specific perturbations.[40] Combining SimCLR with GRU (Gated Recurrent Unit) networks has produced models that achieve 98.2% AUC while maintaining low false positive rates, leveraging the temporal sequence learning of GRUs to validate the structural integrity of the execution flow.[41]

Synthetic Data Augmentation

Masked Autoencoders (MAE) and diffusion models are used to generate synthetic malware samples for training. Techniques like *Cluster-Tangent Diffusion (CT-Diff)* generate high-quality synthetic samples that lie on the data manifold, effectively *inoculating* the model against variations it hasn't seen before.[42] The σ-*binary* attack framework, which uses evolutionary algorithms to optimize perturbations, serves as a benchmark for testing these defenses, revealing that even hardened models can still be vulnerable to unrestricted feature modification (94% success rate).[43]

SCALABILITY AND OPERATIONAL DEPLOYMENT

The operational reality of malware analysis, i.e., processing millions of files daily, demands architectures that balance precision with speed.

Hierarchical Analysis Pipelines

Modern defense systems employ a tiered funnel architecture to manage computational load [5]:

1. **Tier 1: Fuzzy Hashing (filtering):** Incoming files are hashed with TLSH or MinHash. Queries against the database are extremely fast (Hamming/Jaccard distance). This filters out 90% of known threats and close variants.[19]

2. **Tier 2: Deep Hashing (Indexing):** For files that pass Tier 1, a deep hash (e.g., from a CNN or KEENHash) is generated. This allows for semantic nearest-neighbor search in Hamming space, identifying variants that have been obfuscated beyond the reach of fuzzy hashes.[5]

3. **Tier 3: Full Embedding (Forensics):** For high-priority or ambiguous samples, full embeddings are generated using computationally intensive models like Gemini (GNN) or MalBERT. These allow for detailed lineage tracking, function-level clone detection, and authorship attribution.[29]

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

Summary of Deep Learning Architectures

Table 2 provides a comparative overview of the deep learning architectures discussed.

Table 2: Comparative Analysis of Deep Embedding Architectures

Architecture	Input Representation	Core Mechanism	Key Strengths	Limitations
MalConv [37]	Raw Bytes	1D CNN + Global Pooling	End-to-end learning; no feature engineering required.	Highly vulnerable to append/padding attacks; lacks semantic insight.
MalSSL [22]	Grayscale Image	CNN + Self-Supervised Learning	High accuracy (98%+) without labels; robust to visual obfuscation.	Ignores control flow logic; image mapping can be lossy.
Gemini / GeminiNet [27]	CFG / FCG / PCG	Siamese GNN + Adaptive Gating	Structure-aware; robust to instruction substitution; fuses local/global logic.	High computational cost for graph construction and matching.
SAFE [30]	Assembly Instructions	Self-Attention Network (BiRNN)	Fast 1-to-N function search; semantic instruction embedding.	Requires disassembly; limited to function-level scope.
MalBERT [44]	Opcodes / API Tokens	Transformer (BERT) + MLM	Deep contextual understanding; code-aware tokenization.	computationally intensive (inference time); resource heavy.
KEENHash [5]	Function Embeddings	LLM + K-Means + Feature Hash	*215x Faster* than peers; massive scalability for BCSA.	Dependent on the quality of the underlying LLM embedding.
FlowMalTrans [35]	Binary Code	Normalizing Flows (NF)	Cross-ISA translation (x86 *to* ARM); unsupervised training.	Complexity of training Normalizing Flows; translation fidelity.

SUMMARY AND CONCLUSION

The field of malware similarity analysis is undergoing a profound transformation. We have moved from the rigid, byte-level constraints of *Fuzzy Hashing*—where algorithms like ssdeep and TLSH provided the necessary speed for triage but lacked semantic depth—to the fluid, high-dimensional world of *Deep Hash Embeddings*.

The integration of *Graph Neural Networks* allows defenders to analyze malware based on its execution topology, ignoring the superficial obfuscations that plague static analysis. *Transformer models* have unlocked the ability to *read* assembly code, predicting malicious intent from the context of instructions. Innovations like *KEENHash* and *FlowMalTrans* are breaking down the barriers of scale and architecture, enabling real-time, cross-platform detection.

However, this sophistication introduces new risks. The susceptibility of these models to *Problem Space Adversarial Attacks*—where tiny, functional modifications to a binary can radically alter its embedding—remains the critical challenge. The future of malware defense lies in *Neuro-Symbolic* approaches that combine the interpretability and speed of hashing with the learning power of neural networks, and in *Adversarial Training* regimens that inoculate models against the very evasion techniques they are designed to detect. As

threat actors adopt AI to generate malware, the defense must rely on these advanced embedding spaces to maintain the advantage.

REFERENCES

1. Berrios, S., Leiva, D., Olivares, B., Allende-Cid, H., & Hermosilla, P. (2025). Systematic Review: Malware Detection and Classification in Cybersecurity. *Applied Sciences*, *15*(14), 7747. https://doi.org/10.3390/app15147747
2. Manjunatha, Vikram & Ramesh, Raghavendra. (2023). Machine Learning in Malware Detection: A Survey of Analysis Techniques. IJARCCE. 12. 204. 10.17148/IJARCCE.2023.12435.
3. Frieder Uhlig, Lukas Struppek, Dominik Hintersdorf, Thomas Göbel, Harald Baier, Kristian Kersting. Combining AI and AM - Improving Approximate Matching through Transformer Networks. arXiv:2208.11367v3 [cs.CR] 27 Apr 2023.
4. Fuzzy hashing - Wikipedia, accessed November 23, 2025, https://en.wikipedia.org/wiki/Fuzzy_hashing
5. Zhijie Liu, Qiyi Tang, Sen Nie, Shi Wu, Liang Feng Zhang, Yutian Tang . KEENHash: Hashing Programs into Function-Aware Embeddings for Large-Scale Binary Code Similarity Analysis, https://arxiv.org/abs/2506.11612
6. Threat Attribution using ssdeep. Medium, accessed November 23, 2025, https://nikhilh20.medium.com/fuzzy-hashing-ssdeep-3cade6931b72
7. Ssdeep xref - National Security Agency, accessed August 10, 2026, https://code.nsa.gov/emissary/xref/emissary/kff/Ssdeep.html
8. Pure Perl ssdeep (CTPH) fuzzy hashing - Ubuntu Manpage, accessed August 10, 2026, https://manpages.ubuntu.com/manpages/jammy/man3/Digest::ssdeep.3pm.html
9. Fuzzy Hashing Techniques in Applied Malware Analysis - Software Engineering Institute, accessed November 23, 2025, https://www.sei.cmu.edu/blog/fuzzy-hashing-techniques-in-applied-malware-analysis/
10. Oliver, Jonathan & Cheng, Chun & Chen, Yanggui. (2013). TLSH -- A Locality Sensitive Hash. Proceedings - 4th Cybercrime and Trustworthy Computing Workshop, CTC 2013. 7-13. 10.1109/CTC.2013.9.
11. Miguel Martín-Pérez, Ricardo J. Rodríguez, Frank Breitinger, Bringing order to approximate matching: Classification and attacks on similarity digest algorithms, Forensic Science International: Digital Investigation, Volume 36, Supplement, 2021, 301120, ISSN 2666-2817, https://doi.org/10.1016/j.fsidi.2021.301120.
12. All your hashes are belong to us: An overview of malware hashing algorithms - G DATA, accessed November 25, 2025, https://www.gdatasoftware.com/blog/2021/09/an-overview-of-malware-hashing-algorithms
13. Frank Breitinger & Harald Baier & Jesse Beckingham (2012). Security and Implementation Analysis of the Similarity Digest sdhash, https://dasec.h-da.de/wp-content/uploads/2012/08/2012_08_Breitinger_NeSeFo.pdf
14. sdhash package - github.com/eciavatta/sdhash - Go Packages, accessed November 23, 2025, https://pkg.go.dev/github.com/eciavatta/sdhash
15. Udbhav Prasad (2025). Evaluating Similariy Digests: A Study of TLSH, ssdeep, and sdhash Against Common File Modifications Traditional hashes miss unknown malware. Similarity digests like TLSH, ssdeep, and sdhash improve detection by comparing file similarities. This article benchmarks them, https://dzone.com/articles/similarity-digests-tlsh-ssdeep-sdhash-benchmark
16. Jonathan Oliver, Chun Cheng and Yanggui Chen, TLSH - A Locality Sensitive Hash, Trend Micro, https://documents.trendmicro.com/assets/wp/wp-locality-sensitive-hash.pdf
17. Liu H., Hagen J., Ali M. and Oliver J. (2023). An Evaluation of Malware Triage Similarity Hashes. In Proceedings of the 25th International Conference on Enterprise Information Systems - Volume 1: ICEIS, ISBN 978-989-758-648-4, SciTePress, pages 431-435. DOI: 10.5220/0011728500003467
18. Jonathan Oliver and Josiah Hagen (2021). Designing the Elements of a Fuzzy Hashing Scheme, 2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC) Pages: 1–6, DOI: 10.1109/euc53437.2021.00028
19. Matteo Brosolo, Asmitha K. A., Mauro Conti, Rafidha Rehiman K. A., Muhammed Shafi K. P., Serena

Nicolazzo, Antonino Nocera, Vinod P. Security through the Eyes of AI: How Visualization is Shaping Malware Detection, https://arxiv.org/pdf/2505.07574
20. N. Sai Ramana Vashista and K. Abhimanyu Kumar Patro, "Enhancing Malware Analysis Using Data Visualization Through Shared Code and Attribute Analysis," in IEEE Access, vol. 13, pp. 107482-107498, 2025, doi: 10.1109/ACCESS.2025.3582164.
21. D. Kim, J. Hur and M. Yoon, "Scalable and Multifaceted Search and Its Application for Binary Malware Files," in IEEE Access, vol. 9, pp. 112770-112779, 2021, doi: 10.1109/ACCESS.2021.3102157.
22. S. J. I. Ismail, Hendrawan, B. Rahardjo, T. Juhana and Y. Musashi, "MalSSL—Self-Supervised Learning for Accurate and Label-Efficient Malware Classification," in IEEE Access, vol. 12, pp. 58823-58835, 2024, doi: 10.1109/ACCESS.2024.3392251.
23. Zhang, Yunchun & Liao, Zikun & Zhang, Ning & Min, Shaohui & Wang, Qi & Quek, Tony Q.S. & Zhao, Mingxiong. (2024). Deep Hashing for Malware Family Classification and New Malware Identification. IEEE Internet of Things Journal. 11. 26837-26851. 10.1109/JIOT.2024.3353250.
24. Chen, Y.-H., Chen, J.-L., & Deng, R.-F. (2022). Similarity-Based Malware Classification Using Graph Neural Networks. Applied Sciences, 12(21), 10837. https://doi.org/10.3390/app122110837
25. Guo, W., Du, W., Yang, X., Xue, J., Wang, Y., Han, W., & Hu, J. (2025). MalHAPGNN: An Enhanced Call Graph-Based Malware Detection Framework Using Hierarchical Attention Pooling Graph Neural Network. Sensors, 25(2), 374. https://doi.org/10.3390/s25020374
26. Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, Anis Zouaoui. A Survey on Malware Detection with Graph Representation Learning, https://arxiv.org/pdf/2303.16004
27. Z. H. Qaisar, S. H. Almotiri, M. A. Al Ghamdi, A. A. Nagra and G. Ali, "A Scalable and Efficient Multi-Agent Architecture for Malware Protection in Data Sharing Over Mobile Cloud," in IEEE Access, vol. 9, pp. 76248-76259, 2021, doi: 10.1109/ACCESS.2021.3067284.
28. Detection of Prevalent Malware Families with Deep Learning - Microsoft, accessed November 23, 2025, https://www.microsoft.com/en-us/research/wp-content/uploads/2020/07/Siamese_Milcom2019.pdf
29. Kartikeya Aneja, Nagender Aneja, Murat Kantarcioglu, Learning Joint Embeddings of Function and Process Call Graphs for Malware Detection, https://arxiv.org/html/2510.09984v1
30. Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, Roberto Baldoni, SAFE: Self-Attentive Function Embeddings for Binary Similarity, arXiv:1811.05296
31. Jia Y, Yu Z, Hong Z. Semantic aware-based instruction embedding for binary code similarity detection. PLoS One. 2024 Jun 11;19(6):e0305299. doi: 10.1371/journal.pone.0305299. PMID: 38861533; PMCID: PMC11166306.
32. Luca Massarelli, SAFE: a step into the creation of embeddings for binary code similarity detection, Medium, accessed November 24, 2025, https://medium.com/@massarelli/safe-self-attentive-function-embedding-d80abbfea794
33. Rahali, A., & Akhloufi, M. A. (2023). MalBERTv2: Code Aware BERT-Based Model for Malware Identification. Big Data and Cognitive Computing, 7(2), 60. https://doi.org/10.3390/bdcc7020060
34. Pascal Maniriho, Abdun Naser Mahmood, Mohammad Jabed Morshed Chowdhury. EarlyMalDetect: A Novel Approach for Early Windows Malware Detection Based on Sequences of API Calls, https://arxiv.org/html/2407.13355v1
35. Minghao Hu, Junzhe Wang, Weisen Zhao, Qiang Zeng, Lannan Luo, FlowMalTrans: Unsupervised Binary Code Translation for Malware Detection Using Flow-Adapter Architecture, https://arxiv.org/html/2508.20212v1
36. Kshitiz Aryal, Maanak Gupta, Mahmoud Abdelsalam, Moustafa Saleh, Explainability Guided Adversarial Evasion Attacks on Malware Detectors, https://arxiv.org/html/2405.01728v1
37. Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, Fabio Roli, Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables, https://arxiv.org/abs/1803.04173
38. Pavla Louthánová, Matouš Kozák, Martin Jureček, Mark Stamp, and Fabio Di Troia. "A comparison of adversarial malware generators" Journal of Computer Virology and Hacking Techniques (2024). https://doi.org/10.1007/s11416-024-00519-z
39. Kulkarni, S. S., & Di Troia, F. (2025). Robust Hashing for Improved CNN Performance in Image-

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VIII August 2026

Based Malware Detection. Electronics, 14(19), 3915. https://doi.org/10.3390/electronics14193915
40. CrowdStrike, Researchers Explore Contrastive Learning for Malware Detection, accessed November 23, 2025, https://www.crowdstrike.com/en-us/blog/contrastive-learning-enhance-malware-threat-detection/
41. Alsubaei, Faisal & Almazroi, Abdulwahab & Atwa, Walid & Almazroi, Abdulaleem & Ayub, Nasir & Jhanjhi, Noor. (2025). Adaptive malware identification via integrated SimCLR and GRU networks. Scientific Reports. 15. 10.1038/s41598-025-08556-4.
42. Kapoor, G., Nadipalli, S., & Di Troia, F. (2025). Embedding-Driven Synthetic Malware Generation with Autoencoders and Cluster-Tangent Diffusion. Applied Sciences, 15(21), 11791. https://doi.org/10.3390/app152111791
43. Mostafa Jafari, Alireza Shameli-Sendi, Evaluating the robustness of adversarial defenses in malware detection systems, https://arxiv.org/html/2505.09342v1
44. Rahali, Abir & Akhloufi, Moulay. (2023). MalBERTv2: Code Aware BERT-Based Model for Malware Identification. Big Data and Cognitive Computing. 7. 60. 10.3390/bdcc7020060.