Section: CS

MAGNITO

Malware **Gen**ome **I**dent**i**ficati**o**n

Manish Jha

Abstract:

As malware variants proliferate, static signature-based detection struggles to keep pace with code reuse and metamorphic obfuscation. Recent advances in sequential opcode embedding and API-profiling demonstrate strong family classification, yet often require heavy graph extraction or large corpora [4]. We introduce a lightweight malware genome similarity framework that (1) computes block-level sums of unique opcode codes to neutralize junk-code insertion, (2) normalizes operands to reg/const/addr to counter register renaming, (3) profiles API call frequencies via TF-IDF, and (4) compresses 1.7 M-dim vectors into 300 dims with Truncated SVD before training an XGBoost model. Evaluated on five Windows, Mac, and Linux malware families—including Zeus and Mirai with ≥ 400 unpacked samples, selected via VirusTotal `first_seen` filters. Our approach achieves 100% classification accuracy for features extracted on a host system and ~95% overall classification accuracy. We deliver a toolkit comprising IDAPython script, vectorizer and training script, evaluation script and Mongo Db support functions. This static pipeline is robust to instruction reordering, control-flow flattening, and sandbox fingerprinting, complementing existing dynamic analysis.

Introduction –

Malware authors routinely employ instruction reordering, opaque predicates, and junk-code insertion, such as redundant NOP operations, to defeat signature-based detectors by breaking linear opcode patterns without altering program semantics [11]. Control-flow flattening and register renaming (e.g., swapping ADD EAX,1 for INC EAX) further obscure code structure [10]. While dynamic analysis can expose hidden behaviours, advanced samples detect sandbox environments via timing checks and environment fingerprinting (e.g., SMBIOS/WMI queries) and then withhold malicious payloads during execution.

Sequential opcode embedding methods, such as SOEMD, capture malicious patterns via random-walk sampling of opcode graphs, achieving > 90% detection [4]. Hybrid models that fuse static opcodes, API calls, and dynamic traces further push accuracies beyond 95% but often demand extensive feature-engineering and large labelled corpora [9]. Graph-based CFG hashing schemes provide compact structural signatures but incur high preprocessing costs [4]. Our malware genome framework bridges these gaps by focusing on lightweight, order-invariant representations that scale to diverse platforms without reliance on extensive unpacking or heavy graph analysis.

This research proposes a lightweight solution for family classification by extracting and encoding malware "genes" the opcodes of instructions and static APIs derived from disassembler tools such as IDA Pro or GHIDRA into a condensed format. This work focuses on static, TF-IDF-driven similarity as a first step. We do not address dynamic or graph-based methods. Analogous to the cellular genome, which encodes the biological functions and traits

of an organism, the "malware genome" comprises the fundamental operational instructions that define malware behaviour.
Reason: Just as the genome in biological systems provides a comprehensive blueprint for an organism's functions and evolutionary history, the opcode sequence of a malware sample encapsulates its intrinsic operational logic and lineage. By focusing on these essential code components, it becomes possible to detect functional similarities that persist despite superficial modifications. This is particularly critical given that threat actors often employ evasion techniques such as instruction reordering and the insertion of junk code (e.g., NOPs) to obscure the true nature of the code. Such normalization at the opcode level allows for the identification of code reuse across different malware families and versions, facilitating the discovery of underlying relationships even when surface-level obfuscations are present. The approach leverages the observation that threat actors frequently reuse code segments in new iterations or entirely new malware strains. APIs help establishing technique, tactic, procedure of family. By extracting and storing these extracted sequences in a structured format, efficient similarity searches can be performed, thereby enabling rapid identification of code reuse and shared operational logic across diverse malware samples. This focus on functional similarity, rather than on transient surface-level differences, significantly enhances the capacity for effective malware detection and classification.
Challenges: Techniques such as encryption and packing significantly complicate opcode extraction by obfuscating the malware code. To address this, the proposed methodology assumes the availability of preprocessing steps such as those implemented via open-source sandbox environments or lightweight dynamic unpacking techniques to recover the underlying code prior to opcode extraction. One such solution is unpacking using Cuckoo sandbox and CAPE.

Background –

Adversaries employ a variety of sophisticated methods to detect and circumvent sandbox and dynamic analysis systems, ensuring that malicious payloads remain dormant or behave benignly when under observation.

- **Environment and Artifact Fingerprinting:** Malware often queries system artifacts to determine if it is running in a virtualized or sandboxed environment. Common checks include inspecting process lists for known analysis tools, registry keys for virtualization software, file paths, or hardware characteristics such as CPU core count and available memory. Some samples use SMBIOS or WMI queries to detect specific sandbox vendor signatures or unusual environment configurations (e.g., single-core CPUs, minimal disk space) that differ from real user machines. Attackers also maintain dynamic lists of sandbox hostnames updating their evasive checks as new sandbox environments are identified to avoid executing on unknown analysis infrastructure.
- **Timing-Based and Stalling Techniques:** Time-based evasion relies on delaying malicious behaviour until after typical sandbox timeouts. Malware may invoke long sleep calls (e.g., Sleep(300000)) or perform no-op loops to exhaust analysis time or evade resource-constrained sandboxes. More advanced variants calibrate sleep durations based on system timers or CPU speed to adaptively bypass dynamic monitors that pause or speed-up execution.

- **Human Interaction and Reverse Turing Tests:** To distinguish automated analysis from real users, malware can require specific user interactions such as mouse movements, keystrokes, or clipboard events before activating malicious routines. Without these inputs, the malware remains inert, effectively passing through sandbox analysis unobserved. Some samples even implement rudimentary "captcha" logic, asking for simple user inputs to confirm a human-controlled environment.
- **Anti-Debugging and Anti-Instrumentation:** Dynamic analysis frameworks often utilize API hooking or debuggers to trace execution. Malware counters this by detecting the presence of debuggers scanning for modified function pointers, or checking for inline hooks in critical APIs. Additionally, some use timing discrepancies between hooked and unhooked API calls to detect instrumentation, then alter behaviour if hooks are detected.
- **Network and Host-Level Checks:** Since many sandboxes restrict or simulate network traffic, malware may test for realistic network conditions such as DNS resolution delays, reachable C2 endpoints, or specific network interface configurations before proceeding. It may also perform domain generation algorithm (DGA) lookups or contact benign-looking domains to verify live Internet connectivity, remaining dormant in isolated sandbox networks.

These slow down and impact response on new malware, due to opaque evasion techniques which can only be detected using static human analysis. To counter this and speed up analysis and response we can use static features extracted from malware to determine family and implement defences used against adversary.

But, static analysis of malware is continually challenged by adversarial evasion techniques at the opcode level, where attackers mutate their code's syntactic appearance while preserving its semantic behaviour. Common metamorphic and obfuscation strategies include:

- **Garbage (Junk) Code Insertion**:
  Adversaries insert instructions that have no effect on program logic such as redundant `nop` operations or other no-op-equivalents to disrupt statistical or signature-based detectors and vastly increase code variants.
- **Instruction Reordering and Opaque Predicates:**
  Malware authors reorder independent instructions or basic blocks, often via unconditional jumps or conditional branches gated by opaque (always-true or always-false) predicates, to break linear opcode patterns while preserving execution order.
- **Register Renaming and Instruction Substitution:**
  Registers may be swapped or renamed arbitrarily, and instructions can be replaced by functionally equivalent alternatives to defeat defences that rely on exact opcode sequences.
- **Control-Flow Flattening and Code Shuffling:** By flattening a function's control flow merging multiple blocks under a dispatcher and shuffling subroutine calls or block order, attackers obscure the original program structure.
- **Packing and Encryption:**
  Packers and crypters wrap code in an encrypted or compressed layer that is unpacked at runtime, effectively hiding all static opcodes until the payload

executes. This requires dynamic unpacking or sandboxing to recover the genuine instruction stream.

These evasion techniques severely limit the effectiveness of conventional static analysis, which inspects binaries without execution and thus relies on recognizable opcode patterns.

We will extract features to counter these limitations and leverage Machine learning models to predict family based on past malware feature set of the family.

Methodology – 1. Data Collection and Preprocessing

- **Sample Acquisition:**
  To train our malware genome similarity framework, we curated a dataset comprising over ~500 non-packed, non-encrypted malware samples spanning five prominent families: Zeus, SpyEye, Mirai, Pirrit, and Flashback. These families were selected based on their historical significance, prevalence across Windows, Linux, and macOS platforms, and the availability of sufficient unpacked samples suitable for static analysis.
  Samples were sourced from VirusTotal using advanced search modifiers and metadata filters. Specifically, we employed the fs: (first seen) modifier to retrieve samples first observed before February 24, 2024, ensuring a temporal separation between training and evaluation datasets. To confirm the suitability of samples for static analysis, we utilized VirusTotal's metadata to exclude files exhibiting characteristics of packing or encryption. This included analysing file entropy levels, absence of known packer signatures, and ensuring that the files could be successfully disassembled using tools like IDA Pro or Ghidra. The selection process was further refined using VirusTotal's clustering mechanisms, such as imphash, vhash, and ssdeep, to group samples accurately by family and to identify variants with shared code bases [8]. This methodology ensured that our dataset not only represented a diverse set of malware families but also maintained the integrity required for effective static analysis and genome extraction.
  By adhering to this rigorous selection and validation process, we ensured that our dataset was both representative of real-world malware threats and conducive to the static analysis techniques employed in our study.

- **Disassembly and Function Identification:**
  Using disassembler tools such as IDA Pro or Ghidra, each malware binary is disassembled to extract user-defined functions and APIs. Library functions are excluded to focus on the code authored by threat actors.

2. Feature Extraction

- **Block-Level Opcode Extraction:**

Each identified function is segmented into basic blocks (a sequence of instructions with a single entry and exit point). For every basic block:

- **Opcode Sum:** Each opcode (mnemonic) is mapped to a unique numerical code. The sum of these codes for the block is computed, yielding a quantitative measure of the block.

  By mapping each mnemonic to a fixed numeric code and summing only unique codes per block, we neutralize the impact of junk code insertion and instruction duplication, since redundant opcodes do not increase the block's signature.
- **Function-Level Statistical Summaries:** Compute per-function statistics (mean, standard deviation, min, max) over block-sum features, then average these across the binary. This smooths out local transformations such as control-flow flattening and block shuffling.
- **Mnemonic Sequence:** The mnemonics are extracted in order, forming a "word" that represents the block.

  Each basic block's mnemonics are treated as a "word" and aggregated per function. Sorting these function-level words before TF-IDF vectorization ensures that instruction and function reordering does not alter the overall representation.
- **Normalized Instructions:** Instructions are normalized by replacing operands with placeholders (e.g., registers → "reg", constants → "const", addresses → "addr"), capturing the functional structure while eliminating irrelevant variations.

  This helps to counter register renaming and immediate-value polymorphism. This preserves semantic structure while removing irrelevant syntactic variance.

- **API Extraction:**
  The set of APIs imported by the malware is extracted from the disassembled code to capture functional dependencies and operational characteristics that are less easily obfuscated by opcode-level transformations.

3. Feature Aggregation and Packing

- **Function-Level Aggregation:** For each function:
  - The block-level mnemonic "words" are concatenated to form a functionlevel text representation.
  - The block-sum values from each function are summarized using statistical measures (e.g., mean, standard deviation, minimum, maximum).
- **Binary-Level Aggregation:**
  To address order variability (i.e., functions may appear in different orders across samples), the function-level texts are sorted and concatenated, resulting in an aggregated text representation that is order invariant. Similarly, the numerical features are averaged across functions to represent the entire binary.

4. Feature Vectorization

- **Textual Feature Vectorization:**
  The aggregated function-level text is converted into a fixed-dimensional vector using TF-IDF. This bag-of-words approach ensures that the inherent order of the

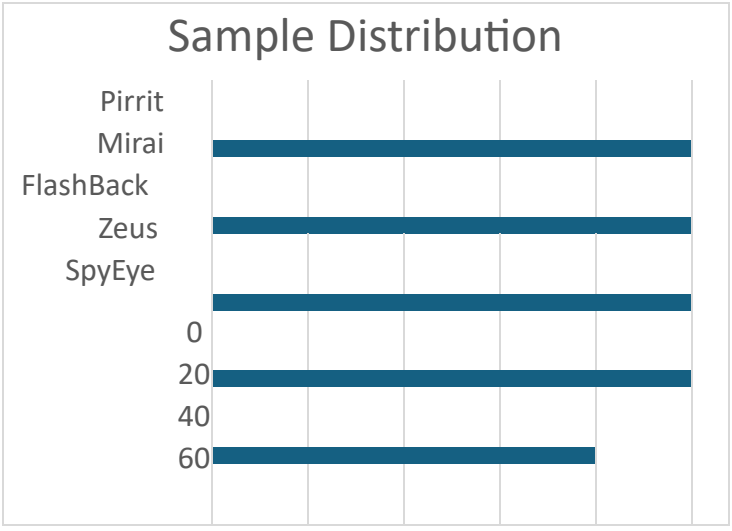
code blocks is rendered invariant while preserving token frequency information. A similar TF-IDF vectorization is applied to the extracted API names.

- **Numerical Feature Normalization:**
  The computed statistical features from the block-sum values are normalized using standard scaling to ensure comparability across samples.
- **Multi-Modal Feature Fusion:**
  The TF-IDF vectors (for mnemonic text and APIs) and the normalized numerical vectors are concatenated using sparse matrix operations. This results in a fixedlength feature vector for each malware sample that comprehensively represents both structural (opcode-based) and functional (API-based) attributes.

5. Model Training and Similarity Learning

- **Model Options:**
  Two modelling approaches are considered:
  - **XgBoost Model:**
    XgBoost is a powerful ensemble learning algorithm that uses gradient boosting over decision trees to directly map input features to malware family labels. Instead of learning a continuous embedding space for similarity-based search, XgBoost focuses on capturing complex, non-linear interactions within the extracted static features—such as block-level opcode sums, mnemonic sequences, and API usage—to achieve robust classification. By optimizing a loss function through sequential tree boosting and providing insights via feature importance metrics, XgBoost effectively distinguishes malware families.

  - **Direct Classification Models:**
    As an initial solution, a RandomForestClassifier is employed to directly predict the malware family from the aggregated feature vector.
- **Training Procedure:**
  The combined feature vectors are split into training and test sets. The chosen classifier is trained using the training set, and hyperparameters are tuned to optimize performance. Standard metrics—precision, recall, F1-score, and accuracy—are used to evaluate the model's performance.



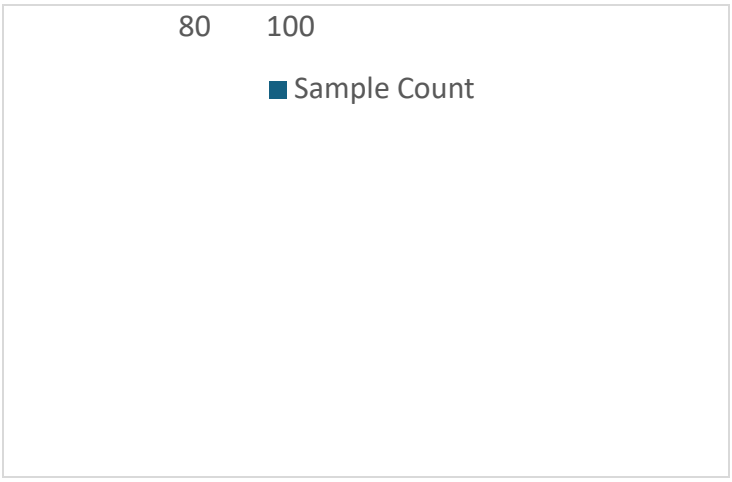
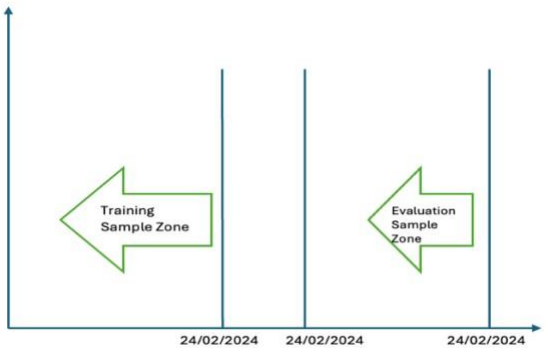


Fig 1. Training Set

6. Inference on New Samples

- **Sample Acquisition:**
  To evaluate our similarity framework, we curated a dataset comprising over ~50 non-packed, non-encrypted malware samples spanning five families under study: Zeus, SpyEye, Mirai, Pirrit, and Flashback.
  Samples were sourced from VirusTotal using advanced search modifiers and metadata filters. Specifically, we employed the fs: (first seen) modifier to retrieve samples first observed after February 25, 2024, ensuring a temporal separation between training and evaluation datasets. Our evaluation data set comprised of samples having fs modifier was 2025 as virus total lists sample in descending order of first seen. To confirm the suitability of samples for static analysis, we utilized VirusTotal's metadata to exclude files exhibiting characteristics of packing or encryption. This included analysing file entropy levels, absence of known packer signatures, and ensuring that the files could be successfully disassembled using tools like IDA Pro or Ghidra.
- **Disassembly and Function Identification:**
  Using disassembler tools such as IDA Pro or Ghidra, each malware binary is disassembled to extract user-defined functions and APIs. Library functions are excluded to focus on the code authored by threat actors.
- The aggregated features are vectorized using the pre-trained TF-IDF models and scaler, ensuring the new sample’s feature vector has the same fixed dimension.
- The trained model is then used to predict the family of the new sample or to compute its similarity to known families.
- Predicted family for file is dumped in json file along with hash for sample. If model fails to determine family, it marks family as ‘unknown’.



Training & Evaluation Sample Distribution

7. Architecture

Extracted features should be stored for model retraining or updating new features. Also, numerical coding of opcode should be consistent for accuracy. Using NoSQL DB like Mongo helps to achieve this goal.

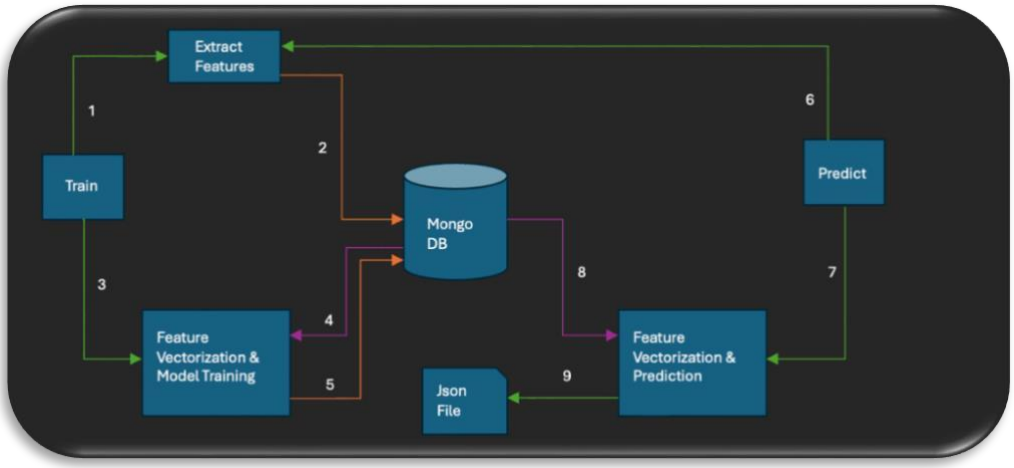


Fig. II Architecture Diagram

Result –

Feature extraction was performed on native host system per malware except Mirai, which was processed on Windows host.

Family	File Type	VM OS
Pirrit	MachO	MacOS
Mirai	ELF	Windows
SpyEye	PE	Windows
Zeus	PE	Windows
flashback	MachO	MacOS

OS used per family

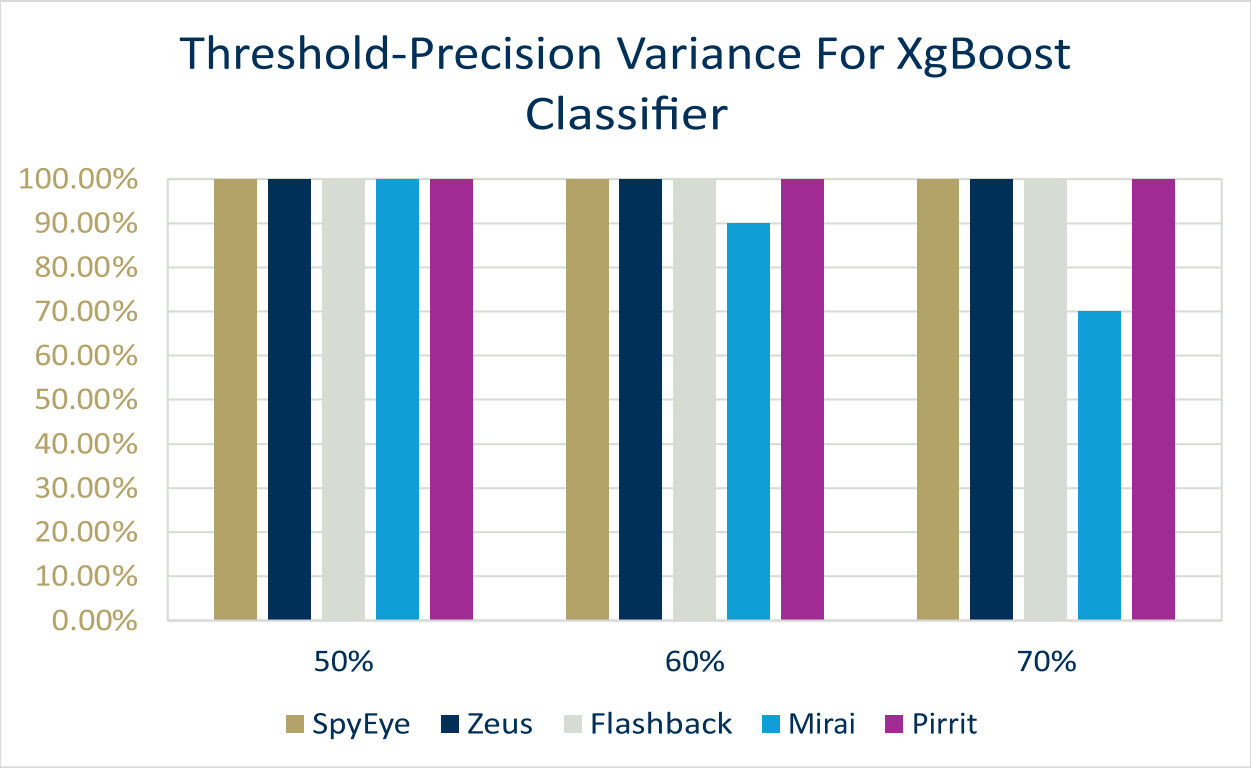
Classification performance was evaluated elevated (0.70) decision threshold to balance sensitivity and specificity across five malware families (Zeus, SpyEye, Mirai, Pirrit, Flashback).

Solution achieved accuracy of ~95% overall. Accuracy per family is depicted in table below

Family	Sample#	Accuracy
Pirrit	7	1.00
Mirai	10	0.70-0.80
SpyEye	10	1.00
Zeus	10	1.00
flashback	10	1.00

Evaluation result Table

On closer analysis, accuracy is 100% for all malware families under study except Mirai. To evaluate this further, classification performance was evaluated with 3 different decision threshold – 0.50, 0.60, 0.70. Mirai performance gradually drop on these different decision threshold highlighting accuracy of feature extraction on native host versus on foreign host.



Accuracy for different decision threshold

Performance –

We evaluated the end-to-end runtime of our static feature-extraction and classification pipeline on a mid-range analysis workstation (Intel i7, 16 GB RAM). Feature extraction from each malware sample, comprising disassembly in IDA Pro, basic-block segmentation, opcode normalization, and API parsing, required 2 to 5 minutes per sample, depending primarily on binary size and complexity.

Once all samples were pre-processed, the feature vectorization (TF-IDF on block-level opcode "words" plus API call frequencies) and dimensionality reduction (Truncated SVD to 300 components) for the full corpus (≈ 500 samples) completed in under 5 minutes. Training the lightweight XGBoost classifier on these reduced vectors required an additional 12 minutes, yielding a total vectorization + training time of 5-6 minutes. This performance

profile demonstrates that our genome-based similarity framework can process and classify new samples in under 10 minutes each (including feature extraction), making it suitable for near-real-time malware triage in operational environments.

Limitation –

Solution has following limitation-

- **Not Suitable for Script-based Malware:**
  The method is designed for static analysis of compiled binaries (e.g., PE, ELF, MachO) and relies on disassembly of machine code. Script-based malware (e.g., JavaScript, PowerShell, VBScript) cannot be processed by this pipeline.
- **Sensitivity to Obfuscation and Packing:**
  Although normalization (e.g., instruction and operand replacement) mitigates some evasion techniques, highly obfuscated, polymorphic, or metamorphic malware may still produce unreliable or incomplete feature extraction.
- **Dependency on Disassembly Accuracy:**
  The entire pipeline depends on the quality of disassembly. Inaccurate function recovery or misidentification of code blocks due to anti-disassembly techniques can compromise the extracted features. ○ Platform: For accurate feature extraction we need IDA with dependent platform. ○ FAT-Binary: OSX file type which contains multiple arch binaries. This needs to extracted before processing.
- **Accuracy of training set label:**
  Predicted family is dependent on initial label of training data. Any error or incorrect label will impact predicted family.

Conclusion –

By combining these normalized, order-invariant representations with robust vectorization and similarity-learning models, resulting framework effectively distinguishes malware families despite extensive metamorphic and obfuscation techniques. This layered feature extraction strategy directly addresses the adversary's evasion arsenal at the opcode level and enables accurate similarity detection across diverse variants.

Future Work –

The research aims to apply findings directly to production, requiring ready-to-use adjustments.
Develop online learning mechanisms to adapt models incrementally as new malware emerges, maintaining system robustness in dynamic threat environments.

- Combine dynamic analysis with the current static approach to enhance detection, particularly against sophisticated obfuscation techniques.

References:

[1] F. Labs, “2H23 Global Threat Landscape Report,” 2023. [Online]. Available: https://www.fortinet.com/content/dam/fortinet/assets/threat-reports/threat-landscapereport-2h-2023.pdf.

[2] CrowdStrike, “GlobalThreatReport2024,” 2024. [Online]. Available: https://go.crowdstrike.com/rs/281-OBQ-266/images/GlobalThreatReport2024.pdf.

[3] B. Y. Kim et al., “Hybrid Malware Variant Detection Model with Extreme Gradient Boosting,” J. Cybersecurity, 2023.

[4] A. Kakisim et al., “Sequential opcode embedding based malware detection method,” *Comput. Electr. Eng.,* vol. vol. 98, p. 107703, 2022.

[5] X. Zeng, “Static Signature Based Malware Detection Using N gram Opcode Sequences,” in *Proc. Int. Conf. Data Mining*, 2020.

[6] A. G. K. Manrirho et al., “Hybroid: Opcode2Vec, Function2Vec & Graph2Vec for Multimodal Malware Embedding,” in *ISC*, 2021.

[7] M. A. Smith and L. T. F. Alvarez, “Malware classification using XGBoost and Truncated SVD on static features,” in *ASTE Trans. Earth Sci. Eng.*, 2020.

[8] VirusTotal, “VirusTotal Public API v3 Documentation,” 2024. [Online].

[9] J. S. Moon and S. G. Kakisim, “CNN LSTM for Malware Classification using Opcodes & API Calls,” *Inf. Sci.,* vol. vol. 535, p. pp. 1–15, 2020.

[10] V. R. K. Lee and W. H. Lee, “Multiclass Malware Classification with Static Opcodes or Dynamic API Calls,” Springer, 2022.

[11] R. T. S. Wooster, “Obfuscation-Resilient Static Analysis for Binary Malware Detection,” WSU, 2024.