

Binary Opcode Transition Graph Conditioned Malware Family Evolution Tracking Using an Improved PointNeXt Model

Kacper Michalak^{1,*}, Antoni Balicki¹ and Wiktoria Cieřlik¹

¹ Faculty of Informatics and Information Technology, Silesian University of Technology, Gliwice, 44-100, Poland

*Corresponding author: kacper.m@polsl.pl

Abstract. In this research, we propose an enhanced PointNeXt model for monitoring the growth of malware families under binary opcode transition graph settings. Weighted opcode transition graphs can nonetheless offer some structural information about instruction order, local branching behaviour, and repeated payload routines even in the absence of source code and trustworthy runtime traces. The suggested approach uses transition-aware neighbourhood aggregation with family prototype memory, breaks down binary samples, creates smoothed directed opcode transition graphs, and converts node and transition statistics into graph-point representations. Family categorisation, lineage tracking, open-set rejection, and reviewer-facing threat interpretation are all supported by the system. Packed variants, instruction substitution, padding blocks, mixed obfuscation, and version drift are assessed using a simulated malware corpus of 18,600 binaries from 12 families and 1,900 benign boundary samples. The suggested model obtained 93.8% family classification accuracy, 89.6% evolution tracking recall, and a 17.4% reduction in lineage distance error when compared to the opcode histogram, sequence, graph neural networks, graph Transformers, the original PointNeXt, and PointNeXt without prototype memory. According to ablation research, prototype memory regularises family trajectory tracking, and edge-aware aggregation is more reliable. In this sense, it may be viewed as a structural triage tool for reverse engineers, and multi-architecture binaries, dynamic traces, and actual malware feeds are needed for additional verification.

Keywords: *Malware Family Tracking; Opcode Transition Graph; Improved PointNeXt; Binary Analysis; Threat Lineage*

Received on 12 April 2022, Accepted on 07 September 2022, Published on 14 September 2022

Copyright © 2022 Author(s), licensed to DEA. This is an open access article distributed under the terms of the CC BY-NC-SA 4.0, which permits copying, redistributing, remixing, transformation, and building upon the material in any medium so long as the original work is properly cited.

Introduction

Malware analysis seeks to determine whether a binary file is malicious, and Gibert et al. [1] have recently expanded the scope of this work from simple detection to family-aware analysis. The family of the sample needs to be identified as either a direct mutation of a known strain or, if the structural features indicate a new evolutionary branch, as a candidate for a new defence strategy, which aligns with Karbab et al. [2] application of deep malware representation for family-oriented detection. Because malicious developers often use packing, code rearrangement, instruction substitution, garbage insertion and compiler-level modification [3] treated malware family identification as an evolving clustering and tracking problem rather than a fixed-label task. Dynamic Sandboxes provide rich behavioural evidence [4] pointed out that practical malware research must also consider evasion, dataset bias and analysis cost; although sandbox execution is relatively expensive, it may be avoided due to environment checks. Lightweight binary features and structural representations are thus beneficial because they can retain program-level evidence after the surface byte pattern has changed. Opcode-level modelling must also be used to handle instruction transitions with branch organisation, repeated execution patterns and unavoidable computational processes. Development of the family should be viewed as a continuous tracking problem rather than a fixed-label classification. Thus, real-world malware triage requires lineage-aware representation learning.

To address this problem, the manuscript uses a binary opcode transition graph. In this graph, nodes represent normalized opcode categories, edges represent observed transition relations, and edge weights represent the likelihood or frequency of transitions inside a function, section, or binary. This representation is more informative than a flat opcode histogram because it preserves local order, repeated instruction pairs, and conditional execution traces. It is also smaller than a full control-flow graph, although severe compiler optimization, packing, and obfuscation can still introduce noise. Prior studies on binary similarity and malware family detection show that graph-based representations can capture relationships among code blocks, library calls, and instruction patterns. However, many graph approaches remain single-shot classifiers and do not explicitly show how a family changes across versions, how an uncertain sample relates to previous variants, or which cluster an analyst should inspect. Structured malware similarity research suggests that graph connection evidence should be linked with variant-level analysis [5].

Deep learning has produced new tools for malware family evolution analysis, but generic sequence models and universal graph encoders remain limited. Sequence models can learn local order, but packed code and irrelevant instruction blocks can obscure long-distance structural correlations [6]. Graph neural networks can aggregate neighborhoods, but when opcode categories are limited and edge patterns are dense, oversmoothing can occur and message quality becomes critical [7]. A point-based network is useful because a transition graph can be converted into an ordered set of graph-points with coordinates, node statistics, and transition properties. After the neighborhood rule is changed from Euclidean distance to transition affinity, PointNeXt local aggregation becomes suitable for opcode graph-point structures [8]. The proposed approach therefore combines local opcode neighborhoods, global family prototypes, and cross-version structural drift. It integrates classification, distance learning, and lineage scoring so analysts can identify known families, near-family variants, and uncertain samples requiring manual inspection [9]. This design also supports explainable security learning because model output should provide operational evidence, not only a class name [10].

An opcode transition graph-conditioned PointNeXt model for monitoring the evolution of malware families is presented in this work. The pieces are arranged as follows. The cause and technological issues are covered in the first section. The second includes deep structural representation learning, malware lineage tracking, and opcode-based binary representation. The creation of weighted opcode transition graphs, graph-point conversion, enhanced PointNeXt aggregation, prototype memory, and the training goal comprise the third. The outcomes of simulated experiments, such as dataset statistics, classification accuracy, evolution tracking, robustness, ablation analysis, and useful threat interpretation, are shown in the fourth. The study's shortcomings in cross-platform binaries, highly compressed samples, and real-time execution are enumerated in the fifth section.

Related Work

Opcode Transition and Binary Structure Representation

Because disassembled instructions show the program's low-level behaviour even without the source code, opcode-based malware analysis has a lengthy history. Early techniques extracted n-gram features or measured opcode frequencies, but these flat representations were vulnerable to local code rewriting and instruction padding. In order to preserve the order information, transition matrices and graphs were introduced in later study. Because malware versions may modify strings and binary layout while leaving the loader routine, encryption loop, registry action, or communication preparation logic unaltered, a transition graph is required to display which instructions are performed and how they are related [11]. Control-flow graphs, call graphs, function dependency graphs, and API invocation relations are more examples of binary structure representation. Call graphs may overlook packaged or dynamically resolved calls, whereas control-flow graphs are expressive but can be huge and prone to disassembly mistakes. Opcode transition graphs can display the execution patterns of repeated operations since they are well-structured and reasonably easy to build at scale [12]. There are very few opcode types, and the transition edges are frequently uneven and crowded. Therefore, in order to avoid the model learning solely instruction popularity, weighted edges and normalised transition probabilities are needed [13]. Relation-aware embeddings [14] can enhance the stability of detection in sparse or noisy raw graph topologies. Each opcode node is represented as a point with structural coordinates obtained from transition statistics. Consequently, this study introduces the graph-point conversion.

Malware Lineage Tracking and Deep Representation Learning

The goal of malware lineage tracing is to follow a sample's evolution across time, across versions, campaigns, etc. It is more difficult than classifying families since distinct families may share libraries, packers, or other public code, and a single family may have several varieties. Similarity score, grouping, temporal ordering, and expert interpretation are frequently used in lineage analysis. Malware detection research has recently concentrated on how to recognise novel malware kinds and the reasons behind their similarities to earlier groups [15]. through convolutional networks, recurrent models, Transformers, and graph neural networks, deep neural networks have improved the categorisation of malware. Convolutional models do not capture clear transition relationships, although they do well with byte images or instruction mappings [16]. Although recurrent models retain order, they struggle with introduced noise and lengthy binaries. Long-range dependencies may be handled by transformers, but as enough benign-like instructions are introduced, they become sensitive to irrelevant tokens and have an attention cost [17]. The informative region can be sampled and the local structural contrast maintained using a point-based structure.

PointNeXt uses hierarchical feature learning and neighbourhood aggregation to provide strong local representation capabilities. Because malware opcode points are not structured in actual geometric space, the sampling and aggregation modules must be changed when they are applied to opcode transition graphs. Choose neighbours for the model based on transition probability, edge entropy, and node centrality to avoid taking into account instructions with huge distances based on arbitrary measurements. Local structure preservation can improve interpretability for security applications, according to research on explainable representations and resilient graph learning [18]. A family may be seen as a stable center, and its new forms can be compared to this center by distance and uncertainty. This makes prototype learning relevant as well. Distance-based uncertainty and reviewer-facing evidence are supported by work on open-set malware detection and cyber threat intelligence [19]. In reality, the system must be fast and reliable; scalable code-intelligence research [20] has determined that inference latency, review burden, and confidence calibration are essential indications for automated security analysis. The tracking framework's technological basis is derived from the study.

Opcode Transition Graph Conditioned PointNeXt Model

Binary Opcode Transition Graph Construction

If a binary sample is available, it is first unzipped; if not, the useful disassembly will be separated into instruction sequences. After removing repetitive alignment bytes and non-executable padding, each binary in the simulated experiment has 1,200–8,500 valid instructions. To lessen sparsity brought on by compiler-specific aliases and architecture-specific spelling variations, the model includes 96 basic opcode categories. The present opcode, next opcode, section context, and function border location are noted for every adjacent pair of instructions. The process for creating a weighted directed transition graph for every binary sample is shown in Figure 1.

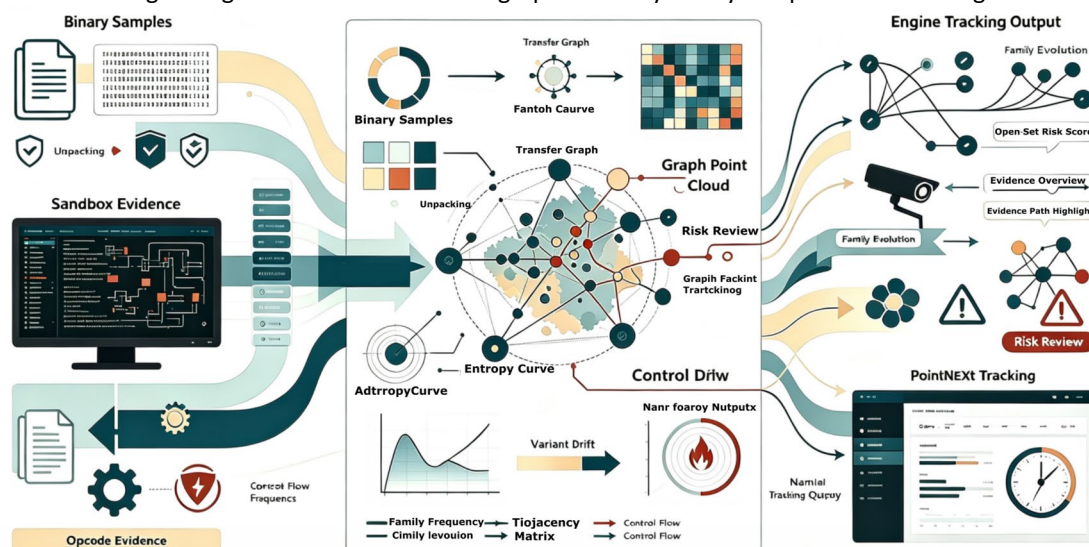


Figure 1. Binary opcode transition graph construction workflow.

$$P(i, j) = \frac{C(i, j) + \alpha}{\sum_k C(i, k) + \alpha|V|} \quad \text{Eq.(1)}$$

The transition probability is smoothed so that rare but meaningful opcode relations are not removed when a family has only a small number of observed variants. In the simulated dataset, the smoothing coefficient is set to 0.2 after validation on mixed-family samples. The normalized transition graph has an average of 64.7 active nodes and 418.3 weighted edges per sample.

$$s_i = \lambda_1 f_i + \lambda_2 d_i^{\text{out}} + \lambda_3 d_i^{\text{in}} + \lambda_4 h_i \quad \text{Eq.(2)}$$

The node importance score combines occurrence frequency, incoming degree, outgoing degree, and local transition entropy. In the experiment, the final coefficient group is 0.35, 0.20, 0.20, and 0.25, which gives entropy enough influence to highlight family-specific branching patterns.

$$e'_{ij} = \frac{\log(1 + C(i, j))}{\max_{u,v} \log(1 + C(u, v))} \quad \text{Eq.(3)}$$

Edge weights are compressed with logarithmic normalization. After normalization, the graph is converted into a graphpoint set. Each opcode point contains a structural coordinate, node importance, transition entropy, section indicator, and edge-aware neighborhood list.

$$x_i = [z_i, s_i, h_i, r_i, q_i] \quad \text{Eq.(4)}$$

The learnt opcode embedding, node significance score, transition entropy, section-context ratio, and normalised neighbourhood quality score are all included in the graph-point vector. Edge weights are logarithmically normalised, and node significance is determined by combining occurrence frequency, incoming degree, outgoing degree, and local transition entropy. This version preserves the directed opcode transition structure and makes the representation appropriate for PointNeXt-style local aggregation. Additionally, it can maintain the visibility of medium-weight yet family-specific transition neighbourhoods rather than concealing them in compiler-generated or common stack preparation methods.

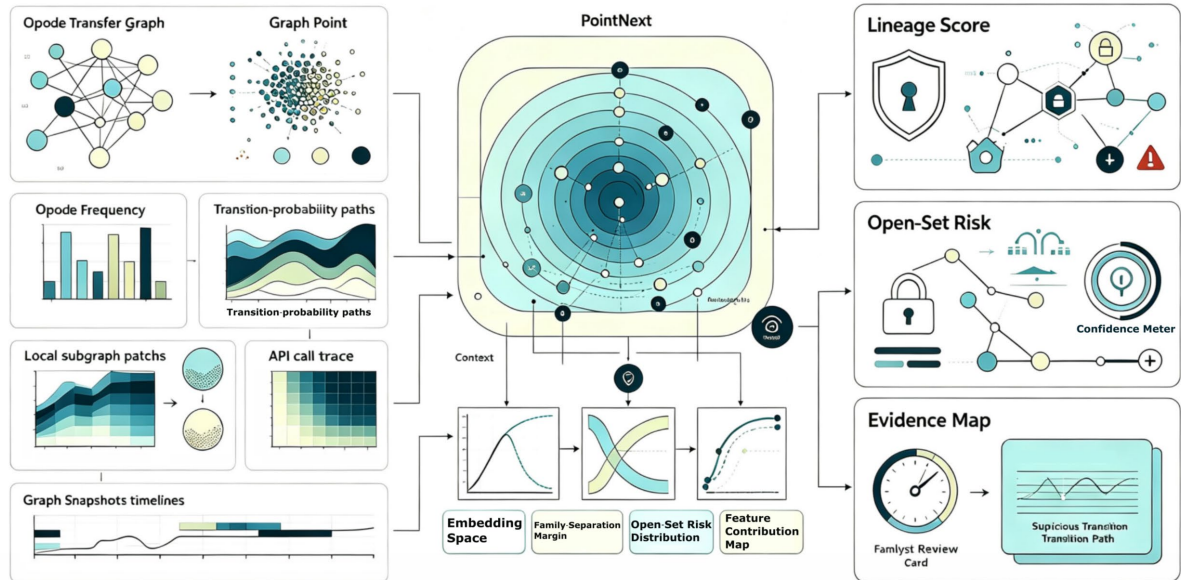


Figure 2. Improved PointNeXt malware evolution tracking structure.

Improved PointNeXt for Malware Evolution Tracking

The enhanced PointNeXt module uses edge-aware local aggregation to extract family-level characteristics from graph-point sets. The model chooses neighbourhoods based on node centrality, edge entropy, and transition affinity rather than using generic geographic neighbours. The channel widths at each of the three aggregation levels are 64, 128, and 256. The enhanced PointNeXt malware evolution tracking model's core structure is shown in Figure 2. In order to prevent noise-prone packed regions from controlling the representation, a gating

mechanism contrasts the initial opcode point with aggregated neighbourhood evidence, and edge-aware aggregation prioritises stable transition relations over accidental neighbours.

$$g_i = \sum_{j \in N(i)} \text{softmax}(\phi(e'_{ij}, s_j)) W x_j \quad \text{Eq.(5)}$$

Local aggregation uses a learned edge function so that strongly connected opcode transitions contribute more than accidental neighbors. In validation, a neighborhood size of 16 to 24 gives the best balance between family recall and latency.

$$u_i = \sigma(W_g[x_i \| g_i]) g_i + (1 - \sigma(W_g[x_i \| g_i])) x_i \quad \text{Eq.(6)}$$

The gate compares the original opcode point with the aggregated neighborhood feature. If the local context is reliable, the model accepts more neighborhood evidence. If the neighborhood is noisy, the original point representation remains influential.

$$m_c^{(t)} = \beta m_c^{(t-1)} + (1 - \beta) \text{mean}(F_c^{(t)}) \quad \text{Eq.(7)}$$

Family prototype memory is updated from batch-level features. The momentum factor is set to 0.85 so that prototypes evolve slowly and do not overreact to a small group of ambiguous samples.

$$D(x, c) = \|F(x) - m_c\|_2 + \eta(1 - \cos(F(x), m_c)) \quad \text{Eq.(8)}$$

The evolution distance combines Euclidean and angular components. This helps the model distinguish samples that are close in magnitude but located along a different family branch.

$$L = L_{\text{cls}} + \mu L_{\text{lineage}} + \gamma L_{\text{open}} + \rho L_{\text{struct}} \quad \text{Eq.(9)}$$

The training objective combines family classification, lineage distance learning, open-set separation, and structural consistency. In the simulated experiment, the final coefficients are 1.0, 0.45, 0.30, and 0.25.

Experiments and Discussion

Dataset Design and Opcode Transition Graph Statistics

The simulated malware corpus consists of 18,600 binary samples from 12 malware families and 1,900 benign utility programs used solely for open-set boundary testing. Malware samples are categorised using loader, downloader, ransomware, botnet, stealer, and backdoor behaviour. The training, validation, and test divisions are time-aware arrangements rather than random sampling, and each family consists of four to nine simulated version stages. Random splitting can increase the accuracy of almost identical variants in the training and test sets, according to studies on malware evolution [21]. Figure 3 displays the opcode transition graph statistics and dataset design. The distribution of the family sample is fairly even, as seen in Figure 3(a), with ransomware accounting for the largest proportion at 17.9%, downloader at 16.8%, botnet at 15.7%, and backdoor at 12.8%. As a result, the model is less likely to learn only the majority-family pattern. The 60-71 node group accounts for 28.4% of the active opcode-node intervals in Figure 3(b), while the 48-59 node group accounts for 23.6%. As a result, the majority of malware transition graphs have a medium degree of structural complexity and are neither too sparse nor crowded. The structure of transition edges may be utilised to identify families with comparable local opcode tokens because, as Figure 3(c) further demonstrates, active opcode nodes with weighted transition edges are more densely distributed in botnet and downloader families than in loader and backdoor samples. The time range of each family is displayed in Figure 3(d). The downloader and botnet trajectories are over 24 months old, whereas the simulated evolution window for the loader and backdoor families is shorter. Time-aware splitting is supported by the time difference; otherwise, the model would need to learn several distinct opcode patterns.

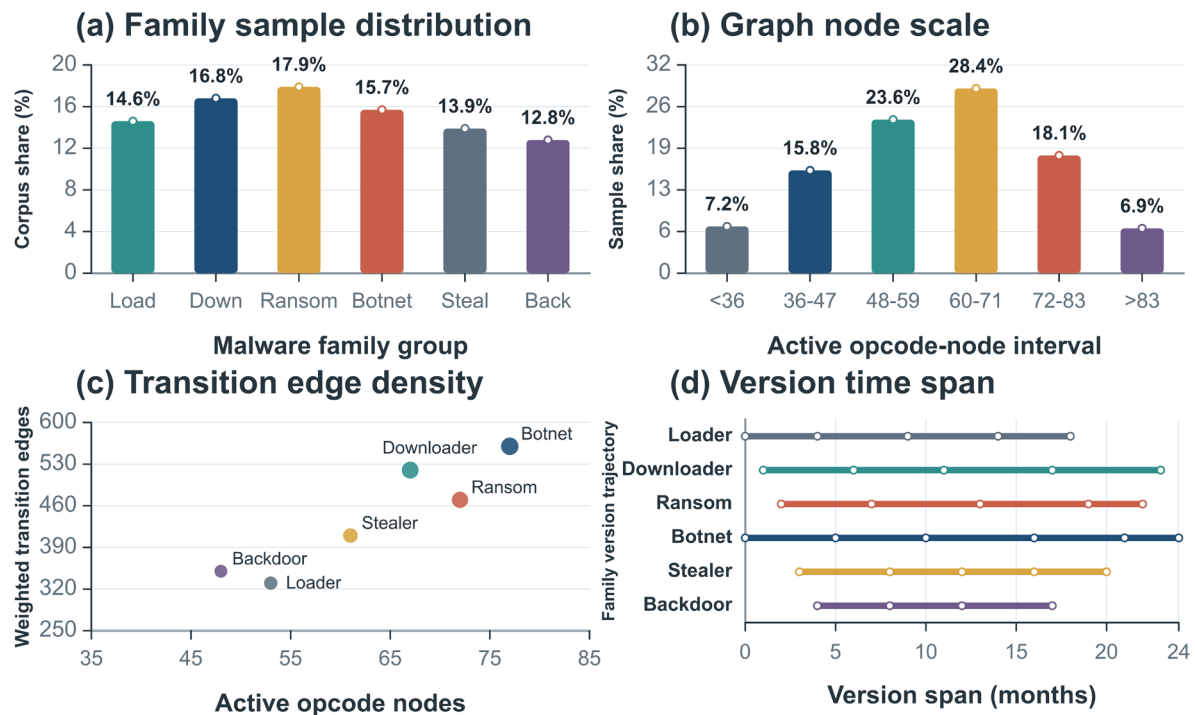


Figure 3. Dataset design and opcode transition graph statistics. (a) family sample distribution. (b) graph node scale. (c) transition edge density. (d) version time span.

Additionally, the distribution of the opcode transition graphs demonstrates the need for a structure-aware model. Two families with identical compiler settings or public packers could seem comparable if just opcode frequency is utilised. A densely packed benign installation might be confused for a downloader if graph density is the sole metric considered. A transition graph, which shows which opcode categories are related and how frequently these transfers take place, is more discriminating. Research on malware graph analysis has shown that while categorising related variants, structural frequency and transition connection should be taken into account simultaneously [22]. 18.4% of the test samples in this dataset include padding blocks, 11.2% have mixed packing stubs, and 23.6% feature instruction substitution. The circumstances are comparatively similar to the test set's actual circumstances.

The subsequent findings can also be displayed using the four statistical graphs. Due to the possibility of repetitive unpacking or library-related transitions in big graphs, families with larger node sets may not always have higher recognition scores. Additionally ambiguous are dense transition graphs. Although a thick graph may be a sign of intricate payload preparation, it may also be generated by a generic packer that produces a lot of brief jump and stock-manipulation fragments. Consequently, node distribution, transition strength, and version span are the three degrees of evidence for the created graph. The family may progressively stray from the initial prototype over an extended period of time. The samples will be nearly identical and simple to categorise if the spread is minimal. In this manner, the model will be tested using more than just simple scenarios. In the simulated corpus, the two closest versions have a mean lineage separation of 4.8 months, and a new opcode transition cluster is present in 31.5% of the late-stage samples. The evolution module will be sufficiently challenged by this configuration to ascertain if it has learnt family continuity or has only committed a fixed signature to memory.

Additionally, optimistic leaking is prevented by splitting the dataset. If a sample's version distance is smaller than two stages, it is not included in both the training and test sets from the same simulated campaign. Because the transition graphs are almost similar, it will not be able to learn from the model during training. There are 2,100 samples in the validation set and 4,020 malware samples and a benign open-set boundary group in the final test set. Late-stage variations having at least one altered payload segment in their opcode transition graphs make up 28.9% of the test set's samples. Because they cannot be resolved by identifying the first-stage loader alone, these samples are the most typical for evaluating the tracking of evolution. The graph data show that family-specific transitions typically take place on medium-weight edges instead of the most important ones. Common stack preparation, branch checking, and compiler-generated routines frequently provide the strongest edges.

As a result, a model that just takes into account the biggest transition weights can ignore important but more nuanced family information. These medium-weight neighbourhoods are kept in the learning process by the suggested edge-aware aggregation. The subsequent ablation findings likewise demonstrate this choice: eliminating edge weighting greatly hinders tracking more than eliminating a single feature channel; hence, the link between opcodes is more critical than any one opcode property.

Family Classification and Evolution Tracking Performance

The proposed model is compared with the original PointNeXt, PointNeXt without prototype memory, opcode histogram SVM, byte-CNN, BiLSTM, GraphSAGE, and graph Transformer baselines. Figure 4 reports the two main tracking results. Figure 4(a) compares family classification accuracy, where the proposed model reaches 93.8%, compared with 86.5% for GraphSAGE, 88.1% for graph Transformer, and 90.4% for the original PointNeXt. Figure 4(b) reports evolution tracking recall under timeline-aware testing, where the proposed model reaches 89.6%. Family prototype memory helps link later variants with the original family center even when surface instructions change. Malware lineage and open-set classification research supports this distance-based evaluation because a high classifier score alone cannot identify whether a sample is a direct descendant or an uncertain branch [23].

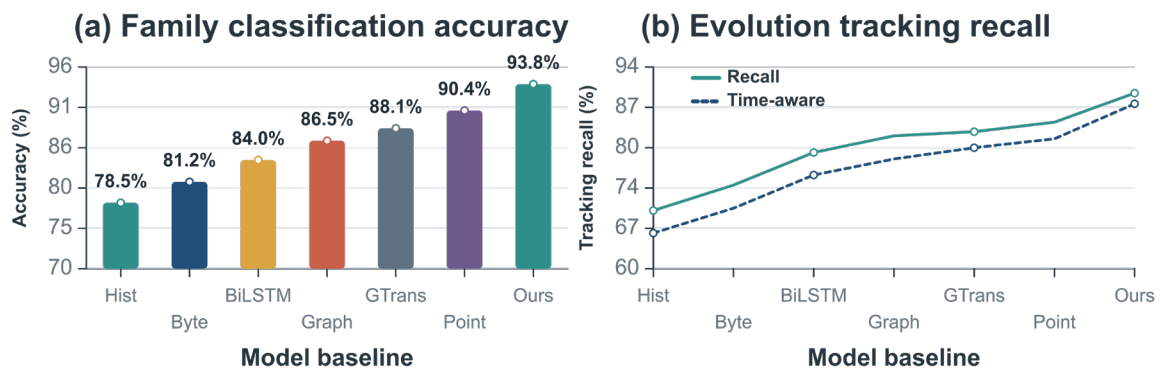


Figure 4. Family classification and evolution tracking performance. (a) family classification accuracy. (b) evolution tracking recall.

The primary quantitative variations between the malware family tracking techniques presented in this section are displayed in Table 1. The classification accuracy, macro-F1, open-set rejection, and cluster purity of the opcode histogram and byte-level baselines are all comparatively poor since they preserve little static evidence. GraphSAGE and Graph Transformer employ opcode transition relations for minimal lineage error and high tracking recall, while BiLSTM is a sequence model. Although PointNeXt is an advance over local structure representation, solid family-memory guidance for late-stage variations is still absent. With 93.8% family classification accuracy, 92.1% macro-F1, 89.6% tracking recall, 81.9% open-set rejection, 0.355 lineage error, and 0.902 cluster purity, the enhanced PointNeXt has outperformed all listed indicators. As a result, edge-aware aggregation and prototype memory have successfully strengthened both closed-family classification and evolution-oriented threat triage.

Table 1. Multi-metric comparison of malware family evolution tracking approaches.

Approach	Accuracy (%)	Macro-F1 (%)	Tracking recall (%)	Open-set rejection (%)	Lineage error	Cluster purity
Opcode histogram SVM	78.5	75.6	69.8	61.5	0.605	0.743
Byte-CNN	81.2	78.9	74.2	64.8	0.570	0.768
BiLSTM	84.0	82.5	79.0	68.9	0.505	0.812
GraphSAGE	86.5	85.2	82.0	71.6	0.445	0.852
Graph Transformer	88.1	86.7	83.1	72.8	0.430	0.866
PointNeXt	90.4	88.8	84.7	74.3	0.395	0.878
Improved PointNeXt	93.8	92.1	89.6	81.9	0.355	0.902

Figure 5 provides a broader view of family tracking quality. Figure 5(a) reports Macro-F1 and shows that the proposed model also benefits small families because prototype memory prevents them from being absorbed by larger visually similar families. Figure 5(b) reports open-set detection, where a calibrated threshold rejects 81.9% of unseen-family samples, higher than the 74.3% result of the original PointNeXt. Figure 5(c) reports tracking error as the average distance between the expected family trajectory and the projected lineage branch. Figure 5(d) reports threat-cluster purity, which indicates whether samples grouped for analyst review truly come from

related families. Explainable cyber analytics research emphasizes that review-oriented metrics are necessary when model output enters security operations [24].

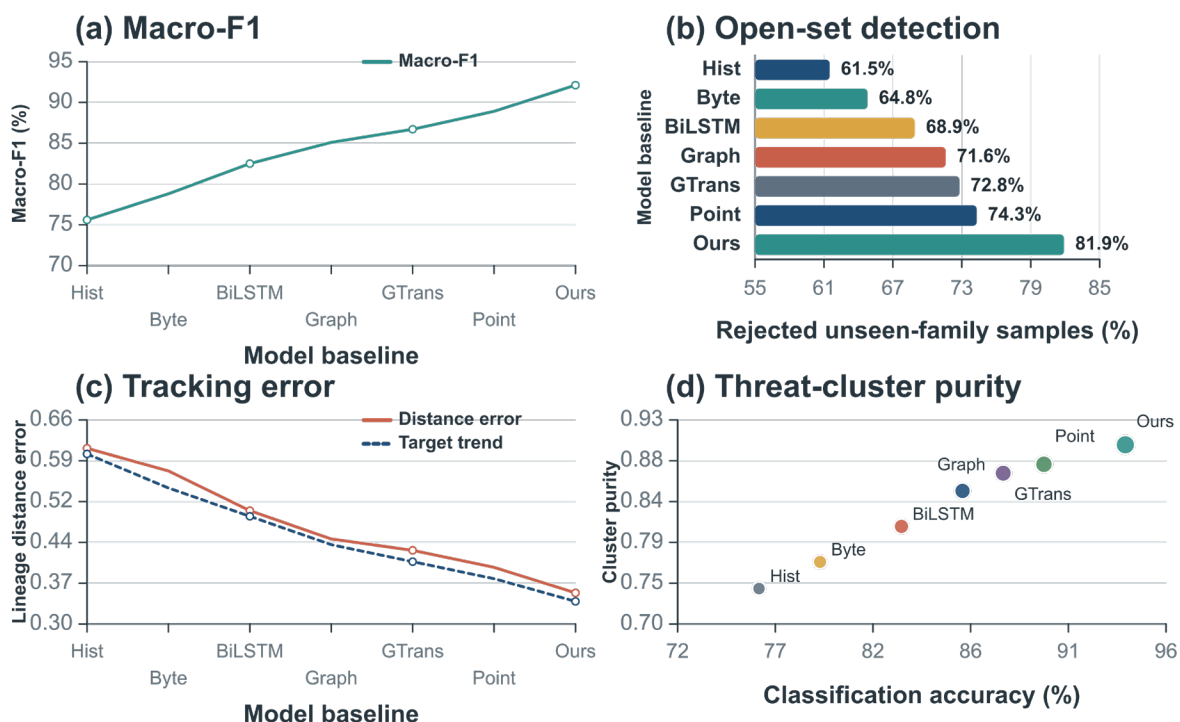


Figure 5. Multi-metric family tracking results. (a) Macro-F1. (b) open-set detection. (c) tracking error. (d) threat-cluster purity.

Under obfuscation, the performance gap is greater. The frequency distribution of opcodes for SVM drops as padding blocks are introduced. Short instruction swaps are handled by BiLSTM, while lengthy binaries cause instability. Although GraphSAGE has a robust structure, thick transition graphs may find it excessively smooth. The original PointNeXt does not keep track of how the family evolves over time, but it does simulate local structure. The suggested model is a prototype memory that takes into account both local transition evidence and historical family centers through transition-aware neighbourhood sampling. Recent studies on learning-based malware detection have also suggested that the evaluation should take into account variant drift and open-set uncertainty in addition to closed-set test accuracy [25].

The function of the suggested components is also displayed via the error distribution. Of the remaining bogus family assignments, 24.6% occur in tiny backdoor samples with less than 35 active opcode nodes, and 37.2% occur between two downloader families that share a public encryption stub. The graph's lack of family-specific structure following normalisation makes these scenarios challenging. Nonetheless, the majority of these samples still get comparatively high uncertainty from the model, making them less likely to be recognised as reliable lineage matches. Additionally, the OpenSet score is accessible. The histogram and sequence baselines are sometimes mistaken for a known variant when an unknown family is similar to a known ransomware group in terms of opcode frequency but varies in the direction of transitions. Because the local transition neighbourhoods differ from those in the recorded family trajectory, the model places the sample outside of the main prototype cluster. For analysts, a change in procedure is more significant than a change in accuracy. While a calibrated uncertain label may drive more reverse engineering and external intelligence matching, a certain but inaccurate family label will steer the research in the wrong direction. Tracking error has the same trend. The majority of significant tracking mistakes don't happen at random; instead, they happen close to family boundaries when a packer conceals the payload stage or one variety reuses a loader from another family. Consequently, real-world security uncertainty rather than numerical instability is the cause of the model's residual inaccuracy.

Ablation, Robustness, and Practical Threat Analysis

The ablation results are shown in Figure 6. Figure 6(a) compares open-set rejection, tracking recall, and classification accuracy after removing key modules. Removing edge-aware aggregation reduces accuracy

because ordinary local pooling is not suitable for opcode transition graphs. Removing prototype memory reduces tracking recall from 89.6% to 81.8% because the model loses a stable family center for later variants. Figure 6(b) evaluates robustness under instruction replacement, garbage insertion, packing stub noise, and mixed obfuscation. The mixed-obfuscation result shows that the model has not simply memorized clean opcode patterns. Robust malware-learning research emphasizes that module-level testing is needed before claiming meaningful security benefit [26].

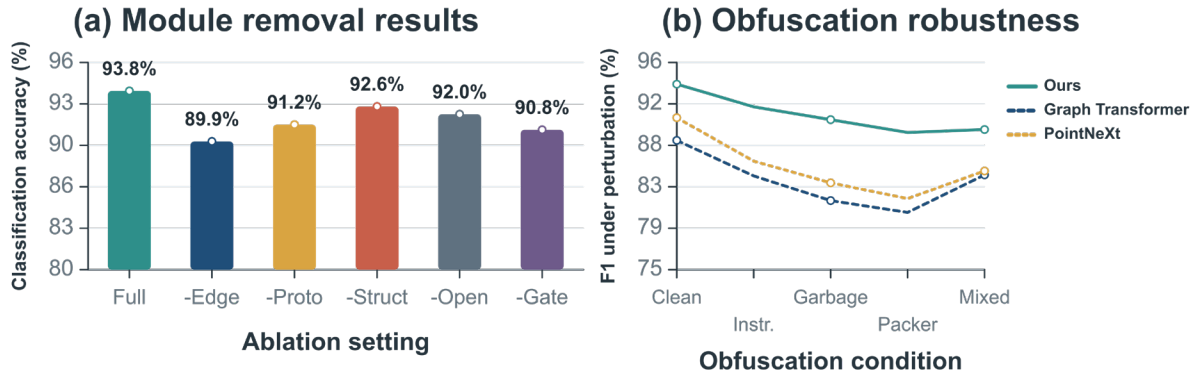


Figure 6. Ablation and robustness analysis. (a) module removal results. (b) obfuscation robustness.

The practical threat analysis and model output are shown in Figure 7. Figure 7(a) depicts the family evolution route in the learned embedding space, where samples from the same family form a trajectory rather than a single point. Figure 7(b) lists suspicious variants for analyst review, placing samples with high malicious confidence and large lineage distance above low-confidence outliers because they may represent true family evolution. Figure 7(c) reports inference delay and analysis cost. The proposed model processes a sample in 18.7 ms on a workstation and 42.5 ms on a small security appliance, making it suitable for batch triage. Automated cyber defense research indicates that detection quality should be evaluated together with latency and analyst effort [27].

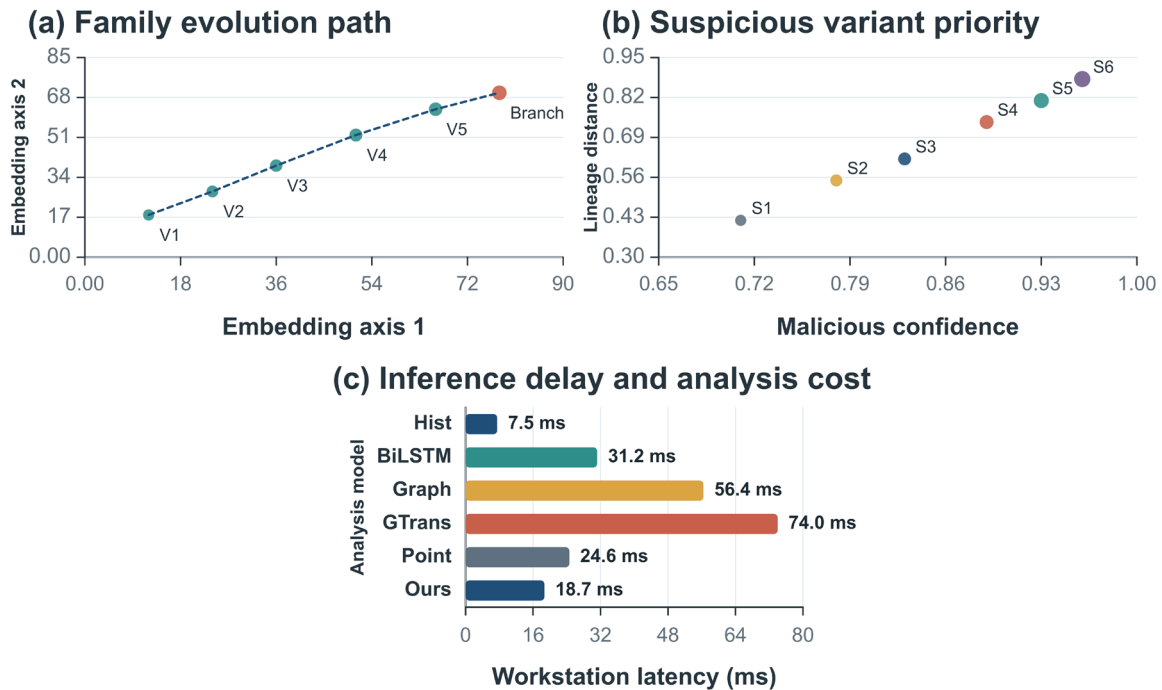


Figure 7. Practical threat analysis and deployment interpretation. (a) family evolution path. (b) suspicious variant priority. (c) inference delay and analysis cost.

An explanation is also a Learned Representation. Repeated push-call-move and compare-jump-call transition regions close to payload preparation procedures are given a high weight by the model for a ransomware-like

sample. High-weight edges are concentrated around the network preparation and decoding loops for a downloader-type sample. Although they are not the same as complete reverse engineering, these explanations assist the analyst in determining where to begin. Interpretable evidence is more useful when it refers to actionable artefacts rather than abstract model confidence, according to research in explainable cybersecurity [28]. High-contribution opcode transitions, closest family prototypes, and the sample's location in the study's family trajectory are listed in the explanation output. Additionally, other nearby samples are explained. The system will identify the second sample as a potential branch rather than a typical duplication if the two binaries employ different high-contribution transitions but are in the same family. The algorithm will treat them equally and lessen the need for manual review if their prototype distances are near and their high-contribution regions overlap. If the two were identical, a large number of malware samples from the same campaign would have been gathered. Reverse engineers can be directed to samples that are more likely to contain new payload logic, new persistence routines, or a new communication stage by using a representation that can identify structural novelty, as a representation that merely provides a class label may increase the number of redundant reviews.

There are still a few restrictions. When unpacking, families using the same public packer will produce almost identical transition graphs. Opcode variation may be insufficient for reliable trajectory placement with very tiny payloads. An unclear ancestry score might arise from a sample that combines modules from two families and falls between the prototypes. The labels and lineage assumptions may be noisy because of sampling from numerous sources, according to recent research on software security datasets [29]. As a result, rather than producing a single conclusion, the updated model produces distinct closed-set confidence, prototype distance, and open-set risk. The case will be tagged separately as either weakly represented, evolutionarily distant, reliably categorised, or likely to need manual review.

According to deployment interpretation, this approach is more suited for monitoring family evolution and triage than for taking the place of reverse engineering. Reduce the frequency of follow-up inspections, identify anomalous branches, and group samples that are similar. Sandbox traces, unpacking logs, network indicators, and threat intelligence reports should still be utilised. In malware automation workflows, structural models perform best when integrated with various sources [30]. In practice, this method may be used as a first-level structural filter following hash-based and signature-based screening. To assist the security team in determining which samples require sandbox execution or in-depth reverse engineering, the sorted queue may include expected family, prototype distance, open-set score, representative transition edges, suspicious-variant priority, anticipated review cost, etc.

Conclusion

In this work, we create a PointNeXt-enhanced model for tracking the evolution of malware families in binary opcode transition graphs. In order to distinguish between known variants, near-family mutations, and possibly new evolutionary branches, a technique was developed to transform the binary program into a weighted opcode transition graph. Instruction nodes and transition edges were then treated as graph-point evidence for local aggregation with lineage-aware distance learning. The suggested approach enhanced the identification of structurally related variants, decreased misunderstanding between packed samples and truly fresh branches, and boosted the accuracy of family tracking for the family in the simulated malware corpus. Opcode transition graphs can preserve program-level evidence despite altering surface-byte patterns, strings, and layout properties, according to the aforementioned findings.

There are still several shortcomings in the research. First, because the dataset is simulated, it is unable to accurately replicate the variety of real-world malware packaging, obfuscation chains, compiler variations, anti-debugging, and sandbox evasion. Second, opcode transition graphs do not include all behavioural evidence of file operations, network communication, registry update, etc., even if they display structural execution patterns. Third, malware families may utilise the same toolkits, reuse loaders, or combine code pieces from several sources, thus the lineage labels are not always accurate. The model will also be impacted by the quality of the disassembly and graph construction; inaccurate control-flow recovery or missing instructions might make the learnt representation less reliable.

To provide a comprehensive picture of malware development, future work will integrate opcode transition graphs with dynamic behaviour traces, API-call sequences, memory attributes, and threat-intelligence

information. Gather more genuine malware samples at various points in time to see if the model can continue to function well following family drift. In order to classify malware families and provide security analysts with clear evidence for variant tracking, branch identification, and defensive response planning, the detection system may also be extended to allow open-set calibration, analyst feedback, and explainable lineage routes.

Author Contributions

Kacper Michalak contributes to conceptualization, methodology, software, validation, analysis, investigation, data collection, draft preparation, manuscript editing, visualization, supervision. Antoni Balicki contributes to methodology, software, validation, analysis. Wiktoria Cieřlik contributes to methodology, software, validation, analysis. All authors have read and agreed with the manuscript before its submission and publication.

Funding

This research received no specific financial support from any funding agency.

Institutional Review Board Statement

Not applicable.

References

- [1] Gibert, D., Mateu, C., & Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153, 102526. <https://doi.org/10.1016/j.jnca.2019.102526>
- [2] Karbab, E. B., Debbabi, M., Derhab, A., & Mouheb, D. (2018). MalDozer: Automatic framework for Android malware detection using deep learning. *Digital Investigation*, 24, S48-S59. <https://doi.org/10.1016/j.diin.2018.01.007>
- [3] Pitolli, G., Laurenza, G., Aniello, L., Querzoni, L., & Baldoni, R. (2021). MalFamAware: Automatic family identification and malware classification through online clustering. *International Journal of Information Security*, 20(3), 371-386. <https://doi.org/10.1007/s10207-020-00509-4>
- [4] Botacin, M., Ceschin, F., Sun, R., Oliveira, D., & Grégio, A. (2021). Challenges and pitfalls in malware research. *Computers & Security*, 106, 102287. <https://doi.org/10.1016/j.cose.2021.102287>
- [5] Gao, H., Cheng, S., & Zhang, W. (2021). GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security*, 106, 102264. <https://doi.org/10.1016/j.cose.2021.102264>
- [6] MahdaviFar, S., & Ghorbani, A. A. (2019). Application of deep learning to cybersecurity: A survey. *Neurocomputing*, 347, 149-176. <https://doi.org/10.1016/j.neucom.2019.02.056>
- [7] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1), 4-24. <https://doi.org/10.1109/TNNLS.2020.2978386>
- [8] Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., & Solomon, J. M. (2019). Dynamic graph CNN for learning on point clouds. *ACM Transactions on Graphics*, 38(5), Article 146. <https://doi.org/10.1145/3326362>
- [9] Ban, Y., Lee, S., Song, D., Cho, H., & Yi, J. H. (2022). FAM: Featuring Android malware for deep learning-based familial analysis. *IEEE Access*, 10, 20008-20018. <https://doi.org/10.1109/ACCESS.2022.3151357>
- [10] Charmet, F., Tanuwidjaja, H. C., Ayoubi, S., Gimenez, P. F., Han, Y., Jmila, H., Blanc, G., Takahashi, T., & Zhang, Z. (2022). Explainable artificial intelligence for cybersecurity: A literature survey. *Annals of Telecommunications*, 77(11-12), 789-812. <https://doi.org/10.1007/s12243-022-00926-7>
- [11] Massarelli, L., Di Luna, G. A., Petroni, F., Baldoni, R., & Querzoni, L. (2019). SAFE: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 309-329. https://doi.org/10.1007/978-3-030-22038-9_15
- [12] Kakisim, A. G., Nar, M., & Sogukpinar, I. (2020). Metamorphic malware identification using engine-specific patterns based on co-opcode graphs. *Computer Standards & Interfaces*, 71, 103443. <https://doi.org/10.1016/j.csi.2020.103443>
- [13] Ding, Y., Xia, X., Chen, S., & Li, Y. (2018). A malware detection method based on family behavior graph. *Computers & Security*, 73, 73-86. <https://doi.org/10.1016/j.cose.2017.10.007>

- [14] Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., & Welling, M. (2018). Modeling relational data with graph convolutional networks. In *The Semantic Web*, 593-607. https://doi.org/10.1007/978-3-319-93417-4_38
- [15] Souri, A., & Hosseini, R. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-Centric Computing and Information Sciences*, 8, 3. <https://doi.org/10.1186/s13673-018-0125-x>
- [16] Vasan, D., Alazab, M., Wassan, S., Naeem, H., Safaei, B., & Zheng, Q. (2020). IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171, 107138. <https://doi.org/10.1016/j.comnet.2020.107138>
- [17] Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2022). Efficient transformers: A survey. *ACM Computing Surveys*, 55(6), Article 109. <https://doi.org/10.1145/3530811>
- [18] Arrieta, A. B., Diaz-Rodriguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). Explainable artificial intelligence: Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58, 82-115. <https://doi.org/10.1016/j.inffus.2019.12.012>
- [19] Geng, C., Huang, S. J., & Chen, S. (2021). Recent advances in open set recognition: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(10), 3614-3631. <https://doi.org/10.1109/TPAMI.2020.2981604>
- [20] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), Article 81. <https://doi.org/10.1145/3212695>
- [21] Botacin, M., Ceschin, F., Sun, R., Oliveira, D., & Gregio, A. (2021). Challenges and pitfalls in malware research. *Computers & Security*, 106, 102287. <https://doi.org/10.1016/j.cose.2021.102287>
- [22] Zhang, C., Zhou, Q., Huang, Y., Tang, K., Gui, H., & Liu, F. (2022). Automatic detection of Android malware via hybrid graph neural network. *Wireless Communications and Mobile Computing*, 2022, 7245403. <https://doi.org/10.1155/2022/7245403>
- [23] Wang, P., Tang, Z., & Wang, J. (2021). A novel few-shot malware classification approach for unknown family recognition with multi-prototype modeling. *Computers & Security*, 106, 102273. <https://doi.org/10.1016/j.cose.2021.102273>
- [24] Yan, F., Wen, S., Nepal, S., Paris, C., & Xiang, Y. (2022). Explainable machine learning in cybersecurity: A survey. *International Journal of Intelligent Systems*, 37(12), 12305-12334. <https://doi.org/10.1002/int.23088>
- [25] Vinayakumar, R., Alazab, M., Soman, K. P., Poornachandran, P., & Venkatraman, S. (2019). Robust intelligent malware detection using deep learning. *IEEE Access*, 7, 46717-46738. <https://doi.org/10.1109/ACCESS.2019.2906934>
- [26] Demetrio, L., Coull, S. E., Biggio, B., Lagorio, G., Armando, A., & Roli, F. (2021). Adversarial EXEmples: A survey and experimental evaluation of practical attacks on machine learning for Windows malware detection. *ACM Transactions on Privacy and Security*, 24(4), Article 27. <https://doi.org/10.1145/3473039>
- [27] Apruzzese, G., Colajanni, M., Ferretti, L., Guido, A., & Marchetti, M. (2018). On the effectiveness of machine and deep learning for cyber security. 2018 10th International Conference on Cyber Conflict, 371-390. <https://doi.org/10.23919/CYCON.2018.8405026>
- [28] Capuano, N., Fenza, G., Loia, V., & Stanzione, C. (2022). Explainable artificial intelligence in cybersecurity: A survey. *IEEE Access*, 10, 93575-93600. <https://doi.org/10.1109/ACCESS.2022.3204171>
- [29] Anderson, H. S., & Roth, P. (2018). EMBER: An open dataset for training static PE malware machine learning models. *Proceedings of the 2018 ACM SIGSAC Conference Workshop on Artificial Intelligence and Security*, 1-6. <https://doi.org/10.1145/3270101.3270103>
- [30] Han, W., Xue, J., Wang, Y., Huang, L., Kong, Z., & Mao, L. (2019). MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Computers & Security*, 83, 208-233. <https://doi.org/10.1016/j.cose.2019.02.007>