$MGAP^3$: Malware Group Attribution Based on PerceiverIO and Polytype Pre-training

Yuxia Sun, *Member, IEEE,* Shiqi Chen, Song Lin, Aoxiang Sun, Saiqin Long, and Zhetao Li, *Member, IEEE*

Abstract—The escalating prevalence of Advanced Persistent Threat (APT) malware demands more effective methods to accurately attribute malware to specific APT groups. Traditional manual attribution processes are labor-intensive and error-prone, while existing automated methods are hampered by small dataset sizes, inadequate representation learning, and poor noise reduction during preprocessing. To address these challenges, we introduce the AMG25 dataset, which expands the pool of malware samples labeled with APT group affiliations. Concurrently, we propose the MGAP³ model (Malware Group Attribution based on PerceiverIO and Polytype Pre-training), which enhances attribution performance by incorporating hierarchical pre-training for disassembled codes and leveraging multi-view statistical features, all within a unified PerceiverIO architecture. This model adeptly captures complex program structures and interactions cross multiple code granularities, through a series of innovative polytype pre-training tasks. Additionally, we have developed a novel noise filtering technique that focuses on user-defined function codes, substantially reducing overfitting and boosting performance. Furthermore, a streamlined version of the model, MGAP³-Lite, has been developed to accelerate training while preserving robust performance. Extensive experiments have validated the effectiveness of our models and underscored the importance of the proposed pre-training technique.

Index Terms—APT malware, group attribution, PerceiverIO, pre-training task, static analysis

1 Introduction

As is widely recognized, Advanced Persistent Threat (APT) attacks have become a significant cybersecurity concern for global organizations, governments, and individuals [1]. APT groups carry out targeted attacks by deploying malicious software covertly, often driven by political, economic, or strategic motives. Attributing the malicious software discovered in APT attacks can help identify the responsible entities behind the attacks. This attribution is essential not only for better detection and defense against future attacks, but also for holding malicious actors accountable, as well as facilitating global cooperation among security professionals to combat the ever-evolving cybersecurity threats [2]. However, attributing APT malware poses significant challenges. APT attackers frequently employ various obfuscation techniques to evade analysis and hide their identities, making the attribution task complex and error-prone [3], [4]. Manual APT attribution is labor-intensive, time-consuming, and requires security experts with a high level of expertise and experience. In recent years, with the increasing frequency and sophistication of APT attacks, there is an urgent need in the cybersecurity field to develop effective automated APT malicious software attribution techniques.

Over the years, significant research has been conducted on the automatic group attribution of APT malware. These attribution approaches can be broadly categorized into dynamic and static methods. Dynamic methods [5], [6], [7], [8], [9], [10] involve tracking and analyzing the runtime behaviors of APT malware, while static methods [11], [12], [13], [14], [15], [16] directly analyze the malware codes. However, dynamic approaches heavily rely on virtual execution environments, which can make them time-consuming and less effective when dealing with malware that employs virtual machine escape techniques. In contrast, static approaches can overcome these limitations as they do not require executing the malware [17]. Therefore, this paper focuses on developing static group attribution techniques for APT malware. Early static attribution works [11], [12], [13] extract features from malware PE-files and utilize traditional machine learning models for group classification. In recent years, deep learning techniques have been increasingly employed to attribute malware functions or classify malware into groups [14], [15], [16]. For example, a state of art malware attribution method known as BinMLM [16] extracts n-gram opcode sequences from malware disassembly codes, pre-trains an LSTM encoder for each APT group, conducts joint training to generate malware encoding vectors, and subsequently attributes malware instances based on vector similarity. To generate high-quality malware code representations, pre-training-based malware attribution methods typically pre-train a sequential model on a large-scale external corpus, followed by fine-tuning

- *Yuxia Sun is with the College of Information Science and Technology, Guangdong Key Laboratory of Data Security and Privacy Preserving, Jinan University, Guangzhou 510632, China. Email: tyxsun@email.jnu.edu.cn.*
- *Shiqi Chen, Song Lin, Saiqin Long, and Zhetao Li are with the College of Information Science and Technology, Jinan University, Guangzhou 510632, China. Email: {shiqichen2000, simplesen960813}@gmail.com, saiqinlong@jnu.edu.cn, liztchina@hotmail.com.*
- *Aoxiang Sun is with the College of Information and Computational Science, Jilin University, Changchun, China. Email: 2835604014@qq.com.*

Manuscript received July 28, 2023; revised XX XX, 20XX.
This work was supported by National Key Research and Development Program of China (Grant 2021YFB3101201), National Natural Science Foundation of China (Grant No. 62032020 and No. U23B2027), Guangdong Key Laboratory of Data Security and Privacy Preserving (Grant No. 2017B030301004). (Corresponding author: Zhetao Li)

this model on a local dataset. For instance, BinMLM pre-trains an LSTM model on an extensive binary file corpus [16], while SecureBERT_Mal utilizes a large malware dataset to fine-tune the SecureBERT model that has been pre-trained on a comprehensive security text corpus [41], [42].

However, current group attribution approaches for APT malware [5]-[16] are constrained by dataset size, representation learning, and data preprocessing techniques, which result in suboptimal attribution capabilities. Next, we will detail these challenges and further outline how we overcome them in this paper to enhance malware attribution:

Firstly, the datasets used in the existing malware attribution approaches are limited in size and not publicly accessible. For instance, the only openly accessible dataset, APTMalware [49], comprises merely 12 APT groups and approximately 3,000 malware MD5s. To address this issue, we introduce a novel dataset of malware with APT group labels, which is larger in size compared to APTMalware.

Secondly, malware attribution approaches often utilize pre-training techniques to enhance code representation, yet they face several challenges: (1) a reliance on a singular mask-based pre-training task similar to BERT's MLM (Masked Language Model), which fails to capture complex program structures and interactions at different syntactic levels; (2) the use of LSTM or Transformer architectures, which not only struggle to manage complex and long-distance dependencies within code sequences but also incur high computational costs [18], [19], [23]; (3) a high dependence on large external corpora, necessitating extensive efforts to build such databases. To overcome these obstacles, we propose a novel malware code representation technique using hierarchical pre-training tasks and the PerceiverIO architecture [26]. This approach extracts features at three different levels of abstraction (i.e., instruction, block, and function). Inspired by the varied pre-training tasks in NLP [24], [39], our approach employs 17 polytype pre-training tasks to learn intricate relationships between various granular code elements (e.g., opcodes, operands, blocks, functions). Our PerceiverIO-based model, equipped with mechanisms like cross-attention, effectively handles complex and long-distance dependencies within a fixed-size latent space, and reduces computational demands. This approach significantly boosts the performance of attribution models, even with a small local training set containing only a few thousand samples, thus eliminating the need for large external corpora.

Thirdly, existing malware attribution methods typically analyze single-modal code data, such as n-gram opcode sequences, visualized images of code features, or images of code bytes. To fully leverage the complementarity and integration of multimodal data to enhance attribution performance, developing a multimodal malware attribution technique is particularly crucial. In this context, this paper introduces a novel multimodal malware attribution model to integrate two distinct data modalities: the disassembled text and statistical feature images of malware, based on the PerceiverIO architecture [26]. We employ unified PerceiverIO architecture for both modalities, not only enhancing the internal consistency of the data representations but also facilitating potential synergies during the representation fusion process, ultimately leading to significant improvement in model performance.

Finally, in malware codes, the implementation of user-defined functions (UDFs) typically reflect the unique programming habits or intentions of specific attack groups, featuring distinct techniques like unique encryption or persistence mechanisms. In contrast, the implementation of non-user-defined functions (non-UDFs), such as standard library functions or system calls, is typically generic and lack specific group identifiers. Thus, for a malware attribution model, the code implementing non-UDFs can be viewed as noise. However, existing attribution models typically use data including this noise, potentially leading to overfitting on irrelevant code features and unnecessary data analysis. To address these challenges, we introduce a novel UDF filtering technique that enhance the focus on user-defined code sections more indicative of specific APT groups, thus improving classification performance and processing efficiency. Our technique targets only the implementation code of non-UDFs while preserving calls to these functions, maintaining crucial contextual information to boost model effectiveness.

In summary, this paper makes the following key contributions to advancing the field of malware attribution:

- **Dataset:** We introduce the AMG25[1] (APT Malware with Group-label) dataset, a new public resource with 5,188 malware samples from 25 APT groups, significantly expanding available resources and providing a more comprehensive foundation for further research compared to existing datasets.
- **Attribution Approach:** We propose a novel approach for malware group attribution named $MGAP^3$ (Malware Group Attribution based on PerceiverIO and Polytype Pre-training). This method effectively filters out code noise from non-UDFs, generates textual representations via hierarchical pre-training, extracts statistical representations from multi-view features, and integrates the multimodal code data, all utilizing the PerceiverIO architecture.
- **Pre-training Tasks:** We introduce innovative hierarchical code pre-training tasks, categorized into instruction-related, block-related, and function-related tasks. These enhance assembly-like code representations by delving into complex program structures and semantic relationships among code elements.
- **Filtering Strategy:** We propose a unique filtering technique that targets non-UDFs in the code to reduce noise, enhancing model performance and preventing overfitting.
- **Implementation & Validation:** We implement the full $MGAP^3$ model and its streamlined variant, $MGAP^3$-Lite, and validate their attribution effectiveness through extensive experiments. Additionally, we demonstrate the importance of key components of the model and the proposed pre-training tasks through experimental results.

The remaining sections of this paper are structured as follows. Section 2 provides an overview of the related

1. https://https://github.com/Yuxia-Sun/MGAP-AMG25

work in the field. Section 3 presents our proposed $MGAP^3$ method. In Section 4, we discuss the experimental setup and results. Section 5 discusses the limitations of our malware attribution approach. Finally, Section 6 concludes the paper and outlines future research directions.

2 Related Work

2.1 Malware group attribution

Automatic group attribution of APT malware, as a critical task of malware analysis and APT attack tracing, has been studied by many researchers in recent years. The group attribution approaches can be categorized into dynamic and static ones. The existing dynamic attribution methods [5]-[10] track and analyze the runtime behaviors of APT malware, such as process activities and network operations [5], [6], and the actual sequence of function calls [7]-[10]. However, dynamic approaches heavily depend on a virtual execution environment, such as a sandbox or virtual machine, to capture malware behavior data. Consequently, these dynamic methods tend to be time-consuming and ineffective when malware utilizes virtual machine escape techniques. Static attribution approaches can overcome the above limitations of dynamic ones by analyzing malware codes directly without executing them. As early works of APT malware group attribution, Laurenza et al. and Rosenberg et al. extract static features from malware PE-files over such file properties on PE-file header, import tables, section tables, and so on, and utilize traditional machine learning models as group classifiers [11], [12]. Liang et al. extract malware PE-file features of entry point, DLL (Dynamic Link Library), resource, and the number of sections, and train an RF (Random Forest) classifier for group attribution [13]. Wei et al. statically extract API system calls as malware features, and generate the vector representation of features based on LSTM and attention mechanism [14]. To attribute malware functions (rather than malware) to APT groups, Liu et al. extract word feature sequences from the disassembly code of each function's control flow graph, and generate the function embedding using an lstm model to learn the dependencies in the code segment [15]. Recently, Song et al. analyze the disassembly codes of APT malware, and extract consecutive opcode sequences as input of a group classifier, called BinMLM, to classify APT malware into 10 groups [16]. As the most recent static attribution model, BinMLM utilizes an LSTM encoder for each APT group and a gate layer for each malware sample to obtain malware encoding vectors, trains a decoder for each APT group, and finally classifies malware samples into APT groups based on vector similarity [16].

The aforementioned group attribution approaches of APT malware enable automated classification of malware into different groups. However, these approaches have limitations, including limited publicly available datasets, limited dataset size, noise interference from non-user-defined functions, reliance on LSTM-based models to learn representations of sequential data, and separate use of sequential code text or various code features rather than merging them into one representation.

2.2 Pre-trained model

Various deep learning-based sequence models, such as RNN, Encoder-Decoder[20], LSTM[21], GRU[22], Transformer[23], and more, have the potential to generate profound representations of sequential data, such as malware codes. However, it is worth noting that, currently, no pre-training techniques have been applied to generate malware representations specifically for APT malware attribution.

Compared with traditional recurrent neural networks (such as LSTM), Transformer has a self-attention and parallel training mechanism, which can capture long-distance dependencies better and have good performance when processing long sequences. Currently, Transformer is the most concerned sequence model and finds extensive application in tasks like NLP. One notable example is the renowned BERT model, which utilizes a multi-layer Transformer as its encoder [24]. In 2021, Jaegle et al. proposed the Perceiver model [25] and the PerceiverIO model [26] that can replace Transformer. The series of Perceiver models retain the self-attention and residual connections of the Transformer, and then use the cross-attention mechanism to convert high-dimensional input data into a fixed-dimensional latent space [25]. Compared with Perceiver, PerceiverIO is more general and can process multi-modal input, such as text and image, and perform better in language understanding and visual understanding [26]. Furthermore, PerceiverIO outperforms Transformer-based BERT on the GLUE language benchmark [26]. In view of the above advantages of PerceiverIO, this paper will construct a pre-training model of malicious code by stacking PerceiverIO encoders, that is, a code text representation model.

2.3 Pre-training task

The pre-training task is usually a self-supervised learning task in which the model is trained with unlabeled samples. The model learns the internal structural information and semantic relationships of the data by accomplishing the goal of the pre-training tasks, thereby generating a generic data representation. In 2018, Google introduced two pre-training tasks for BERT: MLM (Masked Language Model) and NSP (Next Sentence Prediction) [24]. They predict relationships between masked words or sentence pairs. BERT and its pre-training tasks have achieved success in NLP and computer vision [28], [29], [30]. Recently, in the field of source-code analysis, some works have been conducted on pre-training tasks over source codes [31], [32], [33]. Wan et al. provide evidence suggesting that integrating the syntax structure of code into the pre-training process could enhance code representations [33]. Guo et al. extracted the data flow diagram of the source code, and proposed the following three pre-training tasks: masked task, edge detection task, and pairing detection task of source code variables and data flow diagram variables [31]. Bui et al. created an abstract syntax tree(AST) for source codes and proposed a pre-training task to predict the subtree structure of the AST [32]. However, these source-code-oriented pre-training tasks are not suitable for malware for which source code is not available.

Currently, there have been some works on pre-training tasks in the realm of malware analysis [34]-[38], which

aim to train a binary classifier for malware detection or a multiple-way classifier for malware families, rather than a classifier for APT groups. The MalBERT model proposed by Rahali et al. directly uses a malware dataset to fine-tune BERT's parameters [34]. This makes the corpus used for MalBERT's pre-training task not a malware corpus. In order to detect Windows malware, Xu et al. extracted the API call sequence of malware and defined two API-related pre-training tasks, namely the API mask task and the next API prediction task, for pre-training the Malbert model based on Transformer [35]. In order to make family classification of malware, Wu et al proposed a pre-training task NBP (Next Block Prediction) to encourage the model to learn the relationship of basic blocks [36]. For the detection and family classification of malware, Alvares et al. used BERT to generate word embedding for each opcode of the malware sample [37]. Oak et al. used BERT to generate an embedding of an Android program's API call as an input to a malware detector [38].

In contrast to the aforementioned approaches, we will introduce and utilize a novel set of pre-training tasks that focus on capturing the syntax and syntactic representations of disassembly codes in programs.

3 $MGAP^3$ Approach

To elevate the performance of malware group attribution, we introduce a novel approach named $MGAP^3$ (Malware Group Attribution based on PerceiverIO and Polytype Pre-training) in this section. $MGAP^3$ improves the preprocessing and representation learning of malware codes, addressing the limitations of current attribution methods: (1) To mitigate code noise from non-UDF implementations, we developed a UDF identifier to filter out this noise. This preprocessing strategy aims to sharpen the focus on UDF code sections, which are more indicative of specific APT groups, thereby enhancing attribution performance and processing efficiency. (2) To overcome the challenges associated with singular pre-training tasks and less effective architectures in traditional pre-training for code representation, we introduce a suite of hierarchical pre-training tasks coupled with a PerceiverIO-based model. These tasks are designed to deeply understand the program structure and the complex interactions among various code elements at different granularities and levels. Our model employs PerceiverIO encoders to efficiently capture complex, long-range dependencies within code sequences, thereby optimizing computational performance. (3) To address the limitations of unimodal code data, we combine malware representations from disassembled text and statistical feature images, both processed using PerceiverIO encoders. This unified PerceiverIO architecture aims to enhance data consistency and fosters synergies during fusion, ultimately boosting model performance.

Fig. 1 illustrates the architecture of our $MGAP^3$, which takes APT malware samples in binary codes as input and generates the predicted APT group label as the output. The $MGAP^3$ framework consists of four modules: (1) Pre-processing module, which initially acquires the assembly code of a malware sample, and then transforms it into specified pseudo-code text; (2) Statistical representation module, which generates deep representations of statistical features extracted from binary and assembly malware code; (3) Text-representation module, which generates malware text representations based on the pseudo-code; (4) Representation-fusion & Group-attribution module, which fuses malware text and statistical representations into the final representation, and classifies the malware into an APT group. Our $MGAP^3$ approach firstly runs the pre-processing module, then executes the statistical representation module and the text-representation module in parallel, and finally performs the Representation-fusion & Group-attribution module. In the following four subsections, we will provide detailed explanations of each of these four modules/steps.

3.1 Pre-processing module

As shown in Fig. 1, the pre-processing module of $MGAP^3$ firstly converts the binary code of a malware instance into its disassembly code using a disassembly tool such as IDA pro, and then normalizes the assembly code. The pre-processing module also attempts to filter out the non-user-defined functions (i.e., non-UDFs) in the code to reduce potential noise from non-UDF code when analyzing the characteristics of user code.

3.1.1 Normalizing code text

The pre-processing module of $MGAP^3$ normalizes the assembly code of each malware sample into pseudo-assembly text, as illustrated in Fig. 2, through the following key process:

1) **Normalize sample-size, function-size & block-size:** For each malware sample, the module establishes predefined constants (denoted as S, F, and B, respectively) for sample size (i.e., the total number of functions in each sample file), function size (i.e., the total number of blocks in each function), and block size (i.e., the total number of instructions in each basic block) by padding the missing ones with "PAD" flags. The constant values for S, B, and F can be determined using either a "majority strategy" or a "full strategy." The full strategy sets S, B, and F to encompass all the malware samples, blocks, and functions in the dataset, respectively, while the majority strategy sets S, B, and F to represent the majority of them, respectively. Compared to the full strategy, the majority strategy yields shorter input text, resulting in faster model training and reduced computational resource consumption. Thus, in our experiments, we adopted the majority strategy.

   Taking the dataset of AMG25 (detailed in Section 4.1) used in our experiment as an example, we employed the majority strategy and set S, F, and B to 127, 15, and 8, respectively. As illustrated in Appendix A, due to the commonly observed long-tail distribution, 76% of the functions, 73% of the blocks, and 72% of the instructions of the original malware samples in the dataset can be covered by this normalization strategy. Additionally, we conducted an extra experiment to evaluate the impact of S, F, and B values on our model and report the results in Appendix D. As Appendix D shows, higher values

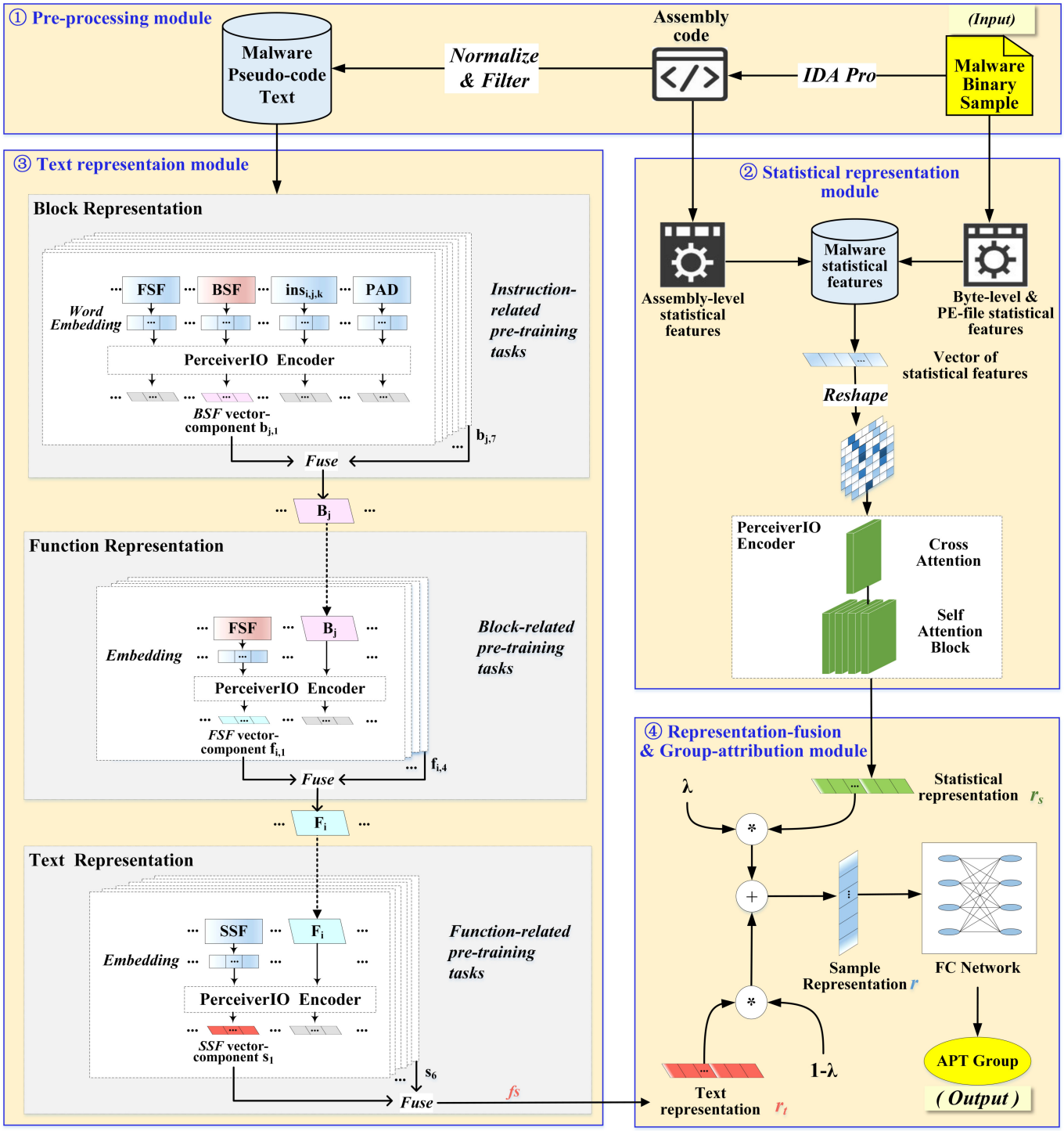


Fig. 1. Architecture of $MGAP^3$

for S, F, and B can considerably prolong model training time with only marginal performance improvements. This further emphasizes the rationale for our decision to utilize the majority strategy over the full strategy in our experiment.

2) **Insert start flags for samples, functions & blocks:** The module Inserts Sample Start Flag (SSF), Function Start Flag (FSF), and Block Start Flag (BSF) at the beginning of each sample file, function, and basic block, respectively.

3) **Tokenize code text:** The module treats each instruction, start flag, or padding flag as a separate token, and separate these tokens with commas.

Definition 1. Let *SampleText* be the pseudo-assembly text of a malware sample. Assume the number of functions in the sample, the number of basic blocks in each function, and the number of instructions in each block are S, F, and B, respectively, where each instruction consists of one opcode and possibly some operands. The **pseudo-code text** *SampleText*

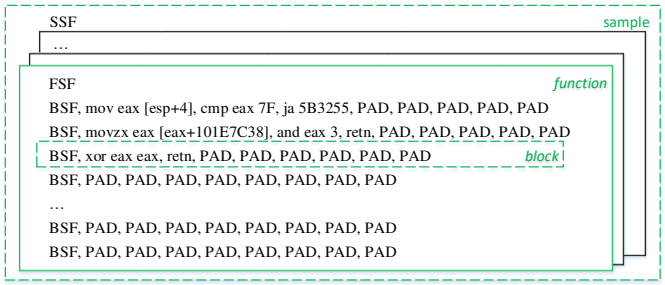


Fig. 2. An example of pseudo-code text

is defined in the following form of data dictionary entries:

$$
\begin{aligned}
SampleText &= \text{"SSF"} + \{\text{","} + [func \mid func_{PAD}]\}^{\mathrm{S}} \\
func &= \text{"FSF"} + \{\text{","} + [block \mid block_{PAD}]\}^{\mathrm{F}} \\
func_{PAD} &= \text{"FSF"} + \{\text{","} + block_{PAD}\}^{\mathrm{F}} \\
block &= \text{"BSF"} + \{\text{","} + [instru \mid instru_{PAD}]\}^{\mathrm{B}} \\
block_{PAD} &= \text{"BSF"} + \{\text{","} + instru_{PAD}\}^{\mathrm{B}} \\
instru &= opcode + (operand1) + (operand2) \\
instru_{PAD} &= \text{"PAD"}
\end{aligned}
\tag{1}
$$

3.1.2 Filtering UDF

The code of such non-UDFs as compiler-related functions, system DLL (Dynamic Link Library) functions, and so on, not created by APT-groups, might be noise for APT-group-attribution analysis of malware disassembly code. To reduce the potential interference of non-UDF noise during the code analysis, $MGAP^3$ attempts to identify and filter out non-UDFs before normalizing each sample's size. The pre-processing module of $MGAP^3$ first trains a PerceiverIO-based model to identify UDFs from non-UDFs, using labeled a dataset including obfuscated samples to support obfuscation-resilient identification, and then filters the UDFs from the malware functions to create the malware pseudo-code text. The construction of the datasets as well as the UDF-identification model will be introduced in the following parts of this subsection.

1) **Create function dataset with UDF/non-UDF labels:** Firstly, we collected 2447 DDL functions from Windows operating system, and utilized disassembly tools such as IDA Pro to get their assembly functions, which served as a part of the non-UDFs in the dataset. Secondly, we obtained the 100000 C++ functions from a source code dataset called AI-SOCO [50], compiled them to get the binary codes and PDB (Program Database File) files, disassembled the binaries into assembly functions using IDA Pro, and labeled the assembly functions with UDF/non-UDF based on the function type information in the PDB files. Thirdly, we used code obfuscation tools such as VMProtect to generate obfuscated versions of the aforementioned functions while keeping the function labels unchanged. Finally, we created an assembly-function dataset with UDF/non-UDF labels, including both obfuscated and non-obfuscated functions.

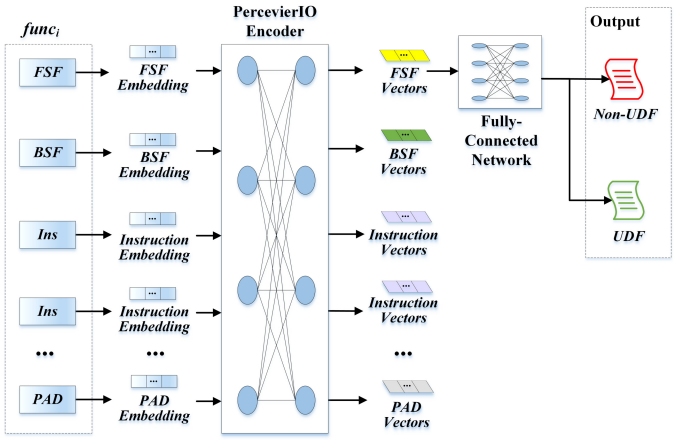


Fig. 3. UDF-identification model based on PercevierIO

TABLE 1
Statistical features of malware sample

Category	Features
Byte-level	Byte counting of 0∼255; Information entropy of byte n-gram.
PE file	File size; PE file header properties; Count of SectionTables; Count of ImportTables; Count of ImportFunctions; Count of ExportFunctions.
Disassembly-level	Count of Opcodes; Count of Jump opcodes; Count of Constants; Count of Registers; Count of Opcode types; Count of Operand data types; Count of Addressing types; Count of Functions types; Graph properties of CFG and FCG.

2) **Create UDF-identification model:** $MGAP^3$ create a PerceiverIO-based model to identify UDFs, as illustrated in Fig. 3. The pre-processing module pre-process the assembly functions by normalizing the sizes of functions and blocks with padding, inserting the flags of FSF and BSF, and tokenizing each function into instructions, start flags and padding flags. Next, for each function, $MGAP^3$ feeds their tokens into the model. The model firstly creates the word embeddings of the tokens, and then utilizes a PerceiverIO encoder to generate the encoding vectors. The FSF vectors, as the representations of the functions, are input into a classifier, a fully-connected network, to output the UDF/non-UDF labels of the functions.

3.2 Statistical representation module

The statistical representation module of $MGAP^3$ extracts static statistical features from disassembly codes as well as

binary codes of malware samples, and then creates representation vectors of those statistical features. The features fall into the following three categories, as listed in Table 1: (1) Byte-level statistical features, extracted using the Hexdump tool from malware binaries, involve the counting of bytes from 0 to 255, and the information entropy of each n-gram byte sequence $(1 \leq n \leq 4)$. (2) PE file statistical features, obtained by utilizing the parse tool of LIEF from the sample file, include the file's size and virtual size, the PE file header's properties (e.g., file version number, stack size), and the number of section tables/import tables/import functions/export functions. (3) Disassembly-level statistical features, extracted from disassembly code based on the IDA Pro tool, include the occurrence count of each Opcode/Jump opcode/Constant/Register, the occurrence number of each Opcode type (e.g. arithmetic, logic, etc.)/Operand data type/Addressing type/Functions type (e.g. static, etc.), and graph properties of CFG/FCG (e.g., the graph's depth, width, number of nodes or edges).

As shown in Fig. 1, firstly, the statistical representation module denotes the above numeric statistical features of each sample as a 5400-dimensional vector and further reshapes it into a three-channel vector. Next, the module feeds the sequence of three-channel vectors into a PerceiverIO encoder, which repeats the cross-attention and self-attention modules multiple times to iteratively refine the input vectors' representations. The cross-attention module facilitates the integration of spatial and sequential information, while the self-attention module captures intricate relationships and dependencies within the vectors. Finally, the PerceiverIO encoder encodes the input vector sequence into a representation sequence, where each representation is a malware sample's statistical representation, namely r_s in Fig. 1.

3.3 Text representation module

To generate the representation vector of a malware sample's pseudo-code text, the text representation module of $MGAP^3$ introduces a set of PerceiverIO networks, with the same architecture and model parameters, as the encoders, as illustrated in Fig. 1. The module constructs three layers of PerceiverIO encoders along with three levels of pre-training tasks to fully utilize unannotated code text. The first-layer PerceiverIO encoders are trained using instruction-related pre-training tasks at the first level to generate the representation vector of each basic block in the pseudo-code text, referred to as the block-representation. The second-layer PerceiverIO encoders are trained using block-related pre-training tasks at the second level to create the function-representation. The third-layer PerceiverIO encoders are trained using function-related pre-training tasks at the third level to produce the text-presentation of the malware sample.

3.3.1 Creating block-representation

To obtain deep representations of basic blocks, we define the following **7 instruction-related pre-training tasks** to learn semantic and syntactic relations between instructions:

Task 1-1. **Masked Instruction Prediction (INSTRUCTION):** Similar to the Masked Language Model (MLM) task proposed in BERT [24], this task randomly masks 15% of instructions in the embedding sequence of each malware sample, and then predicts those masked instructions. Since a [MASK] instruction may not always appear during fine-tuning, this task adopts the following masking strategy: If an instruction is selected, it will be replaced by the [MASK] token 80% of the time, by a random instruction token 10% of the time, and left unchanged 10% of the time. Inspired by the "token type" task presented by Yamaguchi et al. [39], we propose the following six type-related pre-training tasks (namely *Task1-2*∼*Task 1-7*). All the above type-related tasks can be considered as mutations of POS (Part-of-Speech Tagging) tagging tasks in NLP. However, unlike the "token type" task that predicts the type of the entire token, each of our six tasks aims to predict the type of one component (either an opcode or an operand) within an instruction token.

Task 1-2. **Opcode Type Classification (OPCODE TYPE):** Our second pre-training task is a four-way classification aimed at predicting the opcode type of an instruction as either an Arithmetic-operation (e.g., "sub", "add", "mul", "div", "dec", "inc", etc.), a Logic-operation (e.g., "and", "or", "not", "xor", etc.), a Stack-operation (e.g., "push" and "pop"), or an ordinary operation, which refers to an operation not belonging to any of the previous three types. This task randomly masks 15% of instructions in each sample's embedding sequence, and outputs the opcode-type labels of the masked ones based on the cross-entropy loss.

Task 1-3. **Jump-opcode Type Classification (JMP TYPE):** The third pre-training task is a three-way classification task, which classifies an instruction into one of the following three types: Non-jump instruction, Short-distance jump instruction, and Long-distance jump instruction. In 32-bit software, the range of address jumps for Short-distance jump instructions is between 127 bytes in the lower address range and 128 bytes in the higher address range. However, for Long-distance jump instructions, the range extends beyond the mentioned interval.

Task 1-4. **Operand1 Address-Type Classification (OPERAND1 ADR-TYPE):** This pre-training objective is to predict the addressing type of an instruction's first operand. It is a six-way classification task involving the following six address types: (1) Empty: It indicates the instruction has no operand, and an example of such instruction is "retn"; (2) Immediate-addressing: In the instruction "push 0," the immediate number "0" serves as an example of immediate addressing; (3) Register addressing: e.g., in the instruction "push ebp," the operand "ebp" is a register; (4) Memory addressing: e.g., "[ebx+8]" in the instruction of "push dword ptr ds: [ebx+8]" indicates memory addressing; (5) Base-index addressing: e.g., in the instruction "push dword ptr ds: [ebx+esi]," "ebx" and "esi" denote the base address and index, respectively. (6) Relative base-index addressing: For example, in the instruction "push dword ptr ds: [ebx+esi+8]," "ebx," "esi" and "8" represent the base address, index, and relative position, respectively.

Task 1-5. **Operand2 Address-Type Classification (OPERAND2 ADR-TYPE):** This pre-training task is very similar to the *Task 1-4*, also a six-way classification task of the same six categories. The only difference is that it focuses on predicting the addressing type of the second operand,

rather than that of the first operand, in an instruction.

Task 1-6. **Operand1 Data-Type Classification (OPERAND1 DATA-TYPE):** The sixth pre-training objective is to produce the data type of the first operand in an instruction. It is a three-way classification involving the following three data types: (1) Byte: e.g. the first operand's type in "mov byte ptr ds: [eax], 0x16"; (2) Word; (3)Double-word: e.g. the first operand's type in "mov dword ptr ds: [eax], 0x16".

Task 1-7. **Operand2 Data-Type Classification (OPERAND2 DATA-TYPE):** This pre-training task is almost identical to the *Task 1-6*, also a three-way classification using the same three categories. The only difference is that its goal is to output the data type of the second operand in an instruction, instead of the first operand.

$MGAP^3$ utilizes 7 first-layer PerceiverIO encoders together to create the vector representation of a basic block in the malware pseudo-code text, by performing the above 7 instruction-related pre-training tasks. Each first-layer encoder inputs the word-embedding sequence of the tokens, including all the instructions and paddings,1 SSF, 15 FSFs and 8 BSFs per block. For each instruction-related pre-training task, the PerceiverIO outputs all the blocks' encoding vectors, namely BSF vector-components. Taking a basic block of $block_{i,j}\,(1 \leq j \leq 15)$ in a function of $func_i\,(1 \leq i \leq 127)$ as an example, the PerceiverIO trained by the *t-th* instruction-related pre-training task encodes $block_{i,j}$ as $b_{j,t}\,(1 \leq t \leq 7)$,which is one component of the block BSF vector $B_j = (b_{j,1}, \cdots, b_{j,7})$. $MGAP^3$ creates the final BSF vector B_j (i.e., the block-representation) of $block_{i,j}$ by weighted fusion of all the 7 vector-components, including the following two steps:

1) Construct a single-layer FCN as the weight-computing network. The FCN inputs the 7 vector-components of $b_{j,1}, \cdots, b_{j,7}$, and maps them into 7 values which can be used as the weights, namely $w_{j,1}, \cdots, w_{j,7}$, for combining those vector-components into a single vector.
2) Perform weighted fusion on all the vector-components according to the following formula to obtain the fused vector, namely the final BSF vector B_j.

$$B_j = \sum_{t=1}^{7} b_{j,t} * w_{j,t} \tag{2}$$

3.3.2 Creating function-representation

To learn the representations of functions, we employ the following **4 block-related pre-training tasks** to learn semantic and syntactic associations between basic blocks:

Task 2-1. **Block with Input-register Detection (BLOCK INPUT):** This is a binary classification pre-training task with the objective of predicting whether a basic block involves input registers or not. If the first instruction in the block uses register addressing for one of its operands, the block is considered to contain input registers, which indicates that the block accepts input from the external.

Task 2-2. **Block with Output-register Detection (BLOCK OUTPUT):** This pre-training task aims to predict whether a basic block contains output registers or not. The prediction result is true when the last instruction uses register addressing for one of its operands, indicating that the block generates output to the external.

Task 2-3. **Block at Function-Tail Detection (TAIL BLOCK):** This pre-training goal is to tell whether a block is a tail block of its function or not. A tail block typically includes the return instruction of a function, which is usually the "retn" instruction. Due to the possibility of multiple function exits, a function may contain multiple tail blocks.

Task 2-4. **Control-Flow Edge Detection (CF EDGE):** For this control-flow-graph (i.e., CFG) related training task, to increase the task difficulty, we attempt to predict whether two block nodes are connected by a control flow edge or not. This task is more difficult than the prior block-related tasks, because the model needs to understand the complex control flow relation between blocks, which is semantic and possibly indirect.

To generate the final function representations, $MGAP^3$ pre-trains 4 second-layer PerceiverIO encoders simultaneously using the above block-related tasks. For the *t-th* block-related pre-training task and a function $func_i\,(1 \leq i \leq 127)$, the second-layer encoder accepts the vector sequence consisting of FSF embedding and block-representation vectors (e.g., B_j), and then encodes the function into $f_{i,t}\,(1 \leq t \leq 4)$, namely a component of the function FSF vector $F_i = (f_{i,1}, \cdots, f_{i,4})$. Similar to the vector-combining process depicted in Section 3.3.1, $MGAP^3$ constructs the final FSF vector F_i (i.e., the function-representation) of $func_i$ by combining the 4 vector-components based on a FCN weight-computing network that produces the weights of $w_{i,1}, \cdots, w_{i,4}$, as the following formula shows:

$$F_i = \sum_{t=1}^{4} f_{i,t} * w_{i,t} \tag{3}$$

3.3.3 Creating text-representation

To represent the semantic and syntactic meaning of a malware pseudo-code text, $MGAP^3$ utliizes the following **6 function-related pre-training tasks** to learn function relations.

Task 3-1. **User-defined-function Invocation Detection (UDF):** This pre-training task is a binary classification task to predict whether a function invocates UDFs or not.

Task 3-2. **Non-User-defined-function Invocation Detection (NON-UDF):** This pre-training goal is to tell whether a function invocates non-UDFs or not.

Task 3-3. **Return Value Detection (RETURN):** We define this pre-training task to predict whether a function has a return value or not.

Task 3-4. **Stack Access Detection (STACK):** This pre-training objective is to predict whether a function accesses stack or not. When a function needs to access the elements in stack, the function typically uses an instruction with an ebp register, which serves as the pointer to the stack bottom. For example, the instruction "mov eax, dword ptr ds: [ebp+8]" accesses the stack element through the ebp register.

Task 3-5. **Static Function Detection (STATIC):** This pre-training task is also a binary classification task, aiming at predicting whether a function is static or not.

Task 3-6. **Call Edge Detection (CALL EDGE):** To increase the difficulty of function-related tasks, we consider the function-call-graph (i.e., FCG) of a malware sample, and predict whether two function nodes are linked by a calling edge or not. We utilize this pre-training task to encourage the model to learn semantic invocation relation between functions.

To generate the text representation of a sample, the text representation module of $MGAP^3$ employs the above function-related tasks to pre-train 6 third-layer PerceiverIO encoders in parallel. For the *t-th* function-related pre-training task, the third-layer encoder takes a sequence that includes SSF embedding vectors and function representation vectors (e.g., F_i) as input and transforms the sample into $s_t\,(1 \leq t \leq 6)$, which is a constitute vector of the sample's final SSF vector $fs = (s_1, \cdots, s_6)$. Similar to the vector-fusing method described in Section 3.3.1, the text representation module also utilizes an FCN to fuse these 6 vector components, compute the weights $w_1, \cdots, w_6$, and ultimately generate the text representation r_t of the sample, which is identical to the final SSF vector fs, as shown in the following formula:

$$r_t = fs = \sum_{t=1}^{6} s_t * w_t \tag{4}$$

3.4 Representation-fusion & Group-attribution module

As explained in Sections 3.2 and 3.3, for each given malware instance, $MGAP^3$ employs PerceiverIO models to generate two distinct representation vectors: statistical representation r_s and text representation r_t. Both representation vectors produced by the PerceiverIO models are of the same size, with a dimensionality of 64. To ameliorate the representation of the malware instance, $MGAP^3$ further combines them into one representation r, according to the following formula:

$$r = r_t * (1 - \lambda) + r_s * \lambda \tag{5}$$

where λ denotes the weight for the combination, as the Representation-fusion & Group-attribution module shows in Fig. 1. It is worth noting that when λ is set to be 0.5, this representation-fusion method becomes the conventional approach of merging representation vectors through concatenation. Finally, the module utilizes an FCN as the classifier to attribute malware instances into different APT groups.

4 EXPERIMENT

4.1 Dataset AMG25

In this study, significant efforts were dedicated to the construction of a novel APT malware dataset called AMG25 (APT Malware with Group-label). The dataset comprises 5188 malware samples belonging to 25 distinct malware groups. Each APT group in our dataset consists of more than 30 samples, which ensures that we have sufficient data for robust train/test splitting and mitigates overfitting issues caused by limited training samples. Table 2 illustrates the 25 APT groups present in our dataset along with the corresponding number of malware samples. During the following experiments, we split this dataset into training and testing subsets according to the ratio of 9: 1.

TABLE 2
AMG25 Dataset

Group Name	Samples	Group Name	Samples
APT17	1499	FIN7	86
Lazarus Group	873	TA505	75
PatchWork	318	Molerats	70
APT10	308	Gamaredon Group	69
BlackTech	225	IceFog	59
Darkhotel	217	Silence	57
Donot	206	Higaisa	52
Blackgear	184	KONNI	47
APT28	175	PKPLUG	46
APT29	171	Hades	44
BITTER	130	APT37	43
Turla	113	Tick	30
PROMETHIUM	91		

To create the dataset, we initially compiled a list of APT malware, including their MD5 hashes and group labels, by utilizing public threat intelligence sources. Furthermore, the group labels underwent manual verification by security analysts employed at a renowned security company in China. Subsequently, we proceeded to gather the malware samples associated with the MD5 hashes listed, sourcing them from VirusSign [53]. Furthermore, we developed a feature analysis tool to extract statistical features, as outlined in Table 1, for each malware sample. These features were then converted into gray-scale images. We are releasing the malware dataset, which includes the APT group label and hash value for each malware file, along with its corresponding gray-scale feature image.

4.2 Experimental Setup

4.2.1 Evaluation Metric

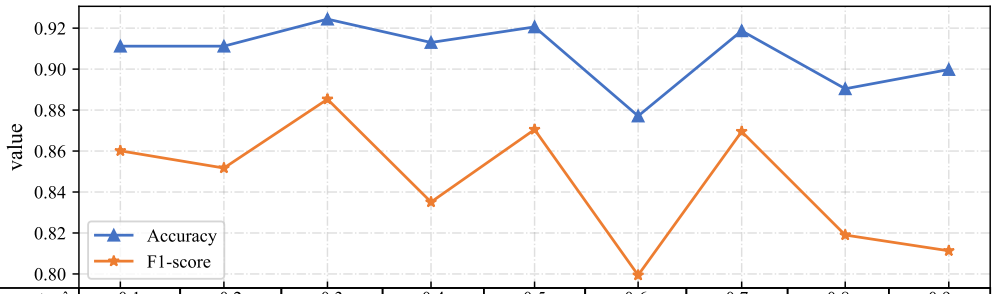
To evaluate $MGAP^3$ performance, we report the accuracy and macro-F1 score for group classification of testing malware instances. Assume that TP_i, TN_i, FP_i and FN_i denote the true positive, true negative, false positive and false negative for the ith APT group, respectively, and N denotes the total number of APT groups, the accuracy and the macro-F1 scores can be calculated using the following formulas:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{6}$$

$$\begin{aligned} \text{F1-score} &= \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \\ where\ \text{Precision} &= \frac{1}{N}\sum_{i=1}^{N}\frac{TP_i}{TP_i + FP_i}, \\ \text{Recall} &= \frac{1}{N}\sum_{i=1}^{N}\frac{TP_i}{TP_i + FN_i} \end{aligned} \tag{7}$$

4.2.2 Compared Approaches

We compare our $MGAP^3$ approach with eight group attribution methods for APT malware. The first competitor,



Parameter λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Accuracy	0.9112	0.9112	0.9244	0.913	0.9206	0.8771	0.9187	0.8904	0.8998
F1-score	0.8601	0.8517	0.8853	0.8351	0.8705	0.7994	0.8695	0.819	0.8113

Fig. 4. $MGAP^3$ performance across various values of weight parameters λ

BinMLM, is the most recent static attribution model proposed by Song et al [16]. BinMLM utilizes an lstm encoder and a gate layer to generate each malware instance's representation vector, and then classify malware based on vector similarity. As the BinMLM model is not publicly accessible, we implemented the model based on our understanding of its principles.

To evaluate our method against potential approaches that utilize advanced representation learning techniques for malware group attribution, we implemented three competitor methods by adapting state-of-the-art (SOTA) techniques from other domains: (1) CAE: For each APT malware sample, we created a combined grayscale image of its assembly code and features. We then used a SOTA self-supervised representation learning encoder, the Context AutoEncoder (CAE) [40], to extract representations from these images. (2) MalConv2: We modified the leading end-to-end malware detection model, MalConv2 (also known as MalConv-GCG) [43], by incorporating a multi-classification head. This adaptation allows the model to categorize malware into specific groups. (3) SecureBERT_Mal: SecureBERT [41] represents a state-of-the-art pre-trained cybersecurity language model built on the Roberta architecture. We adapted SecureBERT_Mal [42], a variant fine-tuned specifically for malware family classification, by modifying its classification head and further fine-tuning it to attribute APT malware groups.

The remaining four competitor methods are Ours-with-RNN, Ours-with-GRU, Ours-with-LSTM, and Ours-with-Transformer. These methods were implemented by substituting the PerceiverIO encoders in the text representation module of our $MGAP^3$ approach with RNNs, GRUs, LSTMs, and Transformers, correspondingly.

4.2.3 Implementation Detail

Network Architecture: We used IDA Pro 7.5 to extract statistical features of malware instances. We implemented $MGAP^3$ model based on the Pytorch 1.7.1, LIEF 0.11.0, Hexdump 2.0.0 packages with Jupyter Notebook as the IDE, and ran models on a GPU server with Tesla P40 cards and Windows Server 2016 OS. The network architecture of our $MGAP^3$ model, involving a set of PerceiverIO encoders and FCNs, is exhibited in Appendix B.

TABLE 3
Comparison results of $MGAP^3$ and competitors on AMG25

Methods	Accuracy	F1-score
BinMLM [16]	85.26%	0.8531
CAE [40]	79.01%	0.4687
MalConv2 [43]	86.09%	0.7662
SecureBERT_Mal [41][42]	79.58%	0.7278
$Ours$-with-RNN	82.47%	0.7336
$Ours$-with-GRU	90.37%	0.8594
$Ours$-with-LSTM	91.14%	0.8569
$Ours$-with-Transformer	91.52%	0.8759
$MGAP^3$ ($Ours$)	**92.44%**	**0.8853**

Hyperparameters: We adopted the Adam optimizer to train all PerceiverIO encoders with the initial learning rate (η_θ) of 0.001, and used the cross-entropy loss as the loss function.

4.3 Effectiveness study

4.3.1 Determination of parameter λ

As described in Section 3.4, to generate the final representation of malware instances, the $MGAP^3$ approach utilizes weight parameters λ to fuse the text representation vector r_t and the statistical representation vector r_s. In this subsection, we experimentally determine the parameter value λ to achieve the best performance of the $MGAP^3$ model.

In our study on the AMG25 dataset, we investigated the performance of the $MGAP^3$ model with different values of λ ranging from 0.1 to 0.9, increasing by 0.1 increments. The experimental results are illustrated in Fig. 4. From Fig. 4, it can be observed that the $MGAP^3$ model achieves the best performance for malware group classification when λ is set as 0.3. At this value of λ, the $MGAP^3$ model achieves an accuracy of 92.44% and an F1-score of 0.8853. Therefore, we finally set the weight parameter λ to 0.3 based on these findings.

4.3.2 Attribution effectiveness

We conducted a comparison of our $MGAP^3$ approach with eight competitors (BinMLM [16], CAE [40], MalConv2 [43], SecureBERT_Mal [41][42], Ours-with-RNN, Ours-with-GRU, Ours-with-LSTM, and Ours-with-Transformer) on the

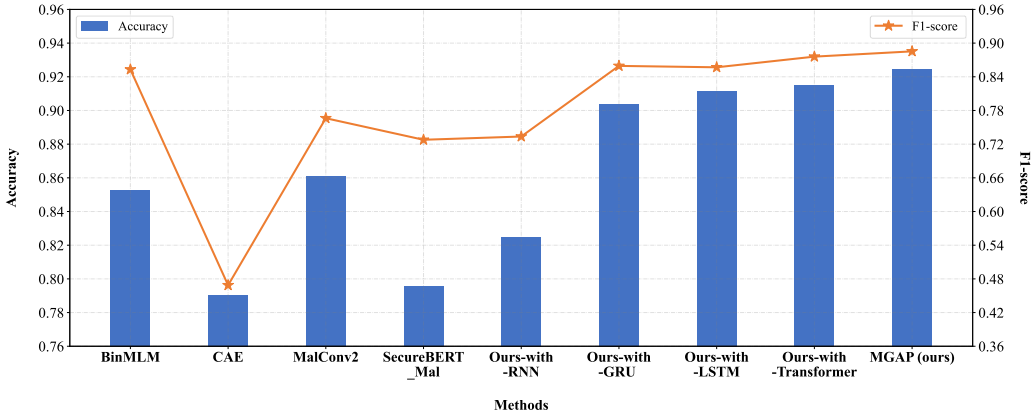


Fig. 5. Visualized effectiveness of $MGAP^3$ and competitors

AMG25 dataset. The comparison results are presented in Table 3. It is evident that our approach outperforms all competitors, achieving the highest classification accuracy of 92.44% and a macro F1-score of 0.8853. The attribution effectiveness of these compared methods is visualized in Fig. 5.

Furthermore, there are three additional observations to be made. Firstly, the approach of "Ours-with-Transformer", which applies our approach's framework but replaces the encoder units of PerceiverIO with Transformer, is considered suboptimal compared to the remaining four approaches that utilize LSTM, GRU, or RNN as sequence encoder units. This outcome can be attributed to the inherent advantages of PerceiverIO and Transformer, such as their utilization of a self-attention mechanism, which allows for more effective capturing of dependencies between different elements in a sequence. Secondly, both the "Ours-with-LSTM" and "BinMLM" approaches utilize LSTM networks for encoding. While "Ours-with-LSTM" shares the hierarchical pre-training tasks and the UDF filtering strategy with our method, "BinMLM" does not. The results shown in Table 3 and Figure 5 indicate that "Ours-with-LSTM" outperforms "BinMLM" in terms of accuracy and F1-score. This suggests that the superiority of our $MGAP^3$ approach is likely due to the hierarchical pre-training technique and the UDF filtering strategy. Finally, it is noteworthy that while the pre-training of "BinMLM" or "SecureBERT_Mal" relies on large external code corpora, the pre-training for "Ours-with-LSTM" or our method utilizes a much smaller dataset, comprising around five thousand malware samples. Hence, "Ours-with-LSTM" and our approach manage to attain a higher quality of code representation with significantly less pre-training effort, thanks to the more effective pre-training technique employed.

4.4 Ablation analysis

Our $MGAP^3$ approach comprises three essential components: the text representation module, designed to handle malware pseudo-codes; the statistical representation module, which processes binary malware instances; and the UDF filtering component, tailored for pre-processing malware assembly code. The text representation module employs hierarchical polytype pre-training tasks to generate malware representation. To further validate the effectiveness of $MGAP^3$, we conduct an ablation analysis on the aforementioned components and pre-training tasks.

4.4.1 *Ablating key component*

To assess the individual impact of each aforementioned key module on the performance of the $MGAP^3$ approach, we conducted an ablation analysis on the AMG25 dataset in three distinct scenarios, where each scenario involves removing one module from the implementation of $MGAP^3$.

(1) Text representation (w/o): In this scenario, the post-ablation model is obtained by excluding the text representation module, and thus only includes the statistical representation module,

(2) Statistical representation (w/o): In this scenario, the post-ablation model is constructed by removing the statistical representation module, and thus solely utilizes the text representation module.

(3) UDF Filtering (w/o): In this scenario, the post-ablation model is created by removing the component for UDF filtering, and eventually uses both UDFs and non-UDFs in malware code texts. By contrast, our $MGAP^3$ only utilizes the UDFs.

The results of the ablation study are presented in Table 4. Each row in the table represents a model variant, including our $MGAP^3$ model and the models following ablation. The columns labeled "ΔAccuracy" and "ΔF1-score" display the performance decline in terms of attribution accuracy and F1-score, respectively, following the removal of the component listed in the first column from our $MGAP^3$ model. This performance reduction is visualized in Fig. 6 (a). Notably, a larger decrease in performance resulting from an ablated component suggests a more significant contribution of that component to the overall performance of our $MGAP^3$ model.

Based on the findings presented in Table 4, it is clear that all three components enhance the attribution performance of our $MGAP^3$ model. Fig. 6 (a) visualize the impact of each component on improving model performance. According to Table 4 and Fig. 6 (a), the Text representation module makes the most significant contribution, resulting in an 8.31% increase in accuracy and a 0.1245 enhancement in

TABLE 4
Ablation study for key components of $MGAP^3$

Model	Accuracy	ΔAccuracy	F1-score	ΔF1-score
Text representation (w/o)	**84.31%**	**-8.13%**	**0.7608**	**-0.1245**
Statistical representation (w/o)	86.39%	-6.05%	0.8118	-0.0735
UDF Filtering (w/o)	90.94%	-1.5%	0.8671	-0.0182
$MGAP^3$	92.44%	0	0.8853	0

TABLE 5
Ablation study for 17 individual pre-training tasks of $MGAP^3$

Task category	Task Name	Accuracy	ΔAccuracy	F1-score	ΔF1-score	Minutes/epoch	#Epoch
Instruction-related task (w/o)	**INSTRUCTION**	**90.17%**	**-2.27%**	**0.8322**	**-0.0531**	**6.60**	**5**
	OPCODE TYPE	91.87%	-0.57%	0.8584	-0.0269	6.50	5
	JMP TYPE	90.36%	-2.08%	0.8448	-0.0405	6.52	5
	OPERAND1 ADR-TYPE	90.55%	-1.89%	0.8515	-0.0338	6.52	5
	OPERAND2 ADR-TYPE	91.49%	-0.95%	0.8509	-0.0344	6.53	5
	OPERAND1 DATA-TYPE	91.49%	-0.95%	0.8481	-0.0372	6.52	5
	OPERAND2 DATA-TYPE	91.49%	-0.95%	0.8661	-0.0192	6.53	5
Block-related task (w/o)	BLOCK INPUT	90.93%	-1.51%	0.8494	-0.0359	6.37	3
	BLOCK OUTPUT	91.49%	-0.95%	0.8547	-0.0306	5.55	3
	TAIL BLOCK	91.68%	-0.76%	0.8689	-0.0164	5.50	3
	CF EDGE	**90.17%**	**-2.27%**	**0.8193**	**-0.0660**	**5.55**	**3**
Function-related task (w/o)	UDF	90.74%	-1.70%	0.8517	-0.0336	0.35	5
	NON-UDF	**90.55%**	**-1.89%**	**0.8431**	**-0.0422**	**0.20**	**25**
	RETURN	91.68%	-0.76%	0.8609	-0.0244	0.22	80
	STACK	91.12%	-1.32%	0.8446	-0.0407	0.20	5
	STATIC	90.74%	-1.70%	0.8485	-0.0368	0.38	80
	CALL EDGE	91.68%	-0.76%	0.8736	-0.0117	0.38	80

F1-score. Meanwhile, the Statistical representation module leads to a 6.05% rise in accuracy and a 0.0735 gain in F1-score. Additionally, employing the UDF-filtering component of the pre-processing module results in a 1.5% improvement in accuracy and a 0.0182 increase in F1-score.

The aforementioned results underscore the vital importance of integrating text and statistical representations in our $MGAP^3$ approach. Additionally, harnessing the combined advantages of UDF-filtering is crucial for maximizing the performance of our approach.

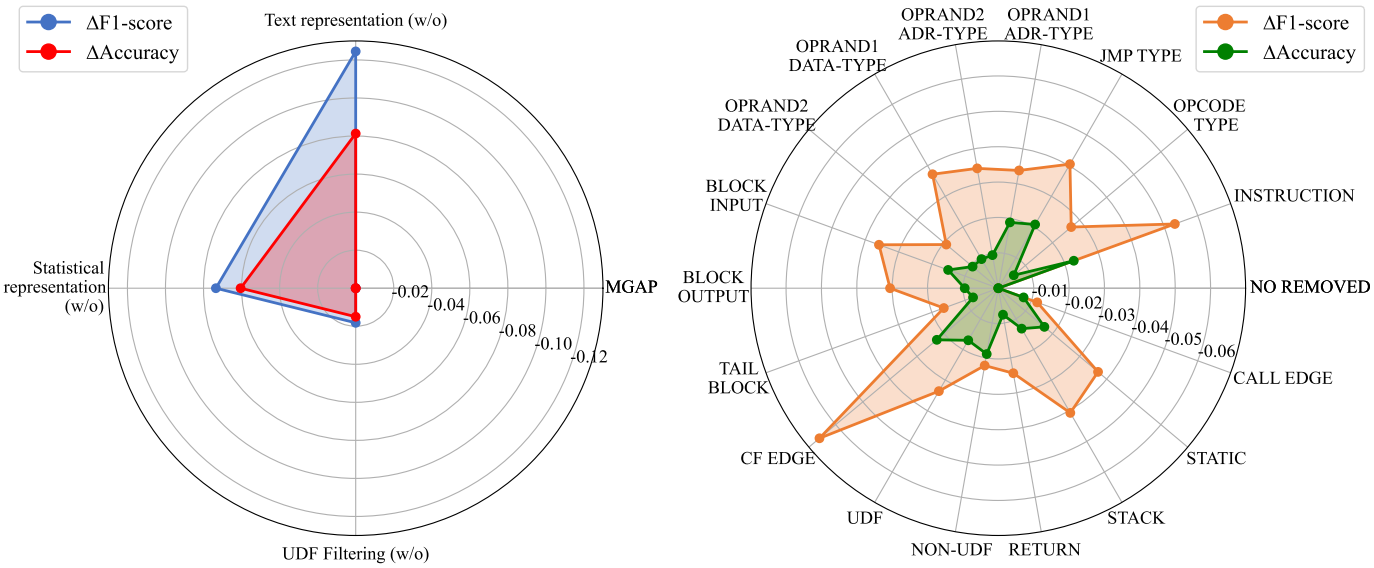
4.4.2 Ablating individual pre-training task

As demonstrated in the previous subsection, the Text representation module contributes most significantly to the performance of our approach, $MGAP^3$. Given that the text representation module incorporates 17 pre-training tasks to generate malware instance representations, we conducted further evaluation on the individual contributions of each pre-training task to the overall performance of our approach. To this end, we conducted ablation studies by sequentially removing each task from our $MGAP^3$ model.

The outcomes of these ablation experiments are detailed in Table 5. In the table, each row corresponds to a post-ablation model missing one of the 17 pre-training tasks. The category and name of each task are specified in the first two columns. The columns labeled "ΔAccuracy" and "ΔF1-score" show the decreases in accuracy and F1-score, respectively, following each ablation. These variations are also illustrated in Fig. 6 (b), where a steeper variation indicates a more substantial impact of the respective pre-training task on the performance of $MGAP^3$. Additionally, we investigate the time efficiency of each pre-training task, reporting the time cost (in minutes) for one training epoch and the total number of epochs for our $MGAP^3$ model in the last two columns of Table 5.

From Table 5 and Fig. 6 (b), we can make the following observations: (1) Each of the 17 pre-training tasks in this study contributes to the improved classification performance of $MGAP^3$. The enhancement in accuracy ranges from 0.57% to 2.27%, while the improvement in F1-score ranges from 0.0117 to 0.0660. (2) The top-performing tasks within each hierarchical task category, which achieved the most significant performance improvements, are highlighted in bold in Table 5. Specifically, these tasks are the instruction-related task "INSTRUCTION," the block-related task "CF EDGE," and the function-related task "NON-UDF." (3) The time cost per pre-training epoch is slightly higher for instruction-related tasks (6.50-6.60 minutes) compared to block-related tasks (5.50-6.37 minutes), and much higher compared to function-related tasks (0.20-0.38 minutes). The total number of training epochs is also larger for instruction-related tasks (5 epochs) compared to block-related tasks (3 epochs), while for function-related tasks, it ranges from 5 to 80 epochs.

These findings underscore the significance of each pre-training task introduced in our study for enhancing the performance of the $MGAP^3$ model. Each category includes multiple pre-training tasks that significantly improve the model's metrics, achieving increases in accuracy greater than 1.5% and in F1-score exceeding 0.04. Notably, of the 17 pre-training tasks, the function-related task "CF EDGE" achieves the best performance improvement with low pre-



(a) After removing individual key components **(b) After removing individual pre-training tasks**

Fig. 6. Performance Reduction after Ablation

TABLE 6
Performance evaluation of $MGAP^3$-Lite

Model	Task	Minutes/epoch	#Epoch	Total Minutes	Accuracy	F1-score
***MGAP*^{3}-Lite**	Instruction-related task: INSTRUCTION (with)	6.60	5	33	-	-
	Block-related task: CF EDGE (with)	1.47	3	4.41	-	-
	Function-related task: NON-UDF (with)	0.06	25	1.5	-	-
	3 pre-training tasks	-	-	**38.91**	**90.70% (±0.3183%)**	**0.8451 (±0.0103)**
$MGAP^3$	17 pre-training tasks	-	-	383.66	92.44%	0.8853

training time cost. Compared to traditional instruction-related tasks, our block-related and function-related tasks deliver comparable performance improvements while requiring less training time.

4.4.3 $MGAP^3$-Lite: Lightweight model with simplified pre-training tasks

As described in the previous subsection, the top tasks within each task category are the three hierarchical pre-training tasks: "INSTRUCTION," related to instructions, "CF EDGE," related to blocks, and "NON-UDF," related to functions. Notably, compared to the tasks related to instructions, the block-related "CF EDGE" and the function-related "NON-UDF" tasks provide comparable or even superior performance improvements while requiring less training time. Therefore, to accelerate the training of our attribution model, we concentrated solely on these three hierarchical tasks during pre-training. This strategy enabled us to develop a lightweight model named $MGAP^3$-Lite, based on 3 (rather than 17) pre-training tasks. We experimentally evaluated the time expenditures and attribution performance of $MGAP^3$-Lite, and present the results in Table 6. This table features columns for "Model," listing both our lightweight model $MGAP^3$-Lite and our full model $MGAP^3$ for comparison, and "Task," indicating the pre-training tasks utilized by each model.

For each task of the model $MGAP^3$-Lite, the pre-training time is calculated by multiplying the duration of each epoch by the total number of epochs. The column labeled "Total Minutes" for each model represents the aggregate pre-training time for all tasks within the model. As indicated in the table, the total pre-training time for $MGAP^3$-Lite is 38.91 minutes, which is merely 10% of the 383.66 minutes required for $MGAP^3$.

To mitigate the overfitting issue in our model on the dataset, we implemented 10-fold cross-validation to assess the attribution performance of the model $MGAP^3$-Lite. For detailed results of the 10 iterations, please refer to Appendix C. In Table 6, the columns "Accuracy" and "F1-score" display the performance metrics formatted as "Mean±SD (Standard Deviation)". As shown in the table, the performance of our simplified model $MGAP^3$-Lite is comparable to that of our full version, $MGAP^3$, and significantly surpasses that of the state-of-the-art (SOTA) methods, including BinMLM [16], CAE [40], MalConv2 [43], and SecureBERT_Mal [41][42], as presented in Table 3.

In conclusion, we optimized the training of $MGAP^3$-Lite by focusing on three key hierarchical tasks instead of the prior seventeen, drastically reducing training time while maintaining robust performance. Despite its simplicity, $MGAP^3$-Lite matches the full model in key metrics and surpasses state-of-the-art methods, proving its efficiency and effectiveness.

5 Discussions and limitations

Firstly, in real-world scenarios, our MGAP3 method holds the potential to significantly enhance model performance, by effectively utilizing large datasets through its key techniques: (1) Noise Filtering Technique: It mitigates the increased code noise found in large datasets containing numerous non-UDFs, effectively reducing overfitting risks and processing time; (2) Multimodal Data Fusion Technique: Leveraging the exceptional multimodal fusion capabilities of PerceiverIO, this technique can effectively exploit the diversity of large datasets, uncovering complementarities between different modalities to boost overall model performance and reduce the risk of overfitting a single modality. (3) Hierarchical Pre-training Technique: Pre-training with large datasets or extensive external malware code corpora can greatly refine the quality of code representations. By integrating hierarchical pre-training tasks with Perceiver IO's compact latent space, this technique delves into code features and structures, exploring complex relationships within the codes, thereby enhancing the model's performance and generalization ability. Additionally, our streamlined MGAP3-Lite model, designed to reduce system complexity and processing time while preserving nearly full effectiveness, is ideally suited for deployment in real-world scenarios.

Secondly, when constructing our attribution model, we assume that APT group labels for each malware sample in the dataset are accurate. However, these labels originate from various sources such as different security analysis firms, diverse open-source intelligence, and automated tools. Due to differences in analysis and potential biases inherent to these sources, the dataset may contain uncertain or even incorrect labels. Like existing studies on malware group attribution, our method does not consider the impact of these low-quality labels on model performance. Nevertheless, in the field of malware family classification, researchers have developed effective techniques to address the issue of low-quality labels. Techniques such as confidence learning [44] and hybrid representation learning [45] have been proposed. These methods have the potential to be adapted to address similar challenges in malware group attribution.

Thirdly, our group attribution model is trained under a closed-set assumption, which assumes that the training dataset encompasses all APT groups that might appear in the test data. However, in the ever-evolving realm of malware, APT groups continually adapt, with new entities emerging. As a result, a test malware instance may belong to an APT group that was not included during training. This discrepancy underlines the importance of research on malware group attribution in open-set scenarios, where known and previously unknown groups can be accurately identified. Techniques developed for the open-set classification of malware families, as detailed in references [46] and [47], offer valuable insights and could be adapted to enhance our methods in open-set environments.

Finally, APT groups often employ sophisticated techniques to evade detection, such as polymorphism, encryption, and obfuscation. These techniques make it challenging to precisely link malware to specific groups. Currently, our approach does not incorporate adversarial robustness. Therefore, malware that employs obfuscation and similar strategies for adversarial attacks could reduce the effectiveness of our model's attribution capabilities.

6 Conclusion and future work

In this study, we have alleviated the challenges of attributing malware to specific APT groups. Our $MGAP^3$ model, integrating multiple PerceiverIO encoders to create a unified code representation, effectively improved the performance of APT malware group attribution. The model innovatively uses polytype and hierarchical pre-training for text representations, combined with multi-view features for statistical representations. The proposed pre-training tasks played a pivotal role in enhancing code representations and, in turn, improving overall attribution performance. We also introduced the AMG25 dataset for group attribution research. Our experiments demonstrated the superiority of $MGAP^3$ over existing baseline and state-of-the-art models, showcasing its potential in accurately attributing APT malware to their respective groups. Furthermore, we streamlined $MGAP^3$-Lite by concentrating on three key tasks, which cut down training time while preserving strong performance, thereby demonstrating its efficiency and efficacy.

While this research has made significant progress in APT malware attribution, several avenues remain for future exploration: (1) Expanding pre-training base: Pre-train our textual code representation model on large real-world datasets or external malware code corpora. This initiative aims to not only enhance the MGAP3 method's generalizability, but also to develop a robust pre-trained model that serves diverse downstream malware analysis tasks. (2) In-depth exploration of our pre-training tasks: Delve into the optimal combination of the proposed pre-training tasks to further enhance the $MGAP^3$ model's performance and efficiency, and investigate their effectiveness in various downstream malware analysis tasks. (3) Evaluating Large-Scale Scenarios: Conduct comprehensive evaluations of $MGAP^3$ and other attribution models with large-scale datasets in diverse APT scenarios to validate their performance in various real-world situations. (4) Adversarial Robustness: Assess the robustness of $MGAP^3$ against adversarial attacks to ensure its reliability in the face of sophisticated and targeted cyber threats.

References

[1] SECURELIST by Kaspersky. "IT threat evolution in Q3 2022. Non-mobile statistics". securelist.com. https://securelist.com/it-threat-evolution-in-q3-2022-non-mobile-statistics/107963/ (accessed July 27, 2023).

[2] P. Chen, L. Desmet and C. Huygens, "A study on advanced persistent threats," in *Communications and Multimedia Security: 15th IFIP TC 6/TC 11 Int. Conf. (CMS)*, 2014, Proc. 15, pp. 63-72.

[3] C.M. Chen, G.H. Lai and D.W. Wen, "Evolution of Advanced Persistent Threat (APT) Attacks and Actors," in *New Trends in Computer Technologies and Applications: 23rd Int. Computer Symposium (ICS)*, 2018, pp. 76-81.

[4] Y.X Sun, J. Weng and Z.T. Li, "Software Security Technology (in Chinese)," Tsinghua University Press, Beijing, China, April 2022.

[5] W. Chen, X. Helu, C. Jin *et al.*, "Advanced persistent threat organization identification based on software gene of malware," *Transactions on Emerging Telecommunications Technologies*, vol. 31, no. 12, pp. 3884-3897, 2020.

[6] J. Hong, S. Park *et al.*, "Classifying malwares for identification of author groups", *Concurrency and Computation: Practice and Experience*, vol. 30, no. 3, e4197, 2017. [Online] Available: https://doi.org/10.1002/cpe.4197.

[7] W. Han, J. Xue, Y. Wang, F. Zhang and X. Gao, "APTMalInsight: Identify and cognize APT malware based on system callinformation and ontology knowledge framework," *Information Sciences*, vol. 546, pp. 633-664, 2021.

[8] S. Li, Q. Zhang, X. Wu, W. Han and Z. Tian, "Attribution classification method of APT malware in IoT using machine learning techniques," *Security and Communication Networks*, pp. 1-12, 2021.

[9] N. Xu, S. Li, X. Wu, W. Han and X. Luo, "An APT Malware Classification Method Based on Adaboost Feature Selection and LightGBM," in *2021 IEEE 6th Int. Conf. on Data Science in Cyberspace (DSC)*, 2021, pp. 635-639.

[10] Q. Wang, H. Yan and Z. Han, "Explainable APT Attribution for Malware Using NLP Techniques," in *2021 IEEE 21st Int. Conf. on Software Quality, Reliability and Security (QRS)*, 2021, pp. 70-80.

[11] G. Laurenza, L. Aniello, R. Lazzeretti and R. Baldoni,"Malware triage based on static features and public apt reports," in *Cyber Security Cryptography and Machine Learning: 1st Int. Conf. (CSCML)*, June 2017, Proceedings 1, pp.288-305.

[12] I. Rosenberg, G. Sicard and E. David, "End-to-End Deep Neural Networks and Transfer Learning for Automatic Analysis of Nation-State Malware," *Entropy*, vol. 20, no. 5, pp. 390, 2018.

[13] G. Laurenza, R. Lazzeretti and L. Mazzotti, "Malware triage for early identification of advanced persistent threat activities," *Digital Threats: Research and Practice*, vol. 1, no. 3, pp. 1-17, 2020.

[14] C. Wei, Q. Li, D. Guo and X. Meng, "Toward Identifying APT Malware through API System Calls," *Security and Communication Networks*, pp. 1-14, 2021.

[15] J. Liu, Y. Shen and H. Yan, "Functions-based CFG Embedding for Malware Homology Analysis," in *2019 26th Int. Conf. on Telecommunications (ICT)*, 2019, pp. 220-226.

[16] Q. Song, Y. Zhang, L. Ouyang and Y. Chen, "BinMLM: Binary Authorship Verification with Flow-aware Mixture-of-Shared Language Model," in *2022 IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1023-1033.

[17] R. Zheng, Q. Wang, Z. Lin, Z. Jiang, J. Fu and G. Peng, "Cryptocurrency malware detection in real-world environment: Based on multi-results stacking learning," *Applied Soft Computing*, vol. 124, no. 109044, 2022, doi: 10.1016/j.asoc.2022.109044.

[18] H. Shi, S. Gao, Y. Tian *et al.*, "Learning bounded context-free-grammar via LSTM and the transformer: Difference and the explanations," in *Proc. AAAI Conf. on Artificial Intelligence*, 2022, vol. 36, no. 8, pp. 8267-8276.

[19] T. Xiao, Z. Quan, Z.J. Wang *et al.*, "Loader: A Log Anomaly Detector Based on Transformer," *IEEE Transactions on Services Computing*, pp. 1-14, 2023.

[20] I. Sutskever, O. Vinyals and Q.V. Le, "Sequence to sequence learning with neural networks", *Advances in neural information processing systems*, vol. 27, pp. 1-9, 2014.

[21] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[22] K. Cho, B. Van Merriënboer, C. Gulcehre *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*.

[23] A. Vaswani, N. Shazeer, N. Parmar *et al.*, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998-6008, 2017.

[24] J. Devlin, M.W. Chang, K. Lee *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[25] A. Jaegle, F. Gimeno, A. Brock, O. Vinyals, A. Zisserman and J. Carreira, "Perceiver: General perception with iterative attention," in *Int. Conf. on machine learning*, 2021, pp. 4651-4664.

[26] A. Jaegle, S. Borgeaud, J.B. Alayrac *et al.*, "Perceiver IO: A general architecture for structured inputs & outputs," 2021, *arXiv:2107.14795*.

[27] N.D. Nguyen and D. Wang, "Multiview learning for understanding functional multiomics," *PLoS computational biology*, vol. 16, no. 4, pp. 1-26, 2020.

[28] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, 2020.

[29] M. Noroozi and P. Favaro, "Unsupervised learning of visual representations by solving jigsaw puzzles," in *European Conf. on computer vision*, 2016, pp. 69-84.

[30] I. Misra, C.L. Zitnick and M. Hebert, "Shuffle and learn: unsupervised learning using temporal order verification," in *Computer Vision–ECCV 2016: 14th European Conf., Amsterdam, The Netherlands*, 2016, Part I 14, pp. 527-544.

[31] D. Guo, S. Ren, S. Lu *et al.*, "Graphcodebert: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.

[32] N.D.Q. Bui, Y. Yu and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd Int. Conf. on Software Engineering (ICSE)*, 2021, pp. 1186-1197.

[33] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu and H Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proc. 44th Int. Conf. on Software Engineering*, 2022, pp.2377-2388.

[34] A. Rahali and MA. Akhloufi, "MalBERT: Malware Detection using Bidirectional Encoder Representations from Transformers," in *2021 IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC)*, 2021, pp. 3226-3231.

[35] Z. Xu, X. Fang and G. Yang, "Malbert: A novel pre-training method for malware detection," *Computers & Security*, vol. 111, pp. 102458-102476, 2021.

[36] B. Wu, Y. Xu and F. Zou, "Malware Classification by Learning Semantic and Structural Features of Control Flow Graphs," in *2021 IEEE 20th Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021, pp. 540-547.

[37] J.L. Alvares, "Malware classification with bert," M.S. thesis, Dept. Computer Science, San Jose State Univ., San Jose, CA, USA, 2021.

[38] R. Oak, M. Du, D. Yan, H. Takawale and I. Amit, "Malware detection on highly imbalanced data through sequence modelling," in *Proc. 12th ACM Workshop on artificial intelligence and security*, 2019, pp. 37-48.

[39] A. Yamaguchi, G. Chrysostomou, K. Margatina and N. Aletras, "Frustratingly simple pretraining alternatives to masked language modeling," 2021, *arXiv:2109.01819*.

[40] X. Chen, M. Ding, X. Wang *et al.*, "Context autoencoder for self-supervised representation learning," in *Int. Journal of Computer Vision (IJCV)*, vol. 132, no. 1, pp. 208-223, 2024.

[41] E. Aghaei, X. Niu, W. Shadid and E. Al-Shaer, "Securebert: A domain-specific language model for cybersecurity," in *Int. Conf. on Security and Privacy in Communication Systems*, 2022, pp. 39-56.

[42] Kaushik. "SecureBert_Malware-Classification". github.com. https://github.com/kaushik-42/SecureBert_Malware-Classification (accessed May 1, 2024).

[43] E. Raff, W. Fleshman, R. Zak *et al.*, "Classifying sequences of extreme length with constant memory applied to malware detection," in *Proc. AAAI Conf. on Artificial Intelligence*, 2021, vol. 35, no. 11, pp. 9386-9394.

[44] L. Wang, H. Wang, X. Luo *et al.*, "MalWhiteout: Reducing label errors in android malware detection," in *Proc. 37th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2022, pp. 1-13.

[45] Y. Zhang, Y. Sui, S. Pan *et al.*, "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 15, pp. 3401-3414, 2019.

[46] J. Guo, H. Wang, Y. Xu, W. Xu, Y. Zhan, Y. Sun and S. Guo, "Multimodal Dual-Embedding Networks for Malware Open-Set Recognition," *IEEE Transactions on Neural Networks and Learning Systems*, Early Access, pp. 1-15, 2024, doi: 10.1109/TNNLS.2024.3373809.

[47] J. Guo, S. Guo, S. Ma, Y. Sun and Y. Xu, "Conservative novelty synthesizing network for malware recognition in an open-set scenario," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 2, pp. 662-676, 2021, doi: 10.1109/TNNLS.2021.3099122.

[48] Groups | MITRE ATT&CK. "MITRE ATT&CK". attack.mitre.org. https://attack.mitre.org/groups/ (accessed July 27, 2023).
[49] Cyber-research. "APTMalware". github.com. https://github.com/cyber-research/APTMalware (accessed July 27, 2023).
[50] A. Fadel, I. Tuffaha and malayyoub. "AI-SOCO". github.com. https://github.com/AliOsm/AI-SOCO (accessed July 27, 2023).
[51] Qi-AnXin. "APT_Digital_Weapon". github.com. https://github.com/RedDrip7/APT_Digital_Weapon (accessed July 27, 2023).
[52] Google. "VIRUSTOTAL". virustotal.com. https://www.virustotal.com (accessed July 27, 2023).
[53] Google. "VirusSign - Open Malware Database". virussign.com. http://www.virussign.com/ (accessed July 27, 2023).

Yuxia Sun (Member, IEEE) received the B.S. degree from Department of Computer Science, Huazhong University of Science and Technology, Wuhan, China, and the Ph.D. degree from the Department of Computer Science, Sun Yat-sen University, Guangzhou, China. She is currently an Associate Professor with Department of Computer Science, Jinan University, Guangzhou. She was a Research Associate with The Hong Kong Polytechnic University, Hong Kong SAR., and The University of Hong Kong, Hong Kong SAR., and was a Research Scholar with College of Computing, Georgia Institute of Technology, Atlanta, GA, USA. Her current research interests include machine learning, software safety, system safety, and software engineering.

Shiqi Chen is currently working toward MSc degree in College of Information Science and Technology, Jinan University, Guangzhou, China. Her current research interests include machine learning and software safety.

Song Lin received the MSc degree from College of Information Science and Technology, Jinan University, Guangzhou, China. His research interests include machine learning and software safety.

Aoxiang Sun is an undergraduate student in College of Information and Computational Science, Jilin University, Changchun, China. His current research interests include machine learning.

Saiqin Long received the BE in software engineering from Hunan Normal University and the Ph.D degree in computer applications technology at the South China University of Technology, in 2009, 2014, respectively. She was an appointed professor in the School of Computer Science, Xiangtan University, in 2017. Her research interests include cloud computing, cloud storage, parallel and distributed systems, file systems, computer system architecture. She is a member of Chinese Computer Federation (CCF).

Zhetao Li is a professor in College of Information Science and Technology, Jinan University. He received the B.Eng. degree in Electrical Information Engineering from Xiangtan University in 2002, the M.Eng. degree in Pattern Recognition and Intelligent System from Beihang University in 2005, and the Ph.D. degree in Computer Application Technology from Hunan University in 2010. He is a member of IEEE and CCF.