MalFocus: Locating Malicious Modules in Malware Based on Hybrid Deep Learning

Weihao Huang, Chaoyang Lin, Lu Xiang, Zhiyu Zhang, Guozhu Meng, Lei Xue, Kai Chen, Lei Meng, and Zongming Zhang

***Abstract*—In recent years, binary malware detection has attracted extensive attention from industry and academia. However, most of the existing work only focuses on judging whether a sample is malicious or not, rather than identifying malicious modules in malware. Few studies aiming at locating malicious code work on the function granularity and suffer from inaccuracy. In this article, we address this problem by locating malicious code at the functional module (*FM*) granularity, which combines several functions to express the malicious behaviors of malware. We design a tool called MalFocus to automatically divide malware into *FMs* and then identify the malicious functional module (*MFM*) in a multi-model hybrid manner, in which an unsupervised model and an interpretability approach based on a binary classifier are combined, eliminating the workload of labeling malware samples, determining the scope of *MFMs* and ranking them according to their maliciousness. The identified *MFMs* are then passed to security analysts for verification, helping to significantly reduce the scope of manual analysis while providing a comprehensive view of the malware attack flow. Additionally, rules derived from the verified *MFMs* can be used to detect variants and new malware families with different functionalities, offering a more general and flexible detection approach. We evaluate MalFocus's performance on 6764 real-world samples. The results show that MalFocus can correctly identify 95% of *MFMs*, outperforming current state-of-the-art work.**

***Index Terms*—Malicious code localization, functional module, deep learning.**

I. Introduction

Modern malware has been deeply bound to underground economics [1] and industrialized [2] to gain profits. Recent years have witnessed an undiminished trend of malware with over 450 K new malware samples constantly emerging daily [3]. This is propelled not only by new incentives such as the great profit of coin mining, but also the leverage of newly discovered software vulnerabilities and advanced evasive techniques. The fast-evolving malware makes the defenses ineffective [4], [5] and further threatens all the stakeholders.

To counteract the threats of malware, the industry has developed a de facto standardized pipeline [6], where software is first examined by automated detection engines. The automatic analysis spans from identifying known malware with accurate hash code, extracting static code features or dynamic runtime behaviors of maliciousness determination, to triaging malware into specific families. The detected malware may undergo a manual analysis where security analysts perform confirmation and localize the malicious code in the malware. The analysis results (e.g., file hash, indicators of compromise, and behavioral patterns) can be further integrated into the detection engines to improve detection performance. However, ineffective malware detection by automated engines may create many false alarms or miss a large number of real malware. It makes manual analysis a labor-intensive and time-consuming job in front of such a tremendous amount of malware.

The ineffectiveness of malware detection can be attributed to the deviation of malware comprehension. For example, a number of malware is created by injecting malicious code into a benign program, rather than being written from scratch [7], [8]. Thus, taking out code features from the entire malware can inevitably incur inaccuracy and degrade the performance. Unfortunately, the majority of current malware analysis and detection tools simply output a label of maliciousness [9], or at most determine their categories [10]. It is however unclear where the malicious code exactly resides and how it performs attacks. Although a few studies have pioneered localizing malicious code [6], [11], [12], the localization is conducted more on the granularity of functions, posing an implicit assumption that one function can be the evil of malicious behaviors. However, through our manual analysis of several mainstream families of malware, such as *Coinminer*, *Agent* and *Mirai*, one function oftentimes serves as a building block for malicious behavior but not all. This is a growing trend considering malware is becoming more complex and splitting malicious behaviors is beneficial for evading detection. Given that, we call the unit of a complete behavior "*functional module (FM)*" and the ones that conduct malicious operations are "*malicious functional module (MFM)*".

Received 28 September 2023; revised 5 March 2025; accepted 10 March 2025. Date of publication 14 April 2025; date of current version 4 September 2025. This work was supported by in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDB0690100, in part by Alibaba Innovative Research Program, in part by the Open Project Fund of Key Laboratory of Cyberspace Security and Defense under Grant 2024-MS-02, in part by NSFC under Grant 62372490 and Grant W2412110, and in part by CCF-NOSFUS Kunpeng Fund. *(Corresponding authors: Guozhu Meng; Lei Xue.)*

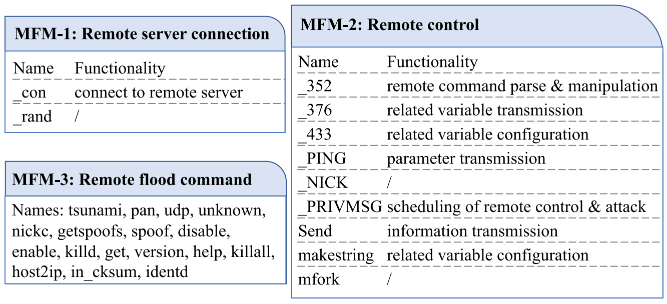
Weihao Huang and Lei Xue are with the School of Cyber Science and Technology, Shenzhen Campus of Sun Yat-sen University, Shenzhen 518107, China (e-mail: huangwh83@mail.sysu.edu.cn; xuelei3@mail.sysu.edu.cn).

Chaoyang Lin is with the Safety & CyberSecurity, NIO, Shanghai 201804, China (e-mail: cylin.cs@gmail.com).

Lu Xiang, Zhiyu Zhang, Guozhu Meng, and Kai Chen are with the Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100085, China (e-mail: xianglu.xyz@gmail.com; zhangzhiyu1999@iie.ac.cn; mengguozhu@iie.ac.cn; chenkai@iie.ac.cn).

Lei Meng and Zongming Zhang are with the Alibaba Cloud Computing, Hangzhou 311121, China (e-mail: shaoqi.ml@alibaba-inc.com; zongming.zzm@alibaba-inc.com).

Digital Object Identifier 10.1109/TDSC.2025.3561052



MFM-1: Remote server connection

Name	Functionality
_con	connect to remote server
_rand	/

MFM-2: Remote control

Name	Functionality
_352	remote command parse & manipulation
_376	related variable transmission
_433	related variable configuration
_PING	parameter transmission
_NICK	/
_PRIVMSG	scheduling of remote control & attack
Send	information transmission
makestring	related variable configuration
mfork	/

MFM-3: Remote flood command

Names: tsunami, pan, udp, unknown, nickc, getspoofs, spoof, disable, enable, killd, get, version, help, killall, host2ip, in_cksum, identd

Fig. 1. MFMs of a Tsunami malware.

There may be multiple *MFMs* in malware that collectively act to achieve the attack purpose. Fig. 1 shows one malware sample from family *Tsunami*, which is a popular IoT botnet family. After manual analysis, we find that there are mainly three *MFMs*, and each *MFM* contains several functions with certain behaviors, such as transmitting parameters via "_PING()" in *MFM-2*. It is inaccurate to determine its maliciousness since other benign software may also have such calls to, for example, test the connectivity of the network. However, the essence of this malware can be interpreted as remote control (i.e., commands parsing, parameters transmission, and information return) by combing all the necessary functional components "_352()", "_PING()" and "Send()". Therefore, if the malicious functionality is located at the function granularity, the one that originally performs normal operations may be misjudged as malicious, resulting in a high false positive rate. Meanwhile, as the abstraction level of the function granularity is lower, it may only complete a step of one malicious functionality and then cannot really achieve the purpose of localization. Thus, it is the *FM* that one malicious functionality should be located.

We propose MalFocus, an approach for locating *MFMs* through human-machine collaboration. This method reduces the scope of analysis for security analysts, provides a comprehensive view of the malware attack flow, and enhances malware detection in a more generalized and precise manner using the identified *MFMs*. The process begins by dividing the malware into *FMs*, each representing a single operation, based on the function call relationships. Next, MalFocus automatically localizes *MFMs* using an unsupervised deep learning model. An interpretable ranking manner is then applied to prioritize the results based on the maliciousness of the *MFMs*. Finally, manual analysis is employed to verify the effectiveness of the automatic model and the identified *MFMs*, enabling exploration of the malware attack path and detection of new malware families or variants.

More specifically, we utilize a community discovery algorithm to divide a malware sample into multiple *FMs*. Next, we determine the scope of *MFMs* using an unsupervised model, *AutoEncoder (AE)* [13], which is a neural network-based self-recovery model. The model is trained exclusively on benign samples, so it excels at recovering benign code but struggles with reconstructing malicious code. Based on this, we can locate *MFMs* by assessing their reconstruction performance. However, because some sensitive yet harmless operations may exist in benign samples, certain *MFMs* may be recovered unexpectedly well by the model. To address this, we propose a mask-based interpretability method using a binary classifier to refine the ranking of the *MFM* sequence by evaluating each *MFM*'s impact on the classification result. The automatically located *MFMs* are then passed to security analysts for final verification, who examine the *MFM* sequence, confirm the *MFMs*, and trace the attack paths based on the program's logical call relationships. Given that various types of malware may share similar *MFMs* (as confirmed in Section IV-D), we generate a set of rules from the verified *MFMs* to detect variants and new malware families.

To enable a comprehensive evaluation, we assess MalFocus with 6,764 malware samples from VIRUSSHARE and a cloud service provider (CSP). The experimental results demonstrate that MalFocus significantly reduces the scope of manual analysis by an average of 71%, with a maximum reduction of 99.6%. To evaluate the accuracy of *MFM* localization, we analyze 45 samples containing 689 sensitive functions and 164 *MFMs* labeled from VIRUSSHARE, as well as 39 samples with 271 sensitive functions and 108 *MFMs* labeled from CSP. MalFocus successfully detect 95% and 96% of the labeled *MFMs* on the two datasets, respectively. Additionally, we conduct a detailed manual analysis on eight samples to confirm that the located *MFMs* indeed exhibit malicious functionalities and reveal the malware's attack flow. Lastly, we generate rules for six *MFMs* located in two samples, which allow us to successfully identify variants and new families of different functional types.

Contributions: We summarize the contributions as follows.

- *Feature extraction & embedding:* We distill a list of instructions including 3,326 system and mainstream library calls and 20 types of sensitive instructions (e.g., memory manipulation and arithmetic operations), which build the basic skeleton of malware. Additionally, these instructions are further embedded for code representation via a learning-based approach.
- *Multi-model hybrid MFM location on the FM granularity:* In this paper, we present a tool called MalFocus designed to locate malicious functionality at *FM* rather than at the individual function granularity. We use a hybrid approach that integrates an unsupervised model with an interpretability algorithm to locate and rank *MFMs* based on their maliciousness. Our results demonstrate that this approach effectively identifies 95% of *MFMs*.
- *A novel tool that offers several key advantages:* 1) reduces the scope of manual analysis required by analysts; 2) pinpoints malicious functionalities at the FM granularity level; 3) comprehensively displays the malware attack flow by utilizing the located *MFMs* and the program's logical execution path; 4) generates templates based on the *MFMs*, enabling the detection of variants and new malware families of other functional types.

II. BACKGROUND & RELATED WORK

A. Malware Detection

Malware detection, in the view of feature extraction methods, falls into two categories–static and dynamic approaches. Static approaches [14] analyze and recognize malware without code execution, relying on features such as strings, instruction sequences, API calls, graphic structures (e.g., control flow graph and data flow graph), and hardware features [15]. As for malware binaries, security analysts usually perform disassembling with IDA Pro [16] or OllyDbg [17] to obtain high-level code representation. However, these static approaches are susceptible to malware evasion techniques [18], [19]. For instance, the malware creator may obfuscate or pack malicious code to evade feature harvest. An endless stream of bypass technologies gives birth to dynamic approaches [20]. Differently, dynamic detection recognizes malware by executing the code and observing the runtime behaviors. In particular, security analysts can monitor the information flow during code execution [21], [22], [23] to determine whether a security violation occurs, e.g., privacy leakage. They can also capture quantizable features such as API calls [24] hardware performance counter [25] or even runtime reports [26]. Not relying on code comprehension and analysis, dynamic approaches are thereby resistant to evasion techniques. However, it is time and resource-consuming for a sufficient dynamic analysis, and even impossible to trigger all functional behaviors.

To determine the genuine malware from suspicious candidates, we can employ pattern-matching or learning-based methods.

Pattern-matching-based determination: It is generally based on manually crafted rules, which are either identifiable signatures or more abstract code behaviors like behavioral graphs. Of the signature-based approaches, Karnik et al. [27] extract assembly instructions as signatures and make an exact comparison through cosine similarity to classify malware. Cha et al. [28] transform file hashes into vectors and detect multiple malware with this. Jang et al. [29] classify malware using automatic classifiers through graph metrics used in social network analysis. As to behavior-based ones, Park et al. [30] make classification by creating system call diagrams and Das et al. [31] combine system call frequencies and n-gram to identify new malware. M. Chandramohan et al. [32] model malware behavior based on a limited set of features and Aslan et al. [33] construct an intelligent behavior-based detection system utilizing both the learning-based and rule-based detection agents. Meng et al. [34] propose abstract attack models to accurately capture the semantics of various attacks, enabling better detection of hidden threats.

Learning-based determination: In the early days, people mostly adopt the method of machine learning. Masud et al. [35] take byte sequence, opcode and DLL as features to train Support Vector Machine (SVM). Schultz et al. [36] combine several Naive Bayes together and make a classification based on the voting result. Abou et.al [37] input the byte sequence to K-Nearest Neighbors (KNN) for classification. As deep learning gains tremendous success in image classification and speech recognition, researchers have started to adopt deep learning to detect malware. In contrast to traditional methods, deep learning-based approaches can learn deeper code semantics from amounts of data. Moskovitch et al. [38] utilize Artificial Neural Network (ANN) and opcode sequences extracted by n-gram for detection. Researchers transform the target program into the form of a picture and adopt Convolutional Neural Network (CNN) for detection, for example, He et al. [39] convert the program into an RGB image, Yan et al. [40] make it into a gray-scale image. Comparing program languages to natural ones, Tobiyama et al. [41] extract features utilizing Recurrent Neural Network (RNN) based on API sequence and make detection through a CNN model. Feng et al. [42] extract the manifest properties and API calls as features and utilize one customized RNN, obtaining better accuracy and efficiency. Kim et al. [43] combine several Deep Neural Networks (DNNs) together fitting features with different properties, thus digging deeper into semantics. Ke et al. [44] construct Long Short-Term Memory (LSTM) based on the semantic structure of Android bytecode for better detection accuracy.

B. Malicious Code Localization

Modern complex software is usually composed of multiple independent components to fulfill specific functionalities, termed as *modular programming* [45]. Meanwhile, developers are prone to utilize open-source projects to build a system or malware, rather than program from scratching, which indirectly raises the prevalence of *software composition analysis* [46]. Malware may only have a small portion of malicious code that in practice performs malicious intentions [47]. As a result, learning from the entire binary of malware may introduce much noise, i.e., benign functions. It is witnessed that researchers have been devoted to the research on localizing the real malicious code in malware, in order to obtain a better portrait for malware. For example, Ma et al. [48] detect malware based on the extracted API sequence and the bidirectional LSTM model, and utilize the Attention mechanism to determine the suspected API in the sequence. Meng et al. [49] localize malicious code by clustering commonalities shared by samples in the same malware family, and then propose a deterministic symbol automaton to present the malicious behaviors. Narayanan et al. [11] adopt a multi-view approach for malware detection and localize the malicious basic blocks through an interpretable manner on SVM which are then aggregated to arrive at the methods and classes encompassing them as well. Evan et al. [6] use a self-recovery model trained on white samples to determine abnormal basic blocks, and locate malicious functions based on the status of abnormal basic blocks in it. Pan et al. [15] collect the hardware performance of malicious files as features to train a deep learning detection model, and combine interpretability methods to determine the feature types that have a greater impact on the judgment results, but not positioning the malicious code in reality.

III. Design

As shown in Fig. 2, the workflow of MalFocus can be summarized in four steps. First, the binary malware is disassembled, and its corresponding features are extracted and embedded into vectors. Second, using a community discovery algorithm, the malware is divided into functional modules (*FMs*) based on its call graph (CG). Third, the malicious functional module (*MFM*) is located using a multi-model hybrid approach. An autoencoder (AE)-based model initially identifies the *MFM's* scope by assessing its reconstruction degree, and a mask-based method is applied to refine the location. Finally, the identified *MFM* is passed to analysts for a final decision. Simultaneously,

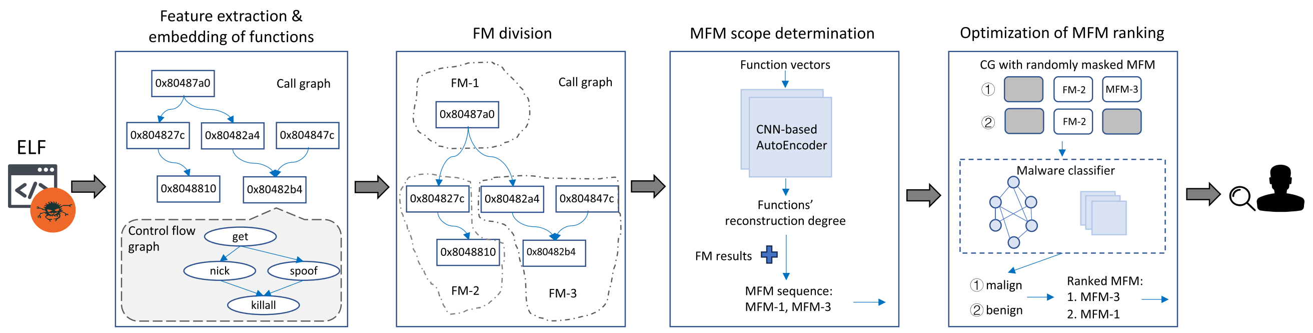


Fig. 2. Overview of MalFocus.

rules generated from the *MFM* are stored in a library as templates for detecting other variants or new malware families of other functional types.

A. Binary Disassembly

To streamline the analysis, we first disassemble the binary into instructions using *Radare2* [50]. Next, we construct a control flow graph (CFG) for each recovered function and a CG among them. Specifically, we traverse the "jmp" instructions within functions, extract the basic blocks, and establish their relationships to form the CFG. For each function, we identify calling instructions to determine the caller-callee relationships in the CG. Additionally, we eliminate redundant information from the samples. Certain functions, generated by the compiler during compilation, are considered "noise functions", they are typically small and lack meaningful semantics. We remove them by filtering the functions containing less than 10 instructions. For statically linked samples, we do not focus on the internal behavior of library functions but treat them as features to capture semantics. We filter them based on a collection of known library functions, which will be further detailed in Section III-B1.

Handling packed malware: The malware in binary format should be converted into an analyzable form. By retrieving the strings in them and matching the features of the packed sample, such as the "UPX" string, we find the packed ones and use the UPX tool [51] to unpack them. For these utilizing non-public packing or obfuscation strategies, since unpacking is another challenging topic and the focus of this paper is not on this, we directly discard them.

B. Feature Extraction & Embedding

Previous work has relied on features like API calls and strings for malware analysis, but most approaches focus on a single category [52], [53] or a limited set of features [54]. In contrast, we adopt a more comprehensive approach, targeting multiple categories of features closely tied to malicious behavior. This includes thousands of library/system calls and various sensitive instructions extracted from the CFG. Rather than using simple statistical methods [55], [56], we employ several deep learning-based embedding models for feature vectorization and select the most effective one.

TABLE I
ACCURACY OF THE MALWARE CLASSIFIERS BASED ON DIFFERENT EMBEDDING METHODS

Embedding	Model structure	Noise function	Accuracy
one_hot	GCN+FN	No	61.83%
node2vec	GCN+FN	No	**88.33%**
graph2vec	GCN+FN	No	83.33%
one_hot	GCN+FN	Yes	76.67%
node2vec	GCN+FN	Yes	56.33%
graph2vec	GCN+FN	Yes	53.67%

The value corresponding to 'node2vec without the noise functions' is highlighted in bold to emphasize its superior accuracy.

1) Library/System Calls: Malware interacts with the system through system calls. For instance, functions like "open()" and "close()" are used to manipulate files, while ELF binaries frequently rely on library functions like "socket()" for communication or "fork()" for subprocess creation. These calls are crucial features for identifying malware. To capture this, we extract all system calls from the Linux specification, along with 2,610 commonly used library functions, resulting in a total of 3,326 calls, which are then one-hot encoded. We traverse the instructions within each basic block, marking the invoked calls. If a basic block contains multiple calls, we split it accordingly. As a result, each basic block is represented by a single call, and those without calls are encoded as zero.

We use three embedding methods—*one-hot*, *node2vec*, and *graph2vec* [57]—to vectorize functions, as they have been shown to effectively capture code semantics. Specifically, we randomly select 2,000 samples from the VIRUSSHARE [58] dataset, spanning 10 malware families. From these, 500 samples are used to train the node2vec and graph2vec models, respectively. To compare the effectiveness of these embeddings, we construct three malware classifiers correspondingly using a graph convolutional network (GCN) model [59] combined with fully connected (FN) layers. Additionally, we evaluate the performance of the classifiers both with and without "noise functions." As shown in Table I, node2vec without the "noise functions" achieves the highest accuracy of 88.33%, even surpassing graph2vec, which is based on graph embedding. Library and system calls tend to cluster in specific basic blocks, forming localized behavior patterns that reflect particular malware actions, such as "file operations", "network communication", or "privilege escalation". Node2vec excels at capturing these localized patterns by simulating random walks through the CFG of a function and generating sequences of library/system calls. This approach effectively captures local features within basic blocks and their surrounding context, modeling the relationships between calls to produce more meaningful node embeddings. In contrast, Graph2Vec focuses on global features across the entire CFG, which may dilute or overlook critical local patterns, such as calls within specific basic blocks. In addition, as a deep learning-based approach, node2vec mines deeper semantic information compared to one-hot encoding, except in cases where "noise functions" interfere with the training of both node2vec and graph2vec. Thus, we select node2vec as the embedding method for library/system calls, resulting in a 256-dimensional vector.

2) Sensitive Instructions: In analyzing malware samples, we observe that some malicious operations, such as encryption, are executed entirely through arithmetic and logic operations without invoking library functions or system calls. To account for this, we identify 20 sensitive instructions that frequently appear in malware as follows.

Control instructions: We select four instructions that influence a program's control flow: "call" invokes a procedure, "ret" determines where the procedure returns, "jmp" performs an unconditional jump to a specified location, and "cmp" compares two operands, triggering conditional execution.

Arithmetic and logic instructions: Arithmetic instructions like "add", "sub", "mul" and "div" are often used in encryption and obfuscation techniques [60], [61]. Logic instructions, such as 'and", "or", "xor", "not", "shr" and "shl", reveal code semantics related to mathematical computation.

Data transfer instructions: These instructions demonstrate how a program manipulates memory. For instance, "push" and "pop" are used to manage the stack, while "mov" and "lea" handle moving and loading data from memory.

We count the occurrences of these instructions in each function and use the counts to create a feature vector. Since the library/system call feature vector consists of floating-point values while the sensitive instruction feature is represented as integers, their numerical ranges differ significantly. To prevent the model from being biased by these differences, we apply three feature normalization techniques: Logic of Data Analysis (LDA) [62], Cumulative Distribution Function (CDF) [63], and Genetic Programming (GP) [64]. By utilizing these methods to normalize the function vectors, we construct a malware binary classifier using the data and model structure described for verifying embedding methods in Section III-B1. As shown in Table II, the CDF-based GP achieves the highest accuracy and is selected as the final normalization method. LDA reorders the values in each dimension and converts them into percentages based on their positions in the sorted list. By comparison, CDF calculates the frequency of each value and transforms it into its corresponding cumulative distribution probability, preserving the original distribution characteristics more effectively. Furthermore, GP leverages evolutionary techniques to refine normalization based on CDF results, adaptively optimizing the process so that the numerical distribution of features better aligns with the deep learning model's feature extraction capabilities, ultimately enhancing its performance.

TABLE II
ACCURACY OF THE MALWARE CLASSIFIERS BASED ON DIFFERENT NORMALIZATION METHODS

Normalization	Model structure	Accuracy
Original	GCN+FN	84.27%
LDA	GCN+FN	84.27%
CDF	GCN+FN	94.58%
GP (LDA based)	GCN+FN	40.21%
GP (CDF based)	GCN+FN	**96.73%**

C. FM Division

Dividing malware into *FMs* presents two key challenges. First, the process must be automated and efficient to complete the division in a short time. Second, the division should be effective in two areas: 1) Structurally, the functions within each *FM* should be highly cohesive and strongly related, with minimal coupling between *FMs*; 2) Functionally, each *FM* should implement a specific task, and multiple independent *FMs* should work together to achieve the overall functionality of the malware. To address this, two main issues need solving: 1) The basis for division—what standards should guide the separation of *FMs*; 2) The division algorithm—what methods should be employed for effective segmentation. Since assembly code operates at a low level of abstraction, there's less information available to guide *FM* division. After manually analyzing various malware samples, we observe that functions within the same *FM* tend to call each other frequently, reflected by a high number of calling edges in the CG. Conversely, functions in different *FMs* interact less, often through a single interface function, resulting in fewer calling edges between *FMs*. Therefore, the call relationships between function nodes in the CG can serve as the basis for division. Functions with dense interconnections are more likely to belong to the same *FM*, and those with sparse connections are likely part of different *FMs*.

Given an undirected weighted graph $G = V, E, W$, where V is the set of nodes, E is the set of edges between nodes, and W represents the weights of the edges, we assume that the graph can be divided into k communities, denoted as $C = c_1, c_2, \ldots, c_k$. Here, c_i represents the ith community containing a subset of nodes, i.e., $c_i \subseteq V$, with $V = c_1, c_2, \ldots, c_k$. An edge is considered internal to community c_i if both connected nodes belong to c_i, and external if one node is in c_i while the other is outside it. The effectiveness of the community division is evaluated by Formula (1), where C represents the communities after division, x_i is the ratio of the sum of the weights of internal edges in the ith community to the total edge weight in the graph, and y_i is the ratio of the external edge weights to the total edge weight. The community discovery algorithm iterates by assigning nodes to communities step by step to maximize the modularity score Q at each state, continuing until Q reaches its maximum value. In our approach, we input the CG of the malware sample and set

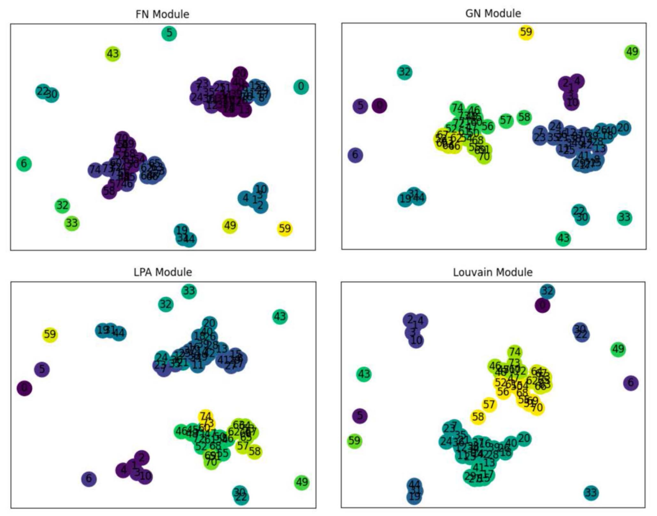


Fig. 3. FM division effect based on the four community discovery algorithm.

TABLE III
ACCURACY OF THE MALWARE CLASSIFIERS BASED ON THE FOUR COMMUNITY DISCOVERY ALGORITHMS

Algorithm	**Louvain**	**LPA**	**FN**	**GN**
Accuracy	72.86%	73.86%	**80.35%**	75.24%

The value corresponding to highlighted in bold to emphasize.

each edge's weight to one.

$$Q = \sum_{i=1}^{C} (x_i - y_i)^2 \tag{1}$$

We evaluate four commonly used community discovery algorithms: Louvain [65], LPA [66], FN [67], and GN [68]. Fig. 3 illustrates their *FM* division results on a sample from the RST, one well-known malware family for remote access and control, where each dot represents a function, and neighboring dots belong to the same *FM*. The differences in division results between the algorithms are apparent, prompting us to verify their effectiveness through both automatic and manual analysis. For the automatic analysis, we calculate the *FM* vector by averaging the vectors of the functions within each *FM*. Using these *FM* vectors and their call relationships, we construct a malware classifier. The data and model structure are the same as those used for verifying embedding methods in Section III-B1. We train four classifiers based on the *FM* division results from the four community discovery algorithms, with the accuracy of each shown in Table III. The classifier based on the FN algorithm achieves the highest accuracy, leading us to select FN as the preferred community discovery algorithm for *FM* division. FN primarily benefits from its greedy merging strategy based on modularity optimization, which effectively identifies *FMs* with dense internal connections and sparse external interactions. Comparatively, Louvain employs a hierarchical clustering approach based on modularity, making it well-suited for large-scale graphs like social networks. However, in malware CGs, which are significantly smaller in scale, Louvain may result in over-merging. LPA clusters nodes by propagating neighbor labels, a process with high randomness, leading to different partitioning results across runs. This variability is problematic for malware *FM* division, which demands stable and repeatable outcomes. GN identifies communities by globally computing edge betweenness and iteratively removing key edges. While it relies on global structural information, it may fail to preserve malware *FMs* with stronger local characteristics.

It's important to note that the goal of training the classifiers here is to identify the best community discovery algorithm, not to focus on the classifier's performance itself. To further confirm the effectiveness of the FN algorithm, we conduct manual analysis of the *FM* division results, which will be discussed in detail in Section IV-C.

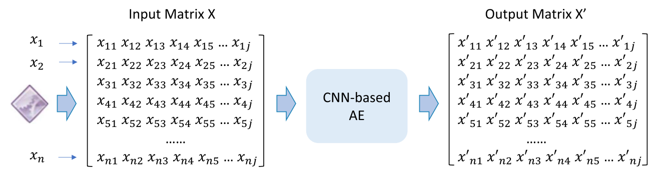

Input Matrix X
Output Matrix X'
x_1
x_2
x_n
CNN-based AE
$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15} \dots x_{1j}$
$x_{n1}\ x_{n2}\ x_{n3}\ x_{n4}\ x_{n5} \dots x_{nj}$
$x'_{11}\ x'_{12}\ x'_{13}\ x'_{14}\ x'_{15} \dots x'_{1j}$
$x'_{n1}\ x'_{n2}\ x'_{n3}\ x'_{n4}\ x'_{n5} \dots x'_{nj}$

Fig. 4. Construction process of the AE-based location model.

D. Multi-Model Hybrid MFM Localization

We locate the *MFM* using a multi-model hybrid approach. First, an AE-based model determines the scope of *MFMs* as a *MFM* sequence. Then, a mask-based interpretability method refines the ranking of the sequence, making the process more efficient for manual analysis.

1) MFM Location: We first determine the scope of the *MFMs* using a model based on AE. AE is an unsupervised model designed to reconstruct input data. It consists of two parts: the encoder, which compresses the input, and the decoder, which restores it. During training, the AE aims to make the output as similar as possible to the input, enabling it to learn key features from the training distribution during the compression process. When trained on benign samples, AE struggles to reconstruct malicious functionalities because it primarily learns benign characteristics where malicious behaviors are rare or nonexistent. Thus, malicious functionalities can be localized based on their reconstruction error. Instead of directly inputting the *FM* to AE for localization, we avoid embedding the entire *FM*, as embedding methods inevitably result in information loss, affecting localization accuracy. As shown in Table III, our classifier based on *FM-embedded* vectors performs significantly worse than one based on function-level vectors, highlighting the difficulty in effectively embedding *FMs*. Therefore, we train AE at the function granularity, evaluate the maliciousness of functions based on their reconstruction error, and locate the *MFM* by analyzing the functions within it.

For model construction, we customize AE according to our requirements. As illustrated in Fig. 4, considering the correlation between functions can improve the assessment of maliciousness. Instead of using individual functions as input, we train the model based on the difference between the input matrix of function

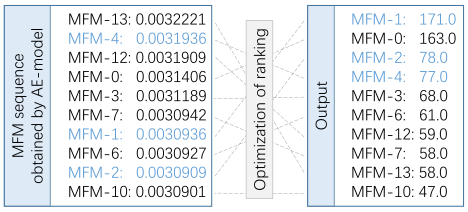


Fig. 5. Optimization of ranking brought by the mask-based approach on a sample from Mirai.

TABLE IV
ACCURACY OF THE MALWARE CLASSIFIERS TRAINED ON VIRUSSHARE & CSP RESPECTIVELY ("M" MEANS "MALWARE", "B" MEANS "BENIGN", "ACC" MEANS "ACCURACY")

Dataset	Size	Training		Testing		Acc
		M	B	M	B	
VIRUSSHARE	20,532	6,953	5,221	3,487	4,871	97.94%
CSP	19,225	6,574	4,386	3,277	4,988	96.04%

vectors $X(x_1, x_2, \ldots)$ and its output matrix X', as shown in Formula (3). Since the input is a vector matrix, we adopt a CNN-based AE model. CNN has proven effective in image processing, which also involves matrix representations. Additionally, with function vectors having 276 dimensions, CNN aids in dimensionality reduction, facilitating model training. To ensure consistency in input size, all input vector matrices must have the same dimensions. According to our statistics, 99% of samples contain fewer than 1,000 functions, so we assume a maximum of 1,000 functions per sample. Samples with fewer than 1,000 functions are padded with zero vectors, resulting in an input vector matrix of size 276*1 k.

$$RD\left(x_i, x_i'\right) = \left(x_i - x_i'\right)^2 \tag{2}$$

$$SRD\left(X, X'\right) = \sum_{i=1}^{n} \left(x_i - x_i'\right)^2 \tag{3}$$

$$FRD\left(X, X'\right) = \frac{1}{m} \sum_{i=1}^{m} \left(x_i - x_i'\right)^2 \tag{4}$$

Specifically, for a target malware sample $\boldsymbol{M}$ that has been divided into *FMs*, we first obtain its vector matrix $X(x_1, x_2, \ldots, x_n)$ and input it into the AE-based model. The corresponding output is the vector matrix $X'(x_1', x_2', \ldots)$. Using Formula (1), we calculate the reconstruction error for each function and arrange them in descending order, resulting in the sequence $F(f_1, f_2, \ldots)$. A threshold value y is then applied, specifying that the top y% of functions in this sequence are of interest for further analysis. These functions are marked as set $S\{s_1, s_2, s_3, \ldots\}$. Next, let the *FMs* of $\boldsymbol{M}$ be represented as the set $FM\{fm_1, fm_2, \ldots, fm_i, \ldots\}$. If any fm_i contains a function belonging to S, we include it in the set H. For each *FM* in H, we measure its reconstruction error using Formula (4), sort them in descending order, and obtain the sequence N, which represents the scope of the *MFMs* for malware $\boldsymbol{M}$.

2) MFM Location Ranking: The AE-based model can significantly reduce the manpower and resources required for collecting and labeling malware samples. However, there is a possibility that some benign samples may contain small portions of malicious functionality, which could lead to certain *MFMs* achieving a higher reconstruction accuracy, resulting in an inaccurate ranking for localization. As shown in the left part of Fig. 5, while AE produces an MFM sequence, the blue-marked MFMs that have been manually verified often rank lower in the list, causing confusion for analysts who must verify MFMs in sequence. This is an inherent limitation of AE that cannot be resolved through model optimization, such as parameter tuning or changes to the network architecture.

To address this, we design an *MFM* localization ranking algorithm based on a mask-based approach to further refine the *MFM* sequence produced by AE, aiming to align the rankings more closely with the actual maliciousness of the *MFMs*. The mask-based approach is a model interpretability method that assesses the importance of different parts of the code by removing them and observing changes in the model's predictions, thereby identifying the key features in the code that contribute to malicious behavior. We construct a malware classification model and randomly alter the features within the *MFMs* identified by the AE-based model, comparing their impact on the prediction results. Naturally, the *MFMs* that exert a greater influence on the model's predictions are likely to correspond to higher maliciousness.

Specifically, we first develop a neural network-based malware classifier as the benchmark for interpretability analysis. Since the determination of *MFMs* depends largely on their impact on classification outcomes, ensuring high prediction accuracy is critical. To address this, we consider three factors: 1) the data imbalance across various malware families [69], as illustrated in Tables VI and VII; 2) the focus on whether a sample is malicious or benign, which simplifies the task to binary classification; and 3) the observation that fewer categories typically result in higher classifier accuracy. Accordingly, we construct a binary classification model that determines whether a sample is benign or malicious. Using the CG and function embeddings described in Section III-B, we combine a graph neural network (GNN) with a convolutional neural network (CNN) to form the malware binary classification model. The GNN processes the entire CG, taking the function features and their call relationships as input to generate a new representation for each node. This representation captures both the semantic information of the functions and the structural information from their call relationships. The output is then passed through the CNN for final classification. We combine the two malware sample datasets with the benign one, obtain the training sets, and construct a classifier based on them respectively, which is used for subsequent interpretability analysis. The sample distribution and classifier prediction accuracy are shown in Table IV, from which we can see that the accuracy of both classifiers is above 96%, meeting the needs of interpretability analysis. For a given malware sample $\boldsymbol{M}$, we randomly select some *MFMs* from the sequence N, remove their function features and internal call relationships, and generate 1,000 new variants of each malware sample. These samples are then classified, resulting in 1,000 predictions per sample. For the samples classified as benign, we count the frequency of *MFM* that are removed. Those with a higher removal frequency are ranked higher in the *MFM* sequence.

E. MFM-Based Malware Discovery

MFMs, being the essential components that perform malicious actions, can be utilized to recognize new variants or malware families exhibiting similar behaviors. During our analysis, we observe phenomena that support this assumption. For instance, encryption functionality is present in both the Coinminer and Dofloo malware families. With this in mind, we propose leveraging the features of these *MFMs* to discover new malware. Specifically, we convert the *MFMs*, verified through manual analysis, into YARA [70] rules. YARA is an open-source tool designed to assist researchers in identifying and classifying malware samples. A YARA rule consists of a series of strings and a Boolean expression that explains its logic. In our approach, we primarily use the extracted features as matching strings in YARA rules, including: 1) function names, 2) the names of functions they call, 3) library/system call numbers, and 4) quoted text strings. For each *MFM*, YARA rules are generated for each function it contains, and only when all functions match can the sample be classified as malicious.

IV. Evaluation

Implementation & Experiment Settings: We use 90% of the benign samples to train the AE-based model, reserving the remaining 10% for testing. To enhance the model's reconstruction ability, we implement three skip connections, achieving a training reconstruction difference of 9.0757e-05 and a testing difference of 1.0128e-04. For the malware classifier employed in interpretability analysis, we first construct a Gated Graph Recurrent Layer to embed the function vectors. These are then passed through two 3×1 Conv1D layers and two 3×1 max-pooling layers, using the ReLU activation function to extract key features. Finally, the output is sent to two fully connected layers, with a sigmoid function at the end for binary classification. The epoch count is set to 200. Both models are trained on a server equipped with 4 NVIDIA GP100GL GPUs, taking approximately 5 hours for the AE model and 16 hours for the malware classifier.

We conduct extensive experiments to answer the following research questions.

RQ1: Can MalFocus enhance the efficiency of malicious code localization?
RQ2: Can MalFocus achieve higher accuracy in locating malicious code?
RQ3: Can the malicious code located by MalFocus indeed capture the malicious functionality and the entire attack flow as well?
RQ4: Can MalFocus contribute to future malware detection efforts?
RQ5: Can MalFocus effectively resist interference from code obfuscation?

TABLE V
The Distribution of Benign Samples

Linux OS	**Package manager**	**# Samples**
Ubuntu 20.04	apt	8,994
Ubuntu 20.04	snap	6,364
CentOS 7	yum	6,399
Fedora 35	dnf	7,708

TABLE VI
The Distribution of Malware Samples From CSP

Category	Size	Category	Size	Category	Size
Agent	354	Meterpreter	397	Rekoobe	539
Cleanlog	106	Mirai	2225	Rootkit	220
Coinminer	316	Osf	34	Rst	107
Downloader	85	Pnscan	781	Scanner	35
Exploitscan	95	Pomedaj	82	Small	79
Gafgyt	453	Prometei	286	Sshdoor	398
Getshell	34	Proxy	82	Tsunami	627
Kryptik	54	Prtscan	2387	Vit	48
Xorddos	27				

TABLE VII
The Distribution of Malware Samples From VirusShare

Category	Size	Category	Size	Category	Size
Agent	428	Gafgyt	5024	Rst	256
BitCoinMiner	95	Mirai	3446	Small	321
Dofloo	167	Osf	124	Tsunami	317
Ganiw	262				

A. Experiment Data

Benign samples: To construct the AE-based *MFM* localization model effectively, it is essential to ensure the coverage and diversity of benign samples. A comprehensive representation of normal functionalities enhances the model's ability to detect malicious parts effectively. To achieve this, we collect benign samples using the Linux Package Manager, the primary method for downloading or installing applications on Linux. We utilize four package managers—apt, snap, yum, and dnf—to cover most mainstream options. Table V presents statistics on the collected samples, categorized by software repository and supported Linux version. We apply three criteria for further filtering: 1) *uniqueness*, where samples with the same hash code or differing only by version are retained as a single copy; 2) *moderate size*, to ensure efficient training; and 3) *benignity*, where samples are examined using VirusTotal [71] to exclude those with any warning of maliciousness. As a result, we obtain a total of 14,728 benign samples.

Malware: For an efficient evaluation, malware samples should be diverse in category and plentiful in number. To achieve this, we collect samples from the VirusShare [58] repository and real-world samples from one of the largest cloud service providers (CSP). Specifically, we obtain 13,581 samples from VirusShare. After filtering out samples without valid labels or with fewer than 10 functions, 10,440 samples are retained, spanning 10 categories, as shown in Table VII. Similarly, we collect 9,851 samples from the CSP, distributed across 25 categories, as detailed in Table VI. Notably, the malware families *Mirai*

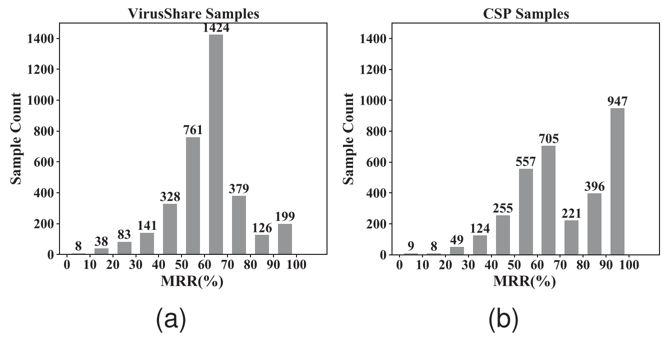


Fig. 6. The distribution of malware's amount in different reduction ranges.

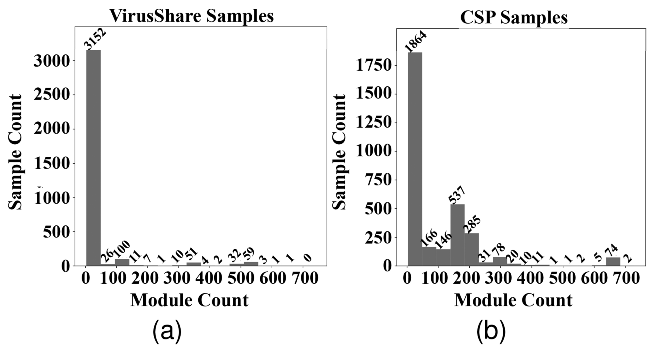


Fig. 7. The distribution of malware's FM amount.

and *Gafgyt* dominate these categories, comprising 22.59% and 48.12% of the samples, respectively.

B. Overall Effectiveness

1) Reduce the "Manual Analysis Scope": Both static- and dynamic-based approaches can trigger false alarms, leading to a labor-intensive process for malware confirmation. According to the CSP, during routine analysis, security analysts often rely on indicators of compromise (IoCs), such as sensitive APIs or suspicious IPs, to quickly narrow the scope of a complex software system for further manual investigation. Once narrowed, they can rigorously examine the pivot code regions, which are hereby defined as the "*manual analysis scope*". However, inaccurate or misleading IoCs can have catastrophic consequences, forcing analysts to sift through vast amounts of code. Given that millions of samples are analyzed daily and await manual confirmation, we evaluate the efficiency of our approach in reducing the "*manual analysis scope*" through a uniform metric, the "Module Reduction Ratio (MRR)," which is calculated as:

$$MRR = (1 - (MN/TN)) * 100\% \tag{5}$$

among which *MN* refers to "the number of *MFM* located by MalFocus" and *TN* refers to "the total number of *FM*", as the analysts just need to manually focus on the *MFMs* for further verification.

Specifically, we analyze all test samples from the VIRUSSHARE and CSP datasets (shown in Table IV) using MalFocus. The analysis takes approximately 2 hours, with each sample averaging only about 1 s to process. For VIRUSSHARE, the average *MRR* is 61%. The highest *MRR* occurs with a sample from the BitCoinMiner family, reaching 99.24%, originally containing 3,438 *FMs*. The lowest *MRR*, from the Mirai family, has only 3 *FMs*, all of which are malicious. For CSP, the average *MRR* is 71%, with the highest from a Coinminer sample, achieving 99.60%, originally containing 1,151 *FMs*. The lowest, again from the Mirai family, contains only 3 *FMs*, all of which are *MFMs*.

Fig. 6 shows the distribution of sample counts across different reduction ranges. For VIRUSSHARE, 1,424 samples fall within the [60%, 70%] range, which includes the largest number of samples. In contrast, for CSP, the range [90%, 100%] contains the most samples. This is because CSP samples generally have a higher number of *FMs*, as clearly shown in Fig. 7, which illustrates the *FM* distribution across both datasets. Fig. 8 presents the average *MRR* across different malware families for VIRUSSHARE and CSP, respectively. In the case of VIRUSSHARE, the Ganiw family, a DDoS Trojan with multiple modules, achieves the highest *MRR* at 96.43%, with an average *FM* count of 578. The Osf family has the lowest *MRR*, at just 34.65%, with an average *FM* count of 29. For CSP, the Meterpreter family, another popular Trojan family, has the highest *MRR* at 96.44%, with an average *FM* count of 578. The Vit family has the lowest *MRR* at 30%, with an average *FM* count of only 10. This suggests that the *MRR* for each category is proportional to its average *FM* count.

2) Localization Efficiency: After confirming the labor-saving benefits, it is crucial to further evaluate and analyze the accuracy of the model's localization. As with any malware detection and localization tool, false positives and negatives are inevitable, and the *MFM* sequences generated by MalFocus may also contain misjudgments or omissions.

MFM label: We select a set of samples from the VIRUSSHARE and CSP datasets as evaluation objects, considering factors such as representativeness, diversity, and novelty. For the VIRUSSHARE samples, we randomly select six samples from the *Gafgyt*, *Mirai*, *Tsunami*, *Agent*, and *BitcoinMiner* families, and three samples from the remaining families. *Gafgyt*, *Mirai*, and *Tsunami* are notorious for creating botnets used in DDoS attacks, while *Agent* is highly adaptable, having been customized into various forms such as Trojans, worms, and backdoors. *BitcoinMiner* represents the growing trend of cryptocurrency mining malware, specifically targeting Bitcoin. For the CSP samples, we apply a similar method, selecting two samples from each of the 14 families with over 100 samples, including *Gafgyt*, *Prtscan*, *Rebooke*, and others, and one sample from each of the remaining 11 families. We label the test samples from both datasets semi-automatically, balancing quantity with accuracy. First, we use VirusTotal's detection engine alongside CSP's internal sandbox for dynamic analysis, recording sensitive operations during execution. We focus on three categories of sensitive operations: process behavior (e.g., `system()`, `execve()`, `popen()`), file behavior (e.g., `read()`, `write()`, `fgets()`), and network behavior (e.g., `connect()`, `recv()`, `send()`, `socket()`). The samples are then labeled using the MITRE ATT&CK [72]

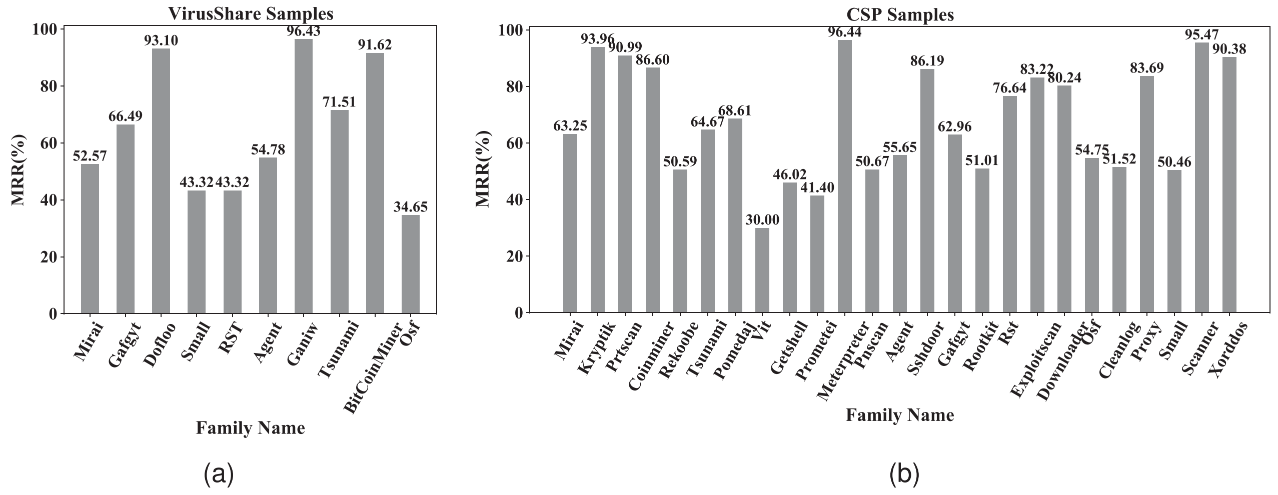


Fig. 8. The average reduction workload of different malware families.

TABLE VIII
THE PERFORMANCE OF MALFOUCS COMPARED TO OTHER STATE-OF-THE-ART SYSTEMS ON THE METRICS ACCURACY, PRECISION, RECALL, FL-SCORE, AND COMPLEXITY

Tools	Accuracy		Precision		Recall		F1-Score		Comple
	VS	CSP	VS	CSP	VS	CSP	VS	CSP	
MalFocus	81%	82%	39%	30%	95%	96%	55%	45%	0.56s
CAPA	71%	70%	38%	30%	66%	59%	48%	40%	2.04s
SHAP	68%	75%	35%	34%	64%	65%	41%	45%	59.4s

framework correspondingly. Next, we identify the functions responsible for performing these behaviors and label the *FM* containing them as *MFM*. In total, for VIRUSSHARE, 689 functions are identified, and 164 *FM* are labeled as *MFM*; for CSP, 271 functions are identified, and 108 *FMs* are labeled as *MFM*.

MFM location: To demonstrate MalFocus's performance and compare it with state-of-the-art systems, we review similar approaches. While many tools exist for malware identification and classification, few are specifically designed to locate malicious code and assist security analysts. For this study, we select two representative tools: CAPA [73], a traditional rule-based analysis method, and SHAP [74], a deep learning-based interpretability tool. We use four metrics—*accuracy*, *precision*, *recall*, and *F1-Score*—to comprehensively assess their localization performance. Additionally, we introduce the *complexity* metric to assess the analysis efficiency of the three tools, measured by the average analysis time per sample. Metrics are recorded for each sample, and the average results are shown in Table VIII. MalFocus achieves a recall of 95% on VIRUSSHARE and 96% on CSP, meaning it locates nearly all *MFMs*, significantly outperforming CAPA (66%, 59%) and SHAP (64%, 65%), guaranteeing the effect of reduction in manual analysis in Section IV-B1. MalFocus also demonstrates a notable improvement in *accuracy*. CAPA, being a rule-based tool, relies on manually crafted features like "HTTP server connections" and "file read/write" operations, which limits its generalization capabilities. In contrast, MalFocus builds an unsupervised model focused on benign samples, effectively addressing generalization challenges. SHAP locates malicious regions by quantifying their influence on the classifier's predictions, but its effectiveness depends heavily on the classifier and assumes independence between regions, which can introduce biases. MalFocus's AE model, however, leverages unsupervised learning and captures relationships between functions, providing a more comprehensive analysis.

While MalFocus's *F1-Score* is lower than expected due to its *precision*, it is important to note that the *MFM* will undergo manual verification, significantly reducing the scope of manual analysis, as shown in Section IV-B1. Since MalFocus is designed to assist security analysts, its primary goal is to ensure no malicious sections are overlooked, allowing analysts to identify regions responsible for malicious actions. Therefore, tools like this often prioritize minimizing false negatives, even at the expense of higher false positives, which also affects the *precision* of both CAPA and SHAP. Nevertheless, MalFocus has considerably narrowed the scope of analysis and provides a more comprehensive view of malicious behavior compared to CAPA and SHAP's function-level localization. Additionally, the semi-automated *MFM* labeling process may have missed some *MFMs*, which impacts the *precision*. To address this issue, we conduct a fully manual analysis on selected samples, leading to improved *precision*, as detailed in Section IV-C. Notably, MalFocus's *complexity* is only 0.56 seconds, significantly lower than CAPA's 2.04 seconds, a traditional rule-based tool. Moreover, it far outperforms SHAP, which has a *complexity* of 59.4 seconds due to its use of a recursive algorithm that requires multiple passes through the neural network.

MFM location ranking: To evaluate the ranking optimization achieved through the mask-based approach, we employ an "effort awareness" method to measure efficiency improvements during the analysis process. Specifically, we calculate the percentage of *MFMs* detected at each phase of analysis. For example, as shown in section (a) of Fig. 9, when 30% of the *MFM* sequence is analyzed, the AE-based model locates 47.19% of the *MFMs*, while the hybrid model detects 63.31%,

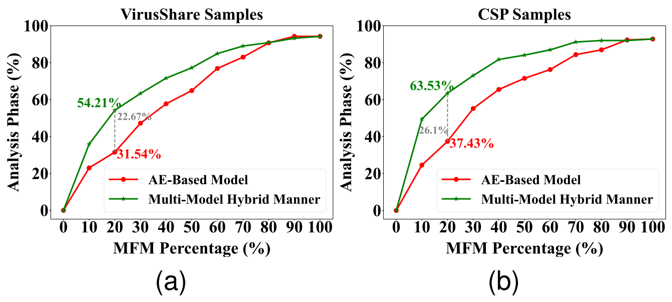


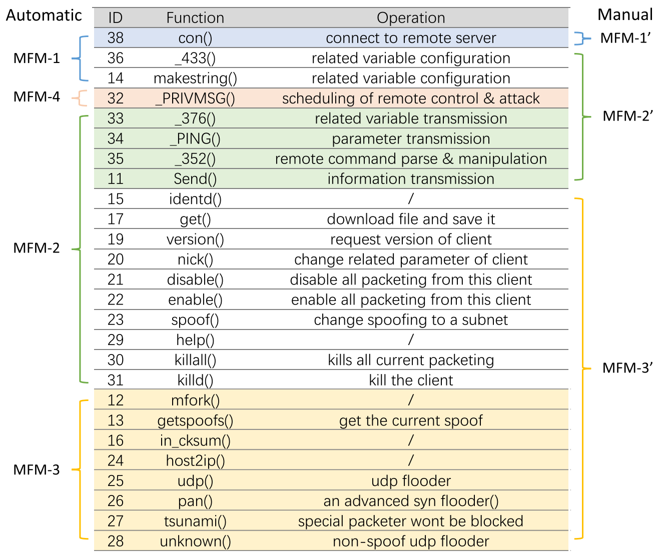
Fig. 9. Comparison of the effort awareness between multi model hybrid manner & AE-based model.

demonstrating the superiority of the hybrid approach. We divide the entire analysis process into ten intervals: [0, 10%], [10%, 20%], ..., [90%, 100%], and calculate the percentage of *MFMs* detected at each key point (10%, 20%, ..., 100%). Section (a) of Fig. 9 presents the results for the VIRUSSHARE dataset. In the range [0, 80%] of the analysis process, the percentage of *MFMs* detected by the hybrid model consistently exceeds that of the AE-based model, with the highest increase at the 20% mark, reaching 22.67%. This indicates that the hybrid model detects more *MFMs* earlier in the process, significantly reducing the analysts' workload. For the CSP dataset, the results are even more promising, as shown in section (b), where the percentage of *MFMs* detected by the hybrid model is higher across a broader range [0, 90%], with a peak of 26.11% at the 20% mark.

Summary: It can be concluded that the overall *MRR* achieved by MalFocus is proportional to the number of *FMs*, potentially reaching a significant level. This indicates that the scope of manual analysis can be greatly reduced, while maintaining lower analysis time complexity, thus addressing RQ1. Additionally, although some normal *FMs* may be misclassified, MalFocus successfully detects nearly all *MFMs*. The integration of manual analysis in the later stages further verifies the *MFMs*, ensuring accurate localization, which addresses RQ2.

C. Focus Analysis

The samples labeled through the semi-automatic method may present the following flaws: 1) inability to cover all functions performing sensitive operations, potentially missing some *MFMs*; 2) lack of visibility into the specific operations carried out by the *MFMs*; and 3) difficulty in fully understanding the malware's attack flow. To address these limitations, we select 4 samples each from the VIRUSSHARE and CSP datasets, focusing on three representative malware families: *Agent*, *Tsunami*, and *Rootkit*, known for botnet attacks, flexibility, and concealment, respectively. We manually partition these samples into *FMs* and identify the *MFMs*, evaluating the effectiveness of our approach. In total, we identify 90 sensitive functions and 18 *MFMs* in the VIRUSSHARE dataset, and 40 sensitive functions and 15 *MFMs* in the CSP dataset. The *accuracy*, *precision*, *recall*, and *F1-Score* for the VIRUSSHARE samples are 90%, 51%, 95%, and 66%, respectively, while for the CSP samples, they are 84%, 43%, 94%, and 59%. Notably, the *precision* and *F1-Scores* show a significant improvement compared to Section IV-B2, with increases of (12%,11%) and (13%, 14%) for the VIRUSSHARE and CSP samples, respectively. Meanwhile, the recall remains consistent as before.



Automatic	ID	Function	Operation	Manual
MFM-1	38	con()	connect to remote server	MFM-1'
MFM-1	36	_433()	related variable configuration	MFM-2'
MFM-1	14	makestring()	related variable configuration	MFM-2'
MFM-4	32	_PRIVMSG()	scheduling of remote control & attack	MFM-2'
MFM-2	33	_376()	related variable transmission	MFM-2'
MFM-2	34	_PING()	parameter transmission	MFM-2'
MFM-2	35	_352()	remote command parse & manipulation	MFM-2'
MFM-2	11	Send()	information transmission	MFM-2'
MFM-2	15	identd()	/	MFM-3'
MFM-2	17	get()	download file and save it	MFM-3'
MFM-2	19	version()	request version of client	MFM-3'
MFM-2	20	nick()	change related parameter of client	MFM-3'
MFM-2	21	disable()	disable all packeting from this client	MFM-3'
MFM-2	22	enable()	enable all packeting from this client	MFM-3'
MFM-2	23	spoof()	change spoofing to a subnet	MFM-3'
MFM-2	29	help()	/	MFM-3'
MFM-2	30	killall()	kills all current packeting	MFM-3'
MFM-2	31	killd()	kill the client	MFM-3'
MFM-3	12	mfork()	/	MFM-3'
MFM-3	13	getspoofs()	get the current spoof	MFM-3'
MFM-3	16	in_cksum()	/	MFM-3'
MFM-3	24	host2ip()	/	MFM-3'
MFM-3	25	udp()	udp flooder	MFM-3'
MFM-3	26	pan()	an advanced syn flooder()	MFM-3'
MFM-3	27	tsunami()	special packeter wont be blocked	MFM-3'
MFM-3	28	unknown()	non-spoof udp flooder	MFM-3'

Fig. 10. MFMs in malware family *Tsunami*. This sample is identified with four MFMs by our approach (in the left) and three MFMs by manual analysis (in the right).

Particularly, we select a sample from the *Tsunami* family to analyze the functionalities of the *MFMs* located and compare them with those manually divided and identified. This comparison evaluates whether MalFocus accurately locates the malicious functionality and assesses the effectiveness of its *FM* division. Since this malware launches DDoS attacks via a botnet, we label functions related to "carrying out DDoS attacks", "random mutation generation," "C&C remote interaction," and "root permission operations" as sensitive, as shown in Fig. 10. Based on the *FM* division algorithm, we mark the *FMs* containing these sensitive functions as *MFMs*. We then input this sample into MalFocus for analysis, and the labeled *MFMs* rank in the top four positions in the *MFM* sequence. Upon analyzing these four *MFMs*, we identify their functionalities as follows: "remote server connection" (*MFM-1*), "remote command transmission" (*MFM-2*), "remote flood attack" (*MFM-3*), and "scheduling of command transmission and flood attack" (*MFM-4*). These *MFMs* execute sequentially to carry out the attack, demonstrating that our approach successfully identifies the key *MFMs*.

Meanwhile, we manually partition the sample into *FMs* and identify the corresponding *MFMs* as "remote server connection" (*MFM-1'*), "remote control" (*MFM-2'*), and "remote flood attack"(*MFM-3'*). As shown in Fig. 10, there are some inconsistencies between the localization results, leading to the following conclusions. First, most discrepancies arise from functions that are not critical to the corresponding *MFM*, such as _433() and makestring(), which handle string operations, and enable(), disable(), get(), version(), and help(), which print corresponding information post-operation. Second, all essential pivot functions are accurately identified and align with human analysis. For instance, the con() function is responsible for creating a socket, binding it to a remote server, and performing related configurations, making it a crucial component of the remote server connection. Similarly, tusunami(), udp(), and pan() generate the payload for the flood attack, naturally placing them within the *MFM* for remote flood attacks. Third, MalFocus further refines the manually identified *MFM-2'* (remote control) by splitting it into two *MFMs: MFM-1* (remote command transmission) and *MFM-4* (scheduling the command transmission and flood attack). Although *MFM-4* only includes one sensitive function, PRIVMSG(), it carries out a distinct and complete functionality, justifying its classification as a separate *MFM*.

Summary: Comparing the *MFM* localization results from both automatic and manual methods reveals that MalFocus effectively identifies the malicious functionalities. Despite some differences, the key components are consistently located by MalFocus. Additionally, it captures the complete attack flow of the malware sample. In certain scenarios, MalFocus provides a more detailed MFM localization. Therefore, this answers RQ3.

D. Detection of Variants & New Families

To assess whether the identified *MFMs* can aid in detecting malware variants or new families of different functionalities, we focus on two of the most prominent botnet malware families, *Tsunami* and *Mirai*, and extract six *MFMs* from them.

1) Detection of Variants: The *MFM* templates matched here include the four *MFMs* from the Tsunami sample discussed in Section IV-C. The *MFMs* used for generating the YARA rules correspond to "remote server connection" (*MFM-1*), "remote command transmission" (*MFM-2*), "remote flood attack" (*MFM-3*), and "scheduling" (*MFM-4*). To ensure both accuracy and scalability, we do not include all sensitive functions from each *MFM* in the rules generation. Instead, we select those present in both the automated and manual analysis results, as shown in the overlapping sections of Fig. 10. The functions chosen for each *MFM* are as follows: 1) *MFM-1:* con(); 2) *MFM-2:* _352(), _376(), _PING(), Send(); 3) *MFM-3:* tsumani(), pan(), udp(), unkown(), getspoofs(), host2ip(), in_cksum(); and 4) *MFM-4:* _PRIVMSG().

The results show that two samples are matched, belonging to the Gafgyt and Dofloo families, respectively. Gafgyt is the most active IoT botnet family after Mirai, while Dofloo is a widely-used botnet known for employing encrypted remote commands. Both malware families are derived from the architecture and design of Mirai and Tsunami, as well as their variants. The matched *MFMs* and their corresponding functions are listed below.

Gafgyt sample: 1) Remote connection (*MFM-1'*): initConnection(); 2) Remote command transmission (*MFM-2'*): recv_line(); 3) Remote flood attack (*MFM-3'*): audp(), atcp(), astd(), vseattack(), getRandomIp(), check_sum_tcp_udp(), cncinput(). The correspondence between these and the template is shown in Fig. 11. It can be observed that Gafgyt's *MFM-3'* corresponds to both *MFM-3* and *MFM-4* in the template, as the function _PRIVMSG() is treated as a single *MFM* in Tsunami,

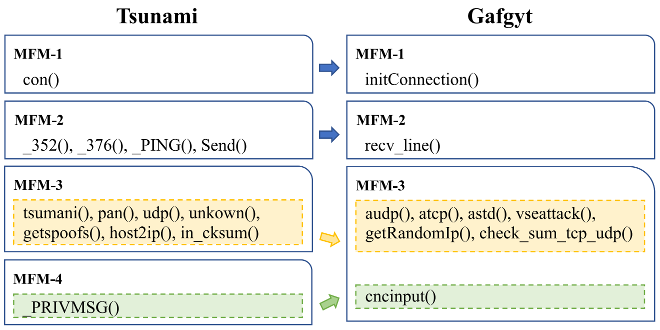

Tsunami
Gafgyt
MFM-1
con()
MFM-1
initConnection()
MFM-2
_352(), _376(), _PING(), Send()
MFM-2
recv_line()
MFM-3
tsumani(), pan(), udp(), unkown(), getspoofs(), host2ip(), in_cksum()
MFM-3
audp(), atcp(), astd(), vseattack(), getRandomIp(), check_sum_tcp_udp()
MFM-4
_PRIVMSG()
cncinput()

Fig. 11. Correspondence between Tsunami sample & the templates.

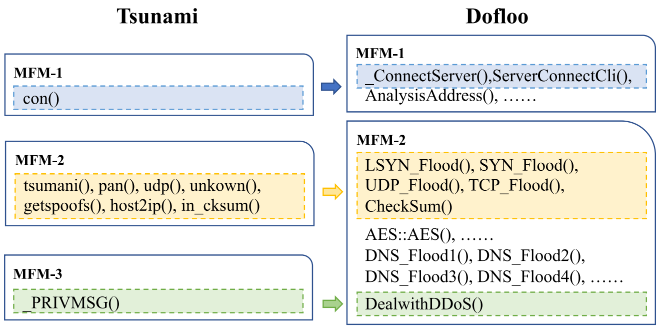

Tsunami
Dofloo
MFM-1
con()
MFM-1
_ConnectServer(),ServerConnectCli(), AnalysisAddress(), ……
MFM-2
tsumani(), pan(), udp(), unkown(), getspoofs(), host2ip(), in_cksum()
MFM-2
LSYN_Flood(), SYN_Flood(), UDP_Flood(), TCP_Flood(), CheckSum()
AES::AES(), ……
DNS_Flood1(), DNS_Flood2(), DNS_Flood3(), DNS_Flood4(), ……
MFM-3
_PRIVMSG()
DealwithDDoS()

Fig. 12. Correspondence between Dofloo sample & the templates.

whereas in Gafgyt, it is included within *MFM-3'* as a sensitive function rather than an independent module. This suggests that, compared to Gafgyt, Tsunami exhibits a higher degree of modularity. Additionally, in Gafgyt, *MFM-2'* is mainly handled by a single function, recv_line(), while in Tsunami, the operation is split across four functions—_352(), _376(), _PING(), and Send()—which sequentially handle command parsing, parameter transmission, and information return. This reflects Tsunami's more fine-grained implementation of functionality. Overall, while Tsunami and Gafgyt, as botnet family members, share most functionalities, Tsunami demonstrates a higher degree of modularity and finer granularity in function execution. This supports the notion that Tsunami is a more evolved variant of Gafgyt, a trend consistent with the real-world development of malware. It also highlights the shift toward increasingly modular and fine-grained malware design.

Dofloo sample: 1) Backdoor module (*MFM-1'*): _ConnectServer(), ServerConnectCli(); 2) Remote DDoS attack module (*MFM-2'*): LSYN_Flood(), SYN_Flood(), UDP_Flood(), TCP_Flood(), CheckSum(), DealWithDDos(). The correspondence between these functions and the template is shown in Fig. 12. It can be observed that only certain components of Dofloo are matched, as its functionality is much more redundant compared to the template. For example, Dofloo not only supports flood attacks like the template but also DNS attacks and encrypted communication. Despite this, Dofloo is still identified, as matching even one template is sufficient for recognition. This approach is effective for malware that exhibits characteristics of multiple types, such

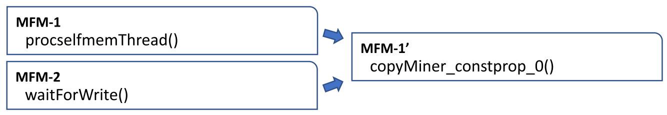


Fig. 13. Correspondence between Pomedaj sample & the templates.

as Dofloo, which functions as both a backdoor and a botnet, and even performs encryption operations like Bitcoin miners. Traditional classification methods often struggle to detect such multi-faceted malware.

2) Detection of New Families: The matched *MFM* templates come from the Mirai family, while the sample matched belongs to the Pomedaj family, a trojan-downloader that retrieves viruses from a URL to carry out the final attack. Functionally, there seems to be little direct correlation between the two. The *MFMs* used for generating YARA rules are responsible for "shellcode written to system file" (*MFM-1*) and "shellcode execution" (*MFM-2*), where the shellcode refers to the malicious payload used for further attacks. The correspondence between the template and the matched sample is shown in Fig. 13. It reveals that one *MFM* matched, completing the functionality of "copying and executing the malicious payload" (*MFM-1'*). In this case, the payload is a virus for coin mining, and *MFM-1'* performs the functionality of both *MFM-1* and *MFM-2*. Although the ultimate attack objectives differ, the attack processes are similar: both involve placing the malicious payload in a specified path, usually a system file, and then executing it—just as the template *MFM* outlines. This is reflected in the YARA rules through a library/system call sequence of functions such as fopen(), fread(), fwrite(), fclose(), memset(), memcpy(), system(), and sleep(). Increasingly, attackers are adopting this method, as it not only reduces programming workload but also better conceals the malicious functionality. This behavior is commonly captured by the template *MFMs* generated by MalFocus.

Summary: This highlights a more general and flexible approach to malware detection, where malware can be identified by matching one or several *MFM* templates rather than the entire sample. Unlike detection methods based on individual functions or code snippets, *MFM* templates provide a higher-level abstraction of malicious functionality, which helps reduce false positives. Moreover, these templates capture core functionalities shared across different types of malware, enabling the detection of new strains. This answers RQ4.

E. Robustness

In real-world scenarios, attackers often obfuscate their code to evade detection. In recent years, adversarial deep learning models have been increasingly employed in programming, enabling attackers to make targeted modifications to malicious code and bypass deep learning-based detection systems. This underscores the need to evaluate the robustness of MalFocus against obfuscation, as both techniques modify code with the goal of circumventing model detection. Generally, obfuscation can be categorized into static and dynamic methods. For our

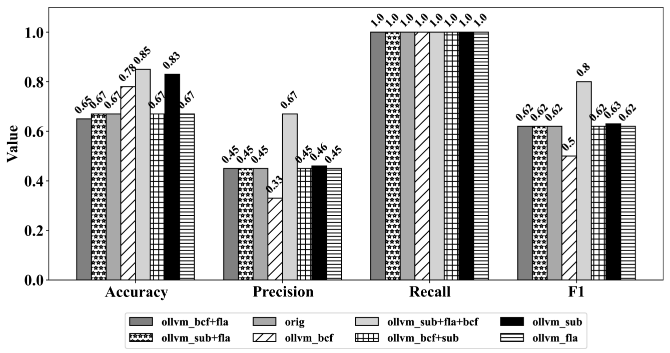


Fig. 14. Comparison of the MFM location effect before and after obfuscation.

assessment, we focus on three commonly used obfuscation strategies from Obfuscator-LLVM (ollvm) [75], excluding more complex dynamic methods and intricate static techniques like encryption, as they fall outside the scope of our research. The selected strategies can be combined in the following pairs: (sub & fla), (sub & bcf), (bcf & fla), and (sub & fla & bcf), resulting in a total of seven combinations.

- *Instruction substitution (ollvm-sub)* complicates simple mathematical operations, particularly focusing on integer operations.
- *Control flow flattening (ollvm-fla)* transforms the program's control flow into a flat structure using multiple switch-case constructs.
- *Bogus control flow (ollvm-bcf)* introduces additional basic blocks into the program's control flow, which serve merely as connections and do not alter the actual execution logic.

We take a malware called "Lightaidra", one variant of the most representative botnet malware family Mirai, which makes DDoS attacks targeted at IoT devices via brute-forcing default or weak passwords, as our evaluation object. A detailed manual analysis allows us to identify sensitive functions related to "daemon creation," "payload transmission," and "remote control," leading to the labeling of the corresponding *MFMs*. In total, we identify 5 *MFMs* out of 18 *FMs*, which are responsible for "daemon creation," "process ID update," "IRC server list creation," "IRC server connection," and "command parsing and execution." We then apply 7 obfuscation strategies to the source code of "Lightaidra" and evaluate the effect of MalFocus on samples before and after obfuscation, as shown in Fig. 14. The results indicate that *recall* remains unchanged across all 7 obfuscated versions, consistently reaching 100%, signifying that *MFMs* are still fully recognized. Although we observe a slight decline in *accuracy* and *F1-Score*, averaging 14% and 20%, respectively, this reduction stems from fluctuations in *precision*, indicating that some *FMs* are misclassified as malicious following obfuscation. As a tool designed to assist manual analysis, the focus should primarily be on *recall*, especially since attackers typically only obfuscate the malicious components in real scenarios. Therefore, we conclude that the robustness of MalFocus is sufficient to withstand these basic obfuscation strategies, thus answering RQ5.

V. Discussion

Threats to validity: In this work, we focus on evaluating ELF-format samples on the Linux platform. While few studies have addressed this area, most research has concentrated on PE-format samples in the Windows environment, largely due to their easier accessibility. This discrepancy can be attributed to the differing popularity of the two platforms; Windows has a broader user base, making it more attractive to attackers. This trend is evident in the landscape of cloud service providers, where the number of malware samples in PE format can be a dozen times of those in ELF. However, as the Linux platform increasingly gains traction in various application fields, such as IoT and embedded devices, due to its open-source nature and portability, our study aims to fill the research gap surrounding ELF-format malware. For a comprehensive evaluation, we collect 20,291 samples from 25 families. Although this number may seem modest compared to those gathered in PE-focused studies, it is considered sufficient given the significant disparity in the volume of malware available in real-world scenarios. Notably, a study conducted by Emanuele et al. [76] examined the characteristics of Linux malware using 10,548 samples, which is half the amount we have collected.

The *precision* and *F1-Score* reported in Section IV-B may not meet expectations, potentially due to the incomplete labeling of *MFMs*. We utilize dynamic analysis to assist our manual labeling, aiming to balance the quantity and quality of labeled data. However, dynamic analysis may not trigger all behaviors and cannot capture static information, which can lead to missing labeled *MFMs*. This is further supported by the findings in Section IV-C, where the *precision* and *F1-Score* based solely on manually labeled *MFMs* are significantly higher than that in Section IV-B. Moreover, since MalFocus operates in a human-machine collaborative manner, the primary goal of the automatic localization is to alleviate the reverse analysis workload and identify as many *MFMs* as possible. This capability has been validated by the substantial reduction in "*manual analysis scope*" and ideal *recall* percentages demonstrated by MalFocus.

As discussed in Section IV-B2, the effectiveness of the mask-based optimization method heavily relies on the accuracy of the malware classification model. To this end, we use 13,527 malware samples for training the classifier, while reserving the remaining 6,764 samples for testing. Although this represents a reduction from the original dataset size, it remains a sufficient volume for effective Linux malware analysis due to the reasons previously outlined.

Limitations Any tool will have flaws, MalFocus is no exception, the drawbacks of it are analyzed as follows.

- One of the key characteristics of MalFocus is its ability to locate malicious functionality within *FMs*. However, embedding *FM* in a rational approach presents challenges, as discussed in Section III-D1. To address this, we propose a compromise method that first identifies key functions and then locates the *MFMs* based on these functions, though it does not allow for direct assessment of the FMs themselves. Additionally, as the number of *MFM* templates increases, clustering templates that share similar functionalities becomes essential. However, this process may be hindered by the difficulties in effectively embedding the *MFMs*.
- The AE-based model is constructed solely from benign samples, making the quantity and quality of these samples critical to the model's effectiveness. These samples should encompass as many normal behaviors as possible; if certain behaviors are not included, corresponding normal components in malware may be misclassified as malicious. Additionally, some benign samples might contain malicious elements, potentially allowing those components to bypass detection. As an inherent characteristic of the AE, it is hard to entirely eliminate, even the optimization method discussed in Section III-D2.
- As previously mentioned, the accuracy of the malware classifier significantly affects the performance of the mask-based ranking algorithm, as interpretability analysis relies on this accuracy. Therefore, a sufficient quantity and variety of malware samples must be collected as training data to ensure classification accuracy, which inevitably consumes time and resources. Additionally, we evaluate the maliciousness of the *FM* by randomly adjusting the features it contains. This approach introduces a potential bias: *FMs* with more functions are more likely to influence judgment and are consequently easier to classify as malicious, which may impact the overall effectiveness of the assessment.

VI. Conclusion

In this work, we present MalFocus, a tool designed to identify the malicious functionality of binary malware at the granularity of *FMs*, addressing the inaccuracies typically associated with function-level localization. MalFocus combines an unsupervised model with an interpretability method based on a binary classifier, eliminating the need for manual malware category labeling, determining the scope of *MFMs*, and ranking them by their level of maliciousness. Through a collaborative human-machine approach, security analysts verify the identified *MFMs*, significantly reducing the scope of manual analysis and revealing the complete malware attack flow. Additionally, the verified *MFMs* can be used as templates to detect variants and new malware families with different functionalities in a more flexible and generalized manner.

Acknowledgment

The work of Weihao Huang was mainly done during his doctoral studies at Institute of Information Engineering, Chinese Academy of Sciences.

References

[1] T. McIntosh et al., "Ransomware reloaded: Re-examining its trend, research and mitigation in the era of data exfiltration," *ACM Comput. Surv.*, vol. 57, no. 1, Oct. 2024, Art. no. 18, doi: 10.1145/3691340.

[2] J. Choi, A. Anwar, H. Alasmary, J. Spaulding, D. Nyang, and A. Mohaisen, "IoT malware ecosystem in the wild: A glimpse into analysis and exposures," in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, New York, NY, USA, 2019, pp. 413–418, doi: 10.1145/3318216.3363379.

[3] AV-TEST, "Malware statistics and trends report | AV-TEST," Mar. 2022. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[4] O. Alrawi et al., "The circle of life: A large-scale study of the IoT malware lifecycle," in *Proc. 30th USENIX Secur. Symp.*, USENIX Association, 2021, pp. 3505–3522. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/alrawi-circle

[5] S. Chen et al., "An empirical assessment of security risks of global Android banking apps," in *Proc. 42nd Int. Conf. Softw. Eng.*, New York, NY, USA, 2020, pp. 1310–1322.

[6] E. Downing, Y. Mirsky, K. Park, and W. Lee, "DeepReflect: Discovering malicious functionality through binary reconstruction," in *Proc. 30th USENIX Secur. Symp.*, USENIX Association, 2021, pp. 3469–3486.

[7] A. R. Kang, Y.-S. Jeong, S. L. Kim, and J. Woo, "Malicious PDF detection model against adversarial attack built from benign PDF containing JavaScript," *Appl. Sci.*, vol. 9, no. 22, p. 4764, 2019.

[8] L. Li et al., "Understanding Android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, Jun. 2017.

[9] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[10] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng, "Image-based malware classification using ensemble of CNN architectures (IMCEC)," *Comput. Secur.*, vol. 92, 2020, Art. no. 101748.

[11] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to Android malware detection and malicious code localization," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1222–1274, 2018.

[12] Q. Wu, P. Sun, X. Hong, X. Zhu, and B. Liu, "An Android malware detection and malicious code location method based on graph neural network," in *Proc. 4th Int. Conf. Mach. Learn. Mach. Intell.*, 2021, pp. 50–56.

[13] Z. Zhang, T. Jiang, S. Li, and Y. Yang, "Automated feature learning for nonlinear process monitoring–An approach using stacked denoising autoencoder and k-nearest neighbor rule," *J. Process Control*, vol. 64, pp. 49–61, 2018.

[14] A. Anwar, H. Alasmary, J. Park, A. Wang, S. Chen, and D. Mohaisen, "Statically dissecting Internet of Things malware: Analysis, characterization, and detection," in *Proc. 22nd Int. Conf. Inf. Commun. Secur.*, Berlin, Heidelberg: Springer-Verlag, 2020, pp. 443–461, doi: 10.1007/978-3-030-61078-4_25.

[15] Z. Pan, J. Sheldon, and P. Mishra, "Hardware-assisted malware detection and localization using explainable machine learning," *IEEE Trans. Comput.*, vol. 71, no. 12, pp. 3308–3321, Dec. 2022.

[16] I. Pro, Nov. 2021. [Online]. Available: https://hex-rays.com/ida-pro/

[17] OllyDbg, Nov. 2021. [Online]. Available: https://www.ollydbg.de/

[18] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based IoT malware detection systems," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1296–1305.

[19] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen, "Subgraph-based adversarial examples against graph-based IoT malware detection systems," in *Proc. 8th Int. Conf. Comput. Data Social Netw.*, Ho Chi Minh City, Vietnam, Springer, 2019, pp. 268–281.

[20] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—A state of the art survey," *ACM Comput. Surv.*, vol. 52, no. 5, Sep. 2019, Art. no. 88, doi: 10.1145/3329786.

[21] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. ACM Conf. Comput. Commun. Secur.*, Alexandria, VA, USA, 2007, pp. 116–127.

[22] W. Enck et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, Vancouver, BC, Canada, USENIX Association, 2010, pp. 393–407.

[23] F. Shen, J. D. Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek, "Android malware detection using complex-flows," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2430–2437.

[24] U. Bayer, A. Moser, C. Krügel, and E. Kirda, "Dynamic analysis of malicious code," *J. Comput. Virol.*, vol. 2, no. 1, pp. 67–77, 2006.

[25] X. Wang and R. Karri, "NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proc. 50th Annu. Des. Automat. Conf.*, Austin, TX, USA, 2013, pp. 79:1–79:7.

[26] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, "Neurlux: Dynamic malware analysis without feature engineering," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, San Juan, PR, USA, 2019, pp. 444–455, doi: 10.1145/3359789.3359835.

[27] A. Karnik, S. Goswami, and R. K. Guha, "Detecting obfuscated viruses using cosine similarity analysis," in *Proc. 1st Asia Int. Conf. Modelling Simul.*, 2007, pp. 165–170.

[28] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "SplitScreen: Enabling efficient, distributed malware detection," *J. Commun. Netw.*, vol. 13, pp. 187–200, 2011.

[29] J.-W. Jang, J. Woo, J. Yun, and H. K. Kim, "Mal-netminer: Malware classification based on social network analysis of call graph," in *Proc. 23rd Int. Conf. World Wide Web*, New York, NY, USA, 2014, pp. 731–734, doi: 10.1145/2567948.2579364.

[30] Y. Park, D. S. Reeves, and M. Stamp, "Deriving common malware behavior through graph clustering," *Comput. Secur.*, vol. 39, pp. 419–430, 2013.

[31] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 2, pp. 289–302, Feb. 2016.

[32] M. Chandramohan, H. B. K. Tan, L. C. Briand, L. K. Shar, and B. M. Padmanabhuni, "A scalable approach for malware detection through bounded feature space behavior modeling," in *Proc. 28th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2013, pp. 312–322.

[33] Ã. Aslan, M. Ozkan-Okay, and D. Gupta, "Intelligent behavior-based malware detection system on cloud computing environment," *IEEE Access*, vol. 9, pp. 83252–83271, 2021.

[34] G. Meng, R. Feng, G. Bai, K. Chen, and Y. Liu, "DroidEcho: An in-depth dissection of malicious behaviors in android applications," *Cybersecurity*, vol. 1, no. 1, Jun. 2018, Art. no. 4, doi: 10.1186/s42400-018-0006-7.

[35] M. M. Masud, L. Khan, and B. M. Thuraisingham, "A hybrid model to detect malicious executables," in *Proc. IEEE Int. Conf. Commun.*, 2007, pp. 1443–1448.

[36] M. G. Schultz, E. Eskin, E. Zadok, and S. Stolfo, "Data mining methods for detection of new malicious executables," in *Proc. IEEE Symp. Secur. Privacy*, 2001, pp. 38–49.

[37] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf.*, 2004, pp. 41–42.

[38] R. Moskovitch et al., "Unknown malcode detection using OPCODE representation," in *Proc. Eur. Conf. Intell. Secur. Informat.*, 2008, pp. 204–215.

[39] K. He and D. S. Kim, "Malware detection with malware images using deep learning techniques," in *Proc. 18th IEEE Int. Conf. Trust Secur. Privacy Comput. Commun./13th IEEE Int. Conf. Big Data Sci. Eng.*, 2019, pp. 95–102.

[40] J. Yan, Y. Qi, and Q. Rao, "Detecting malware with an ensemble method based on deep neural network," *Secur. Commun. Netw.*, vol. 2018, 2018, Art. no. 7247095.

[41] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf.*, 2016, pp. 577–582.

[42] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A performance-sensitive malware detection system using deep learning on mobile devices," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1563–1578, 2020.

[43] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for Android malware detection using various features," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 3, pp. 773–788, Mar. 2019.

[44] K. Xu, Y. Li, R. H. Deng, and K. Chen, "DeepRefiner: Multi-layer android malware detection system applying deep neural networks," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2018, pp. 473–487.

[45] D. Davis, J. Burry, and M. Burry, "Untangling parametric schemata: Enhancing collaboration through modular programming," in *Proc. 14th Int. Conf. Comput. Aided Architectural Des.*, Univ. Liege, Liege, Jan. 2011, pp. 55–68.

[46] D. Foo, J. Yeo, H. Xiao, and A. Sharma, "The dynamics of software composition analysis," 2019, *arXiv: 1909.00973*.

[47] T. Chen, Q. Mao, Y. Yang, M. Lv, and J. Zhu, "TinyDroid: A lightweight and efficient model for android malware detection and classification," *Mobile Inf. Syst.*, vol. 2018, 2018, Art. no. 4157156.

[48] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," 2020, *arXiv: 2002.03594*. [Online]. Available: https://arxiv.org/abs/2002.03594

[49] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of Android malware for effective malware comprehension, detection, and classification," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2016, pp. 306–317.

[50] S. Àlvarez et al., "Radare2: Unix-like reverse engineering framework," 2014. [Online]. Available: https://github.com/radareorg/radare2

[51] UPX, "UPX - The ultimate packer for executables," Mar. 2022. [Online]. Available: https://github.com/upx/upx

[52] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 6, 2015, Art. no. 659101, doi: 10.1155/2015/659101.

[53] F. O. Catak, A. F. Yazi, O. Elezaj, and J. Ahmed, "Deep learning based sequential model for malware analysis using windows exe API calls," *PeerJ Comput. Sci.*, vol. 6, 2020, Art. no. e285.

[54] X. Yang, D. Yang, and Y. Li, "A hybrid attention network for malware detection based on multi-feature aligned and fusion," *Electronics*, vol. 12, no. 3, 2023. [Online]. Available: https://www.mdpi.com/2079-9292/12/3/713

[55] M. E. Ahmed, S. Nepal, and H. Kim, "MEDUSA: Malware detection using statistical analysis of system's behavior," in *Proc. IEEE 4th Int. Conf. Collaboration Internet Comput.*, 2018, pp. 272–278.

[56] S. Rezaei, A. Afraz, F. Rezaei, and M. R. Shamani, "Malware detection using opcodes statistical features," in *Proc. 8th Int. Symp. Telecommun.*, 2016, pp. 151–155.

[57] M. Grohe, "word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data," in *Proc. 39th ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, 2020, pp. 1–16.

[58] Nov. 2021. [Online]. Available: https://virusshare.com/

[59] L. Meng and N. Masuda, "Analysis of node2vec random walks on networks," in *Proc. Roy. Soc. A*, vol. 476, no. 2243, 2020, Art. no. 20200447.

[60] M. S. Kiraz, Z. A. Genç, and E. Öztürk, "Detecting large integer arithmetic for defense against crypto ransomware," 2017. [Online]. Available: https://eprint.iacr.org/2017/558

[61] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for Android apps," *SoftwareX*, vol. 11, 2020, Art. no. 100403.

[62] A. M. Martinez and A. C. Kak, "PCA versus LDA," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 2, pp. 228–233, Feb. 2001.

[63] T. Kincaid, "Cumulative distribution function (CDF) analysis," Nov. 2012.

[64] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. Berlin, Germany: Springer Science & Business Media, 2013.

[65] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels, "Scalable community detection with the louvain algorithm," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 28–37.

[66] S. E. Garza and S. E. Schaeffer, "Community detection with the label propagation algorithm: A survey," *Physica A: Statist. Appl.*, vol. 534, 2019, Art. no. 122058.

[67] M. E. Newman, "Fast algorithm for detecting community structure in networks," *Phys. Rev. E*, vol. 69, no. 6, 2004, Art. no. 066133.

[68] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, 2004, Art. no. 026113.

[69] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *Inf. Secur. Tech. Rep.*, vol. 14, no. 1, pp. 16–29, 2009.

[70] M. Brengel and C. Rossow, "YARIX: Scalable YARA-based malware intelligence," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 3541–3558.

[71] VirusTotal, Nov. 2021. [Online]. Available: https://virustotal.com/

[72] M. ATT &CK, "UPX - The ultimate packer for executables," Oct. 2020. [Online]. Available: https://attack.mitre.org/versions/v7/matrices/enterprise/linux/

[73] capa, 2020. [Online]. Available: https://github.com/fireeye/capa

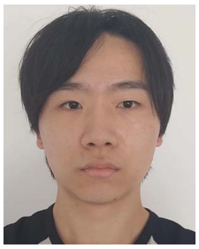
[74] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 4768–4777.

[75] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM–Software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, 2015, pp. 3–9.

[76] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 161–175.

Weihao Huang received the PhD degree in cyberspace security from the Institute of Information Engineering, Chinese Academy of Science (IIE, CAS). He is a postdoctoral fellow with the School of Cyber Science and Technology, Sun Yat-sen University. He is widely interested in software and system security, intelligent system security, especially in program analysis, clone detection, malware analysis, vulnerability mining and software security of intelligent systems.

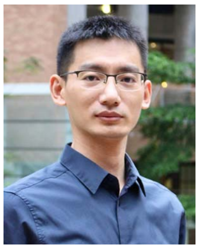
Chaoyang Lin received the master's degree in cyberspace security from the Institute of Information Engineering, Chinese Academy of Science (IIE, CAS). He is a security researcher with Safety & CyberSecurity Department, NIO. He is widely interested in IoT security and Android security, especially in vulnerability detecting and exploiting.



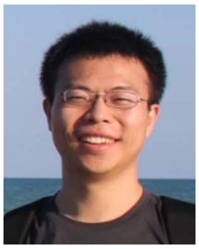
Lu Xiang received the BE degree from Southeast University, China, in 2021. He is currently working toward the ME degree with the Institute of Information Engineering, Chinese Academy of Sciences, China. His main research interests include software security and AI for security.

Zhiyu Zhang received the BE degree in eletronic information and communications from the Huazhong University of Science and Technology, China, in 2021. He is currently working toward the PhD degree with the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include software security and system security. He was the recipient of "National scholarship for undergraduate students, China" in 2021.

Guozhu Meng received the BE and ME degrees from Tianjin University, China, in 2009 and 2012, respectively, and the PhD degree from Nanyang Technological University, Singapore, in 2017. He is currently an associate professor with the Institute of Information Engineering, Chinese Academy of Sciences. Before that, he was a research fellow with Nanyang Technological University and a visiting research fellow with the University of Luxembourg. His research interests include mobile security, vulnerability detection, Big Data analysis, and program analysis.



Lei Xue received the PhD degree in computer science from The Hong Kong Polytechnic University (PolyU). He is an associate professor with the School of Cyber Science and Technology, Sun Yat-sen University. He is widely interested in designing and implementing efficient and practical security systems, with a particular focus on applying hybrid program/application analysis techniques, network traffic analysis methodology, and machine learning based alogrithms to addressing challenging security and privacy issues in mobile, automotive, and network systems. Currently, his research topics mainly foucus on mobile and IoT system security, program analysis, and automotive security.



Kai Chen received the PhD degree from the University of Chinese Academy of Science, in 2010, then he joined the Chinese Academy of Science, in January 2010. He became the associate professor, in September 2012 and became the full professor, in October 2015. His research interests include software analysis and testing, smartphones and privacy.

Lei Meng is currently an expert in security algorithms of Alibaba Cloud. His research areas include host security, network security, data security, etc.

Zongming Zhang received the bachelor of science degree from the Hebei University of Economics and Business, China, in 2005. His research interests include security, cloud computing, and operating systems, especially in the areas of binary analysis, virtualization, and operating system security.