MalSensor: Fast and Robust Windows Malware Classification

HAOJUN ZHAO, Huazhong University of Science and Technology, China
YUEMING WU*, Nanyang Technological University, Singapore
DEQING ZOU, Huazhong University of Science and Technology, China
YANG LIU, Nanyang Technological University, Singapore
HAI JIN, Huazhong University of Science and Technology, China

Driven by the substantial profits, the evolution of Portable Executable (PE) malware has posed persistent threats. PE malware classification has been an important research field, and numerous classification methods have been proposed. With the development of machine learning, learning-based static classification methods achieve excellent performance. However, most existing methods cannot meet the requirements of industrial applications due to the limited resource consumption and concept drift. In this paper, we propose a fast, high-accuracy, and robust FCG-based PE malware classification method. We first extract precise function call relationships through code and data cross-referencing analysis. Then we normalize function names to construct a concise and accurate function call graph. Furthermore, we perform topological analysis of the function call graph using social network analysis techniques, thereby enhancing the program function call features. Finally, we use a series of machine learning algorithms for classification. We implement a prototype system named *MalSensor* and compare it with nine state-of-the-art static PE malware classification methods. The experimental results show that *MalSensor* is capable of classifying a malicious file in 0.7 seconds on average with up to 98.35% accuracy, which represents a significant advantage over existing methods.

CCS Concepts: • **Security and privacy → Malware and its mitigation**.

Additional Key Words and Phrases: Malware Semantic Analysis, Centrality, Disassembly

1 INTRODUCTION

Malware is an umbrella term that describes any software, firmware, or code intended to perform a malicious unauthorized process that will have an adverse impact on the confidentiality, integrity, or availability of a system [15]. As the most widely used operating system in the world, Windows has a huge user base. However, due to the wide range of software sources and the open-use environment of Windows, its users security has long been threatened by malware. Therefore, the analysis and classification of PE malware have always been a significant research field.

*Corresponding author

Authors' addresses: H. Zhao and D. Zou are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China; emails: haojunzhao@hust.edu.cn, deqingzou@hust.edu.cn; Y. Wu and Y. Liu are with Nanyang Technological University, Singapore; e-mails: wuyueming21@gmail.com, yangliu@ntu.edu.sg; H. Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China; e-mail: hjin@hust.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s).
ACM 1557-7392/2024/8-ART0
https://doi.org/10.1145/3688833

Just Accepted

The classification methods of PE malware are generally divided into static analysis-based and dynamic analysis-based [19]. Compared with dynamic analysis, static analysis has wider practical applications in large sample sizes and heavy task processing. In recent years, with the development of machine learning, learning-based malware classification methods [19, 53] are achieving increasingly excellent performance. Compared with the traditional rule-based or heuristic-based methods, the machine learning-based malware classification methods can undoubtedly better deal with new malware or unsigned variants due to its inherent generalization.

Learning-based classification methods can be divided into three categories based on how features are used: image-based, binary-based, and disassembly-based. Image-based malware classification work usually converts binary file streams into fixed-width grayscale image and then combines image classification algorithms for malware classification. For example, methods in [6, 10, 45, 55] represent the binary content of the PE file as a gray scale image and apply image classification techniques to analyze it. However, image-based classification is generally limited in accuracy and scalability when faced with large-scale malware classification tasks in complex real-world environments [9]. Moreover, because of the continuous evolution of malware, image matching algorithms cannot make good use of existing malware information to process newer malware well [39]. With the development of natural language processing (NLP), some research [7, 48, 50] convert PE files into binary sequences as text and adopt NLP sequence models to solve the family classification problem. Nevertheless, these methods cannot reflect the semantic information of the program as it does with natural language text, because the instructions execution of the program is not a sequential model. Research [7, 12, 36, 46] disassemble malware raw files to extract opcodes and some work [6, 49] utilizes Application Program Interface (API) call frequency for malware analysis. However, they all lack the deep mining of program semantic information. In order to understand program semantic information more comprehensively, some researchers find it is an efficient way to combine program execution logic with graphs. For example, Kong et al. [33] abstract malware into an attribute function call graph (AFCG), and then learn malware distance metrics to distinguish different AFCGs. Similarly, Yan et al. [58] extract an attribute control flow graph (A-CFG) of the program by disassembling and applying graph analysis to complete malware classification. However, CFG matching is time-consuming and may not generalize well, potentially affecting classification accuracy and limiting large-scale applicability.

Ma et al. [39] conduct the first large-scale and systematic empirical study on learning-based PE malware analysis methods. Through numerous experiments and discussions with security companies, they find that existing learning-based methods are easily hampered in industrial applications. The main limitations restricting the industry application of existing malware classification research are the precision, the predicting time, and resource consumption. The first two factors decide the user experience, and the resource consumption decides what kind of devices can the methods be deployed on. Some devices like gateway must deploy lightweight malware detection due to their inherent resource constraints and traffic speed requirements. As a concrete example, some products need to meet the requirements of running without GPU, prediction time less than 0.1s, and accuracy higher than 93%. However, in their study, almost all methods failed to meet these requirements simultaneously. They reveal that future researches tend to explore robust and lightweight models with high prediction accuracy and consider malicious semantic features to better deal with rapidly evolving malware families and unknown malware. Unfortunately, most of the existing researches fail to meet these needs.

To meet the requirements of practical applications, we aim to propose a lightweight, accurate, and robust classification method. Given the importance of semantic analysis and the effectiveness of graph structures, we utilize function call graphs for malware analysis. Therefore, we need to address two main challenges:

- *Challenge 1: How to apply precise and fast static analysis to retain complete program semantics for accurate malware analysis?*
- *Challenge 2: How to design succinct yet effective graph analysis for scalable malware analysis?*

Just Accepted

To address the first challenge, we focus on capturing contextual and semantic features of malware for classification. We know that the assembly instruction sets of different CPU architectures are different, and the disassembly decorated names generated by the same function using different function calling conventions differ as well. These factors affect the accuracy of classification. On the one hand, when classifying large-scale malware, sample sources are highly complex. The same upper-level function of the same family samples running on different systems may have different function decorated names after compiling. On the other hand, the same malicious sample may also run in systems with different architectures and has different function decorated names of the same upper-level function. For example, the new botnet *Chaos* attacks Intel, ARM, and other embedded system architectures such as MIPS simultaneously. However, for all we know, no research work has paid attention to these issues. To mitigate the issue, we implement a custom static analysis module including a decorated name peeler to eliminate these differences. In this way, we can extract more precise function call graphs and perform better in complex practical application environments.

To tackle the second challenge, we treat a function call graph as a social network and perform a centrality analysis to retain the graph details. Some work [13, 27] has attempted to apply social network analysis to PE malware analysis, but they only try to analogize the macroscopic attributes of malicious programs to social networks, using social network attribute analysis methods to solve the problem. However, we believe that reverse-mapping the social network to the fine-grained function call graph of the program is more effective. Each function in the program is analogous to the user node in a social network, and the call relationships between functions are analogous to the user relationship in this social network. Therefore, the function call graph can be viewed as a social network graph structure. Centrality concepts are first proposed in social network analysis, and their original purpose is to quantify the importance of a person in a network. Empirical studies [17, 41] have demonstrated that centrality has the ability to reflect the structural characteristics of the network. Therefore, we can leverage weighted centrality analysis to maintain the graph properties to achieve lightweight and effective malware classification.

We implement a prototype system, namely *MalSensor*, and evaluate it in accuracy, efficiency, and robustness compared with nine state-of-the-art static analysis-based malware classification works referenced in the research [39] under the same conditions. The experimental results indicate that *MalSensor* is more accurate than other methods, especially 3.7%-11% higher than similar methods. As for scalability, *MalSensor* consumes less running time and it can be several times to thousands of times faster than other methods at different analysis stages. As for robustness, our concept drift experiments verify that *MalSensor* is more robust than other methods in the evolution of Windows applications.

In summary, the main contributions of our work are as follows:

- We propose an effective method for program semantic analysis based on function call graph. This method generates concise and accurate function call graph through data and code cross-referencing, as well as function name normalization. Additionally, it enhances the features of function call graph using weighted social network analysis.
- We design and implement a lightweight, accurate, and well-compatibility PE malware classification system called *MalSensor*[1], which can perform well on large-scale PE malware.
- We conduct extensive evaluations in terms of malware classification effectiveness and scalability. The experimental results show that *MalSensor* achieves a 98.35% F1 score with the lowest resource overhead, which is superior to the other nine state-of-the-art methods.

Paper organization. The remainder of the paper is organized as follows. Section II presents the preliminary study on function decorated name and social network centrality. Section III introduces our system. Section IV

[1]https://github.com/johorun/MalSensor

Just Accepted

reports the experimental results. Section V discusses future work. Section VI describes the related work. Section VII concludes the present paper.

2 PRELIMINARY STUDY

This section presents the threat model and provides pertinent information on function decorated name and social network centrality that enable the techniques that we present in this work.

2.1 Threat Model

In industrial scenarios, large-scale analysis of malware from complex sources is often required. Malicious samples often have different families, timestamps, programming languages, and operating architectures (malware that exploits a cross-system vulnerability may often run on different architectures, e.g., some worms exploiting HTTP protocol stack). We aim to perform fast and efficient analysis on large-scale malware programs from complex sources.

In order to illustrate our work more clearly, we clarify two concepts as follows: (1) In this paper, the input malicious program is pure, that is, it does not use adversarial attack methods such as packing, adding fake functions, hiding function imports, etc. All the work based on static analysis have a common inherent limitation, that is, it is difficult to deal with various means of adversarial attack, resulting in misjudgments. We will detail these limitations and mitigation methods in the Section 5 (such as attaching automatic unpacking model before the preprocessing of our work). (2) We believe that the application requirements of static analysis-based malware classification in the real world are broad, especially in terms of performance and resource consumption. In the real world, there are many malware analysis application scenarios with tight computing resources. For example, gateway devices are usually with limited computing resources. In order to ensure the use of intranet users, it is necessary to deploy as lightweight and accurate security analysis modules as possible.

2.2 Function Call Decorated Name

We know that the same upper-level functions called by programs running on different CPU architectures, written in different high-level languages, and using different function calling conventions may have different disassembly function names. We find that this phenomenon changes the topology of the function call graph and affects the function call graph comparison between different malware when conducting large-scale experiments. Below we give the related concept definitions and intuitive explanations.

The function calling convention determines the push and pop rules for parameters when a function is called and specifies the generation rules for the function decorated name. The function decorated name rules determine which decorated name the compiler will use to uniquely rename functions when generating the executable file. It ensures that the function names are uniquely identifiable when linking programs. However, the same top-level function uses different function calling conventions and name mangling rules, and may get different actual function names after program linking. The function name decorating rules of different calling conventions for different programming languages(e.g., C and C++) are shown in Fig. 1.

Due to the complex environment of Windows, PE programs are usually generated with different programming languages and compiler strategies. Therefore, the same function in different PE malicious files or in the same malicious file running in different CPU architectures may have different decorated names. This results in different decorated names for the same upper-level function being treated as separate functions during function call analysis, which causes redundancy. In addition, this also happens with overloaded functions, the same functionality functions in different programming languages, etc.

This redundancy will not only affect the comparative analysis between function call graphs, but also cause an explosion in the number of function call graph nodes in large-scale malware classification tasks and decrease

Just Accepted

Calling Convention \ Program language	C	C++
__cdecl	Leading underscore (_)	Leading underscore (?) and a trailing at sign (@@YA) followed by parameter list in symbolic abbreviated form
__fastcall	Leading and trailing at signs (@) followed by a decimal number representing the number of bytes in the parameter list	Leading underscore (?) and a trailing at sign (@@YI) followed by parameter list in symbolic abbreviated form
__stdcall	Leading underscore (_) and a trailing at sign (@) followed by the number of bytes in the parameter list in decimal	Leading underscore (?) and a trailing at sign (@@YG) followed by parameter list in symbolic abbreviated form
thiscall		Leading underscore (?) and insert sign(@) between the function name and parameter list to lead the class name. The start identifier of the parameter list is different according to the properties of the class: Public(@@QAE), Protected(@@IAE), Private(@@AAE). If const modification is used, they become(@@QBE), (@@IBE), (@@ABE). Use "AAV1" if the argument type is a reference to a class instance, and "ABV1" for references to a const type.

Fig. 1. Function name decorating rules of different calling conventions.

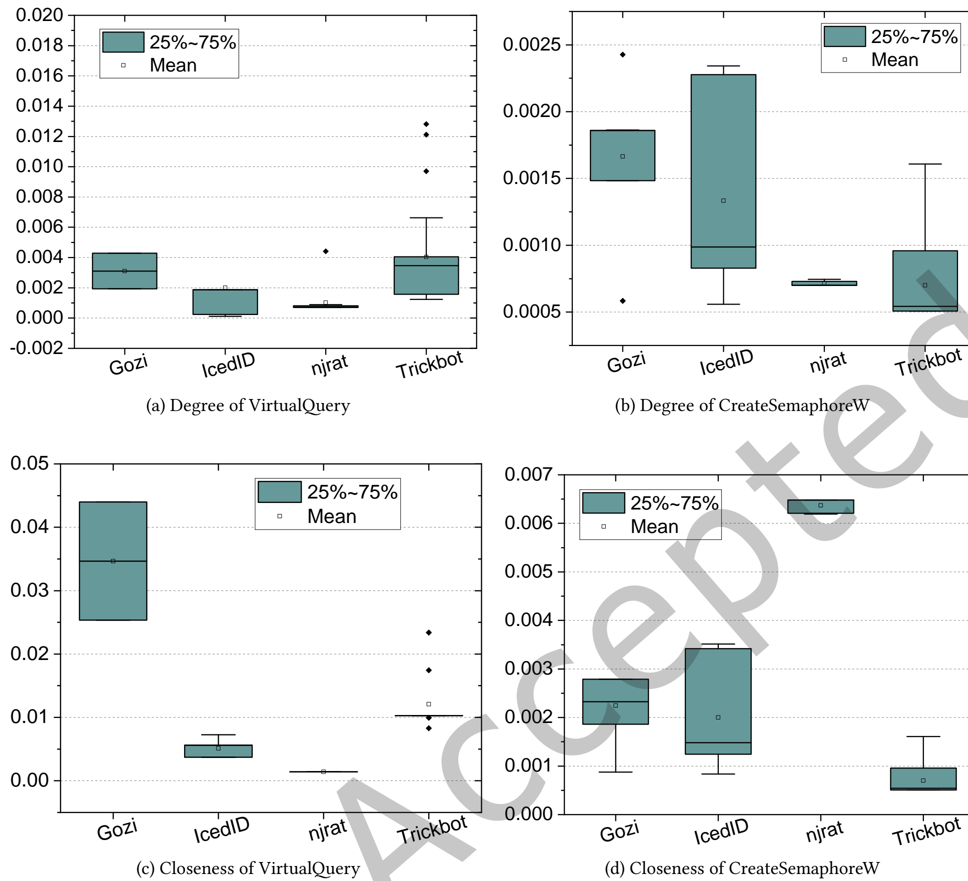
Just Accepted

classification efficiency. However, to our knowledge, no related work mentions this. To validate our hypothesis, we conduct a preliminary experiment. We randomly download 20,000 program samples in 2018-2021 with different CPU architectures and programming languages from VirusTotal[2] and Scoop[3] and perform a disassembly analysis. Through analysis and comparison, we find that if the function decorated name is converted to the original upper function name after disassembly analysis, the number of external import functions will be greatly reduced from 31,571 to 11,049. This affects the construction of the function call graph and significantly reduces the dimensionality of the feature vectors generated during the classification stage, which effectively improves the accuracy and speed of the classification process.

2.3 Centrality

A social network consists of a group of user nodes and the social relations between them. The nodes of the social network graph represent user nodes, and the edges between the nodes represent the social relationships between users. Social network centrality, as the most effective measure to show social network attributes, is widely used in various fields, such as co-authorship network [37], transportation network [21], criminal network [14] and achieves good effects. However, in the field of PE malware classification, no one has proved that the combination

[2]https://www.virustotal.com
[3]https://github.com/lukesampson/scoop



(a) Degree of VirtualQuery

(b) Degree of CreateSemaphoreW

(c) Closeness of VirtualQuery

(d) Closeness of CreateSemaphoreW

Fig. 2. The distribution of degree centrality and closeness centrality of two key API calls (i.e., VirtualQuery and CreateSemaphoreW) in four families (i.e., Gozi, IcedID, njrat, and Trickbot)

Just Accepted

of social network and function call graph is effective for malware classification. In the following, we give the definition of related concepts and the method basis.

Centrality measures the importance of each node in a social network under a certain standard and reveals the deep structural logic of the social network. According to different standards, it can be divided into degree centrality, closeness centrality, betweenness centrality, eigenvector centrality, katz centrality, harmonic centrality, etc. We believe that a program can be regarded as a complete social network, the functions in the program are the user nodes of the social network, and the calling relation between the functions is the social relationship between users. Therefore, we believe that the social network centrality theory can be used to analyze a program. By analyzing the centrality of each function node in the function call graph of a program, we can highlight the importance of functions at different topological locations in the global graph structure, thereby deepening the semantic features of the program. This is very useful for extracting key information from large function call graphs. In order to confirm our idea, we conduct experiments to compare the centrality distributions of the same function in different malware families. We randomly select 3,000 malicious samples from 6 different families from

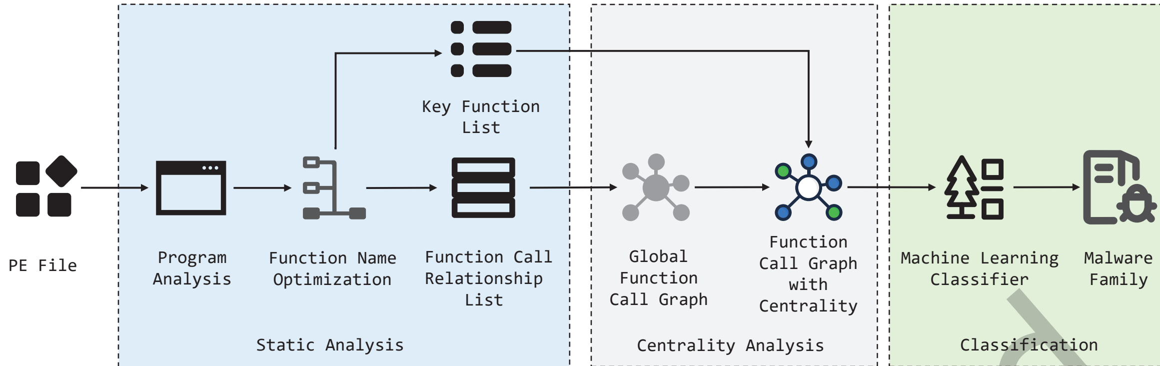


Fig. 3. System architecture of MalSensor

the dataset MalwareBazaar (shown in Section 4) and perform function call graph extraction on them. Then, we perform the function call centrality analysis on some key external import functions to obtain the function call centrality distribution of different families. The experimental results are shown in Fig. 2. Due to page limitations, we only show the value distributions of degree and closeness centrality (as defined in Section 3.3) of two key APIs (VirtualQuery and CreateSemaphoreW) in four families. In the figures, the abscissa represents four families, and the ordinate is the API centrality value corresponding to each sample. We can see that after centrality processing, the API centrality value distributions of different malware families are quite different, which shows that centrality processing can effectively describe the importance of a function in different malware families.

3 SYSTEM ARCHITECTURE

3.1 System Overview

MalSensor operates in three phases: static analysis, centrality analysis, and classification. As shown in Fig. 3, *MalSensor* first performs a quick scan and disassembly analysis on the target program. Using the function cross-reference information and external import function information, for each program, *MalSensor* will get a function call relationship list and a list of key functions (defined in Section 3.2). Then, the function name optimization module renames all function names in the two lists and outputs them for centrality analysis. In the centrality analysis stage, *MalSensor* converts all function call relationships in the program into a graph for storage and uses a series of graph-based centrality algorithms to calculate the node centrality of the functions in the key function list. After this, in the classification stage, all centrality values of the key functions are embedded in the feature vector for representing a malware file, and finally, a variety of classification algorithms are used to classify the malware family of this malware file. To better illustrate the detailed procedures involved in *MalSensor*, we give a simple example in Fig. 4.

3.2 Static Analysis

In this paper, we aim to propose a malware classification system based on semantic information which requires high efficiency in program analysis. To this end, the static analysis component of *MalSensor* is designed as two parts of precise program analysis and function name optimization to capture malware features as well as possible.

Precise Program Analysis. We use the disassembly tool IDA Pro [2] for secondary development to complete the program analysis module of *MalSensor*. Most existing research on static analysis-based malware classification

Just Accepted

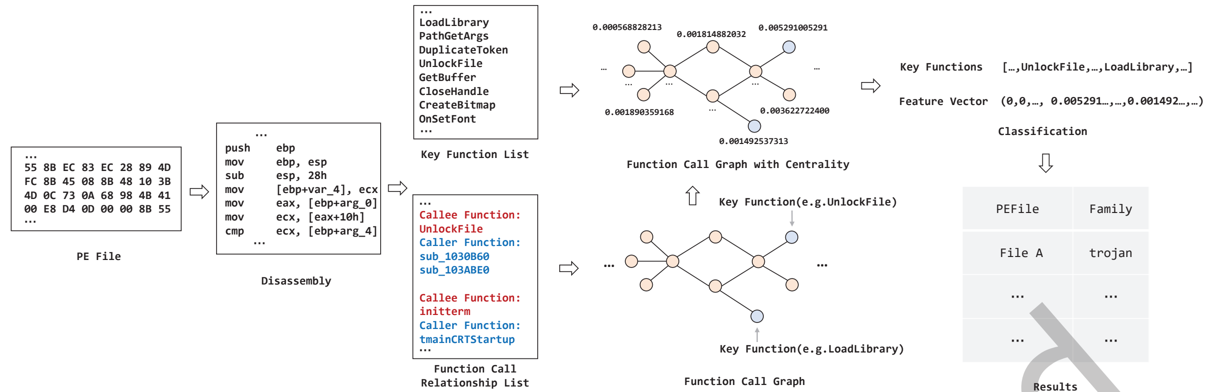


Fig. 4. A simple example to describe the different phases in MalSensor

```
.text:00401138 ; =============== S U B R O U T I N E =======================================
.text:00401138 start:                                  ; CODE XREF: .text:00407220↓j
.text:00401138                                         ; DATA XREF: HEADER:004000A8↑o
.text:00401138                 push    ebp
      ...                      ...     ...
.text:00401158                 lea     eax, TlsGetValue
.text:0040115E                 call    dword ptr [eax]
      ...                      ...     ...
.idata:00419028 ; LPVOID __stdcall TlsGetValue(DWORD dwTlsIndex)
.idata:00419028                extrn TlsGetValue:dword ; DATA XREF: .text:00401158↑o
.idata:00419028                                        ; .text:00401193↑o ...
```

Fig. 5. Function call through data cross-references indirectly.

often focuses on strings [8], bytes [34, 48, 50], opcodes [7, 12, 36, 46], and API call frequency [6, 49], etc., but lack the mining of program semantic information. We believe that the semantic features of malware are the key to distinguishing their families. Therefore, we disassemble the program and perform cross-reference analysis on all functions to obtain a function call network for classification. This function call network lays the foundation of efficient and fast centrality analysis for the key functions.

In disassembly analysis, calls between code basic blocks are embodied in the form of address jumps, and we can analyze these jumps using the concept of cross-reference. According to the purpose of executing jumps, cross-references can be divided into code cross-references and data cross-references. A code cross-reference is used to indicate that an instruction transfers control to another instruction, including three types named *ordinary* reference, *call* reference, and *jump* reference. Data cross-references are used to track the data accessing within a program, and the three most commonly encountered types of data cross-references are *read* cross-reference, *write* cross-reference, and *offset* cross-reference.

Most function call relations in a program can be obtained by code cross-reference, but some function calls invoked indirectly through data cross-reference are always overlooked. A typical situation is that the address of a function is not directly used as the operand of the call instruction, but indirectly obtained by the call instruction through a register or other means. Although the disassembly analysis will determine that this is an offset data cross-reference, the function call process and control flow transition indeed occur. As shown in Fig. 5, the function *start* calls the API *TlsGetValue* by storing the address of the *TlsGetValue* in eax register and using the *call* instruction to jump to the address pointed to by eax. In this case, since the operand of the call instruction is the

Just Accepted

```
.text:00404190 ; =============== S U B R O U T I N E =======================================
.text:00404190 ; int __stdcall sub_404190(LPCSTR lpString, int)
.text:00404190 sub_404190      proc near               ; CODE XREF: sub_4041E0+41↓p
      ...                      ...
.text:004041B5                 push    eax             ; Size
.text:004041B6                 call    malloc
      ...                      ...
.text:00465C66 ; =============== S U B R O U T I N E =======================================
.text:00465C66 ; void *__cdecl malloc(size_t Size)
.text:00465C66 malloc          proc near               ; CODE XREF: sub_404190+26↑p
.text:00465C66                                         ; sub_405690+39↑p ...
.text:00465C66
.text:00465C66 Size            = dword ptr  4
.text:00465C66
.text:00465C66                 jmp     ds:__imp_malloc
.text:00465C66 malloc          endp
      ...                      ...     ...
.idata:0046A150 ; void *__cdecl malloc(size_t Size)
.idata:0046A150                extrn __imp_malloc:dword
.idata:0046A150                                        ; DATA XREF: malloc↑r
```

Fig. 6. Function call through code and data cross-references.

value in the register, the disassembly analysis will judge it as a dynamic call with an uncertain address, so that the control flow direction cannot be directly analyzed through code cross-references. In reality, this is a static call, and the actual function address can be obtained through data cross-reference, so as to determine the calling relationship. This is because this indirect calling must first obtain the actual destination function address through data reference, and then indirectly input it to the call instruction by register or other ways. In another case, the call instruction often makes a cross-section invoke, which is often done through code and data cross-references. Fig. 6 shows a function call that jumps a long address between sections through data cross-references. In fact, this is a typical case produced by link incrementally. Microsoft documentation describes link incrementally as an option enabled by default in MSVC linker, and it may add jump thunks to handle the relocation of functions to new addresses. When compiling an incrementally linked program, the compiler creates a jump stub for all functions. All the invoke procedures will jump to the stub and then get the real function address indirectly. In this way, by grouping the jump stubs together, only one memory page needs to be rewritten when the functions are relocated. Jump stubs imply that one function calling requires two cross-references and a cross-segment invocation. However, relying solely on code cross-references can only obtain the stub name and not the decorated name of the external static function. This can lead to redundancy when analyzing samples in bulk, as the external imported function names cannot be unified across the samples. Considering the above situation, *MalSensor* uses data cross-references to assist in extracting function call relationships. Specifically, MalSensor begins by scanning the .text segment and obtains preliminary function call information through code cross-reference. Then, it conducts data cross-reference analysis on all function addresses within the valid addresses of the *.text* segment and *.idata* segment in program. This data cross-reference analysis helps to retrieve the source addresses that reference the function addresses. The caller functions are determined based on the functions to which the source addresses belong, thus establishing the indirect function call relationships. Since our data cross-reference analysis focuses solely on function addresses, it avoids false positives by not mistakenly identifying normal data usage as function calls. *MalSensor* captures the easily overlooked call behaviors and draws the precise and meticulous global function call graph by combining code cross-reference and data cross-reference.

External imported functions of a program can reflect its critical behaviors and reveal behavioral differences among samples during extensive sample analysis. Therefore, understanding how a program utilizes external imported functions is considered highly important. *MalSensor* extracts the external functions imported by the

Just Accepted

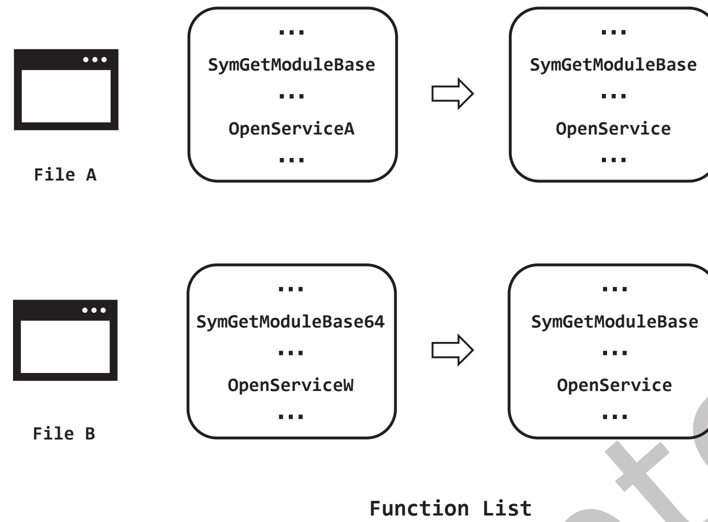


Fig. 7. Changes in key function list by function name optimization.

target program and categorizes them by function library. Since not all external import functions are actually invoked by the program, combining the global function call graph, we keep the focus on those external functions that do be invoked (we call them 'key functions' of the program) and analyze their centralities later.

Function Name Optimization. Apart from the original program analysis module, *MalSensor* has a built-in peeler for function decorated names. As we know, depending on the compilation strategy and programming language, different function calling conventions may be adopted when a function call occurs. Since different function calling conventions use different name decorating rules when programs are linked, the same top-level function may get different decorated names in different PE files. Therefore, even if two programs use the same external function, they probably get different analysis results in disassembly. This may have a negative impact on the function feature analysis between samples. Accordingly, in order to streamline the function call graph and mitigate the curse of dimensionality when generating the feature vector from key functions, *MalSensor* peels the function decorated name for standardization.

Currently, there are mainly five calling conventions: _cdecl, _fastcall, _stdcall, _thiscall, and _naked call. Different function calling conventions adopt different naming formats and prefix modifiers (such as '_', '?') between different programming languages, and use '@' or '@@' to define the argument lists and class or library name of a function. *MalSensor* analyzes the function name decorating rules of the five calling conventions and developed corresponding analysis methods based on these rules, which provide a bridge for the unified analysis of function calling relations in programs from different platforms and languages.

Furthermore, we believe that functions with extremely similar names are likely to be functionally consistent, for example, the windows API *GetMessageA* and *GetMessageW*, where A and W differ only in whether the processing string is in ANSI or UTF-16 format. Function overloading is also a typical case. Due to different function parameter lists, the decorated names of the same overloaded functions are different. To uniformly rename similar functions and reduce the redundant nodes of the program function call network, *MalSensor* matches each function to all other functions by the maximum substring and renames the function to the largest substring whose change

Just Accepted

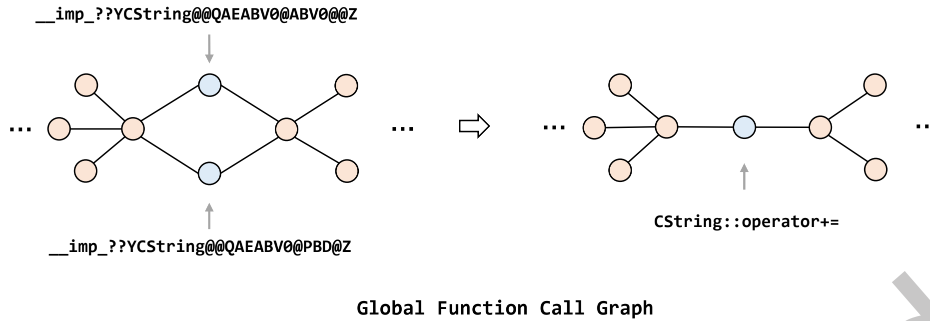


Fig. 8. Changes in global function call graph by function name optimization.

character is up to n (n=2, details are in Section 4.3). Specifically, *MalSensor* compares the target function name with other functions in the sensitive function list, starting from the beginning of the name and proceeding character by character until the comparison is completed. If the comparison is completed and there is a completely identical function name in the sensitive function list, and the length of this function name is less than or equal to the number of characters in the target function name plus 2, then this function name is merged into the target function name. This allows *Malsensor* to ignore the name differences caused by individual decorated characters and aggregate functions with the same function together (as shown in Fig. 7 and Fig. 8).

3.3 Centrality Analysis

Through the static analysis phase, we obtain the function call graph and the list of the key functions of a program, and in the centrality analysis phase, we focus on extracting the centralities of the key functions in the program.

In social network analysis, centrality measures reflect the topological properties of different user nodes in a network. We observe that social networks bear some similarity to function call graphs, where social network users can be likened to function nodes in the function call graph. Therefore, using different centrality measures can quantify the topological characteristics of function nodes in the function call graph. In malware classification, different malware families exhibit different behaviors, including different API call patterns. Thus, using social centrality can enhance or weaken the importance of different APIs, thereby highlighting the API call characteristics of different malware families to achieve feature enhancement. Common centrality measures in social network analysis include degree centrality, Katz centrality, and betweenness centrality, among others. Different centrality measures calculate the topological characteristics of nodes from different perspectives. Since we need to propose a lightweight method, we abandon high computational complexity centralities like betweenness and select the centralities with reasonable computation speed. We will evaluate their effectiveness in Section 4.4.

For a given graph $G := (V, E)$ with $|V|$ nodes and $|E|$ edges, the selected centrality measures are as follows:

Degree Centrality [18] is defined as the number of links incident upon a node (i.e., the number of ties that a node has). The degree can be interpreted in terms of the immediate risk of a node for catching whatever is flowing through the network. The degree centrality of a node, is defined as $deg(v)$. Thus it can be normalized by dividing by the maximum possible degree in G, where $|V|$ is the number of nodes in G (1).

$$C_d(v) = \frac{deg(v)}{|V| - 1} \tag{1}$$

Just Accepted

Closeness Centrality [18] is the average length of the shortest path between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes. For a node v and the remaining nodes t in G, the standardized form is as (2), where $d(t, v)$ is the distance between nodes v and t.

$$C_c(v) = \frac{|V| - 1}{\sum_t d(t, v)} \tag{2}$$

Harmonic Centrality [41] is the sum and reciprocal operations in the definition of closeness centrality. For a node v and the remaining nodes t in G, the standardized form of harmonic centrality of v is as (3), where $1/d(t, v) = 0$ if there is no path from t to v.

$$C_h(v) = \frac{\sum_{t \neq v} \frac{1}{d(t,v)}}{|V| - 1} \tag{3}$$

Katz Centrality [31] computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors. The connections made with distant neighbors are penalized by an attenuation factor α. Its standardized form is defined as (4), where A is the adjacency matrix of the graph, the element at location (i, j) of A^k is the sum of k degree connections between nodes i and j.

$$C_k(v) = \sum_{k=1}^{\infty} \sum_t \alpha (A^k)_{vt} \tag{4}$$

Furthermore, in order to explore the possibility of using combinatorial centrality, we construct two artificial centrality measures named *Average Centrality* and *Concatenate Centrality*. First, we assume a set of different centrality algorithms denoted as $\{C\}$ and *MalSensor* considers four individual centralities: *Degree Centrality*, *Katz Centrality*, *Closeness Centrality*, and *Harmonic Centrality*. The two artificial centralities definitions given by us are as follows:

Average Centrality is the average value of $\{C\}$, which reflects the mean level of different centrality algorithms. Its standardized form is defined as (5).

$$C_{avg}(v) = \frac{\sum_C C_C}{card(C)} \tag{5}$$

Concatenate Centrality is the direct sum of $\{C\}$. In simple terms, it is directly concatenated by centrality measures in C and its standardized form is defined as (6).

$$C_{concat}(v) = (C_d, C_k, C_h, C_c, ...) \qquad (C_i \in \{C\}) \tag{6}$$

In short, the static analysis phase generates the function call graph of a program and then we compute the centralities of all functions in the call graph. Thus, we generate the feature vector of the program using centrality values. Each of the key functions is mapped to one dimension of the feature vector. Therefore, the centrality values of the key functions within the function call graph are filled into the corresponding positions of the feature vector, while the others not in the call graph are denoted as 0.

The above process can be described by a strict mathematical definition as follows:

We assume that all centrality algorithms form the set $\{C\}$, and n key functions defined by us form the set$\{F_{key}\}$. For a given program P, we suppose $V_{all} = (v_1, v_2, ..., v_m)$ is the list of all m functions in P. To P, we have an m-diagonal matrix $C_{centrality}$ for $\forall centrality \in \{C\}$. The element of $C_{centrality}$ denoted $c_{i(0 \leq i \leq m, i \in Z)}$ is the centrality value of v_i in the function call graph. Therefore, after centrality processing, the function vector of

Just Accepted

the program P is expressed as $V_{all}C = (c_1v_1, c_2v_2, ..., c_mv_m)$ and then we get the feature vector of P is $V_{Feature} = (c_1v_1, c_2v_2, ..., c_nv_n)(v_i \in \{F_{key}\})$.

To enhance the effectiveness and distinctiveness of the feature vector, we assign different initial weights w_i to v_i. We conduct statistical analysis on a large-scale of malicious samples and perform weight ranking of APIs based on their frequency of usage. According to this, we assign different weights ($0<w_i<1$) to each key function v_i. This will accentuate the influence of these functions in the feature vector. Finally, we get the feature vector of P is $V_{Feature} = (w_1c_1v_1, w_2c_2v_2, ..., w_nc_nv_n)(v_i \in \{F_{key}\})$.

3.4 Classification

In the final stage, we use classification algorithms to classify malware. We test 5 machine learning algorithms and find that Random Forest work better. To be more convincing, we also conduct tests on four deep learning algorithms and also achieve good performance. Particularly, in the evaluation section, if there is no special note, we adopt the Random Forest for experiments.

4 EXPERIMENT EVALUATION

In this section, we address the following research questions to evaluate the superiority of *MalSensor*.

RQ1: How effective is the function name optimization module of *MalSensor*? (§4.3)

RQ2: How well *MalSensor* performs on classification accuracy? How does it compare to state-of-the-art malware classification methods? (§4.4)

RQ3: How much faster is *MalSensor* than state-of-the-art malware classification methods? (§4.5)

RQ4: Is *MalSensor* more robust than state-of-the-art malware classification methods? (§4.6)

4.1 State-of-the-Art Methods

In order to verify the superiority of *MalSensor* over other works, we need to conduct extensive comparative experiments with current state-of-the-art methods. The comparative methods should cover various types of methods, including not only disassembly-based methods but also image-based and binary-based.

Research [39] aims to systematically study the performance of different PE malware classification works, and they select and reproduce nine state-of-the-art learning-based PE malware family classification methods for empirical study, which comprehensively represent the current state of the PE malware classification study field. The nine state-of-the-art works are as follows:

- ResNet-50 [51] is an image-based malware classification method using the ResNet network. It is fairly efficient and capable of capturing more malware information.
- VGG-16 [55] is a high-fitting capability image-based method using VGGNet.
- Inception-V3 [55] is the first work to apply Inception-V3 to malware classification using image conversion from binary.
- IMCFN [55] stands for image-based malware classification using Fine-tuned Convolutional Neural Network.
- CBOW+MLP [48] is a binary-based malware family classification approach combining Word2Vec [42] and the Muli-Layer Perception (MLP).
- MalConv [50] is the first end-to-end binary-based malware analysis model that allows the entire malware to be taken as input.
- MAGIC [58] is a disassembly-based method for classifying malware by extracting program attribute function call graph.
- Word2Vec+KNN [7] is a representative work based on disassembly extracting malware opcode and processing with Word2Vec.

Just Accepted

- MCSC [46] extracts malware opcode sequences and encodes them based on SimHash [40]. By converting SimHash value to the gray scale, it performs an image classifier to classify malware.

These nine state-of-the-art works are all competitors in our comparative experiments below.

4.2 Dataset and Configurations

4.2.1 Dataset. We use three datasets for our experimental evaluation: Mal-15000 [5], Malwarebazaar [39], and MalwareDrift [39]. To ensure the purity and validity of the analyzed samples, we utilize PyPackerDetect [47] to perform packing detection on the malicious samples with six determined strategies (PEID/Known packer section names/Non-standard entrypoint/Threshold of non-standard sections reached/Imports number/Overlapping entrypoint).

Mal-15000. To further verify the *MalSensor* static analysis capability and optimization effect, we download the 2018-2021 malicious PE sample compression packages with tens of thousands of samples from virusshare [5]. According to the hash value sorting, we randomly screen 15,000 unpacked malicious samples with different architectures and families as the dataset Mal-15000.

Malwarebazaar.

Ma et al. [39] respectively download 1,000 recently uploaded malware samples from each of the top 6 malware families in recent years from the MalwareBazaar website [3] to construct the dataset named *Malwarebazaar*. In order to ensure the validity of the dataset, they eliminate samples in non-PE format and then utilize AVClass [52] and Joe Security [38] to check the labels of the samples and then cull noise samples with inconsistent labels from different websites. Finally, MalwareBazaar consists of 3,971 PE malware samples with six family labels (as shown in Table 1).

Table 1. Details of MalwareBazaar and MalwareDrift Datasets.

Dataset	Family Name	Number of Samples	Total
MalwareBazaar	Gozi	767	3971
	GuLoader	589	
	Heodo	214	
	IcedID	578	
	NjRat	942	
	Trickbot	881	
MalwareDrift	Bifrose	278	3125
	Ceeinject	548	
	Obfuscator	204	
	Vbinject	1032	
	Vobfus	282	
	Winwebsec	487	
	Zegost	294	

MalwareDrift. Wadkar et al.[57] reveal that code changes show up as sharp spikes in the χ^2 timeline statistic. Similarly, the evolution of a malware family can be understood like as code changes. Ma et al. [39] use the dataset and the corresponding χ^2 timeline statistic graph from the study [57] to determine the evolution period time of each malware family and divide each family samples into the pre-drift or post-drift. Through the above steps, the research [39] generates the dataset MalwareDrift containing 3125 samples with 7 families as shown in Table 1 which are used for testing the impact of time lapse on the classification methods. The samples in pre-drift and post-drift are respectively before 2015 and after 2020.

Just Accepted

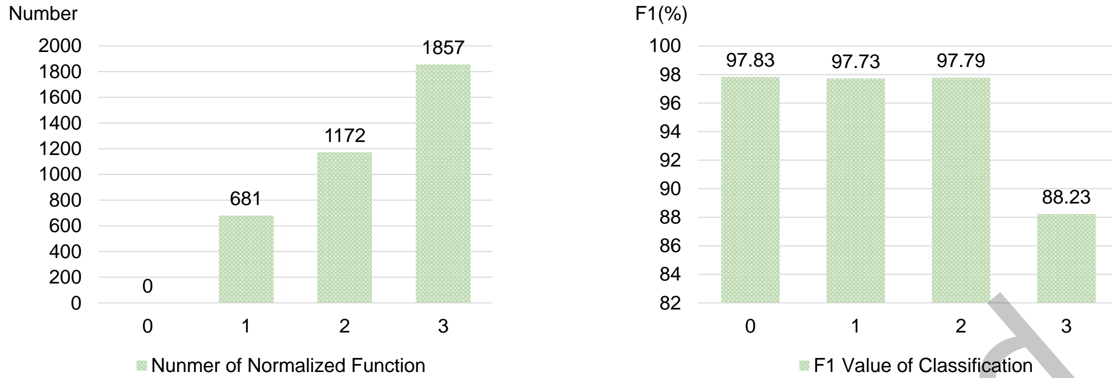


Fig. 9. Normalized functions number and classification F1 values corresponding to n.

4.2.2 Configurations. Following the experimental Configurations of the research [39] on nine representative works, we evaluate *MalSensor* using 10-fold cross-validation and use the macro average metrics of False Positive Rate (FPR), *Accuracy (A)*, *Precision (P)*, *Recall (R)*, and *F1-Score (F1)* to demonstrate the effectiveness. Especially, to ensure the experiment reliability, the model parameters of the MLP, VGG-16, ResNet-50, and Inception-V3 used by *MalSensor* are consistent with the configurations in research [39]. All the experiments are conducted on a server with 2 Intel Xeon Platinum 8260 CPUs @2.30GHz and 4 Nvidia GeForce RTX2080 Ti GPUs (11GB), and 512GB RAM.

It is worth noting that the experimental performance of the nine state-of-the-art works in the research [39] are all reproduced to their respective best results.

4.3 Static Analysis Effectiveness

In Section 3.2, we propose to use maximum substring matching for function normalized optimization. Therefore, we first determine the optimal value of n experimentally. For the consistency of subsequent experiments, we use the dataset MalwareBazaar for experiments. Specifically, after obtaining malware function call graphs and the sensitive function list, we prune the function names by varying the size of n and then perform malware classification using KNN. The number of functions normalized by different values of n and the corresponding F1 value of their classification results are shown in Fig. 9. From the experimental results, we can see that when n is 1 or 2, the classification performance is stable, which is similar to the performance of disabling function name optimization. When n is greater than 2, the classification accuracy drops significantly. This is because irrelevant functions may be aggregated together as n increases, negatively affecting classification performance. In addition, when n is 2, the number of normalized functions is larger than n is 1, which can bring lower overhead for subsequent processing. Therefore, we determine that the optimal n value is 2. This is also in line with common sense, as we find that most of the redundant function names come from the difference in suffixes, that is, the suffixes(e.g., A, W, 32, 64, etc.) are used to distinguish the different processing objects.

Then, we evaluate the effectiveness of the function name optimization module (n=2) for malware analysis. Under the same experimental conditions, we compare the analysis results obtained by *MalSensor* with function name optimization enabled or not. The differences brought by these two analyses are visually demonstrated in terms of the number of key functions (i.e., dimension of feature vectors in classification) and the learning model training time, as shown in Table 2 and Fig. 10. Furthermore, when generalizing to a wider range of real-world scenarios, the benefits of this optimization will be more obvious. We perform the same comparison experiment

Just Accepted

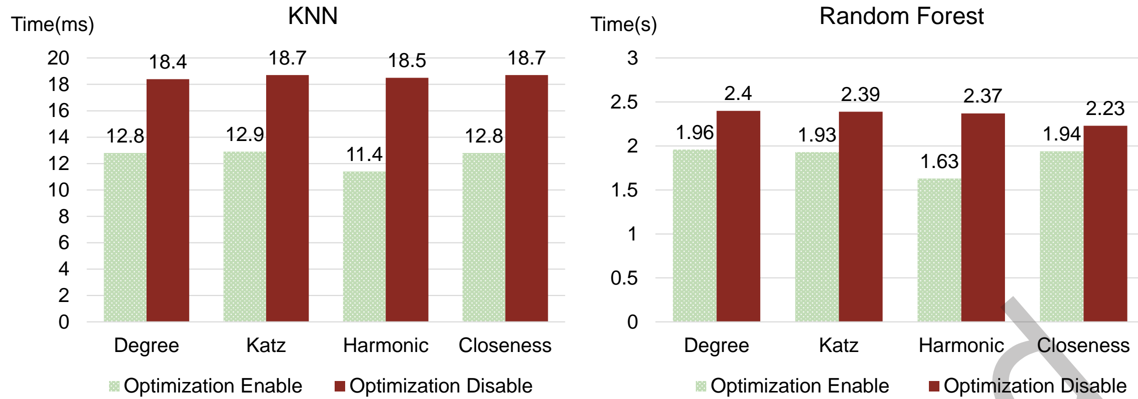


Fig. 10. Training time of different machine learning algorithms with different configurations on dataset MalwareBazaar.

on dataset Mal-15000, which has more sample counts, file types, and architectures, and the results are shown in Table 2 and Fig11.

Table 2. Number of key functions with optimization enabled or disabled.

Dataset	Optimization Disable	Optimization Enable	Decline(%)
MalwareBazaar	5243	4071	22
Mal-15000	24625	12628	49

From the experimental results, there are a total of 5243 key functions obtained after disassembly analysis on the dataset MalwareBazaar. After the optimization module is turned on, the number drops to 4071, which reduces redundancy. As for dataset Mal-15000, the function number drops from 24625 to 12628, with a more obvious effect. This has a positive effect on accurately building function call graphs of each sample and data processing such as feature extraction and learning model training. To verify the effectiveness of the optimization module for model learning, we select two machine learning algorithms: KNN (3-NN) and Random Forest, and use four centralities (degree, katz, harmonic, closeness) for model training. Benefiting from the reduction of key functions (i.e., the reduction of the dimension of feature vectors), the speed of model training improves a lot. On the dataset MalwareBazaar, before and after the optimization is turned on, the training time of KNN based on all centralities is shortened by at least 31%, and the KNN with harmonic centrality has the most significant benefit, which is shortened by 38%. For the Random Forest algorithm, the training time of the model is shortened by more than 13%, of which the training time combined with harmonic is reduced by 31%. One possible reason for the faster training speed of harmonic is that the data processed by harmonic is less discriminative and easier to calculate. Later we can see that the classification effect based on harmonic is indeed weaker than other centralities. When it comes to dataset Mal-15000, the training time is reduced by 50-60%, and the optimization effect is more obvious. This is because in large-scale data, with the growth of vector dimension, the training overhead will increase greatly.

Just Accepted

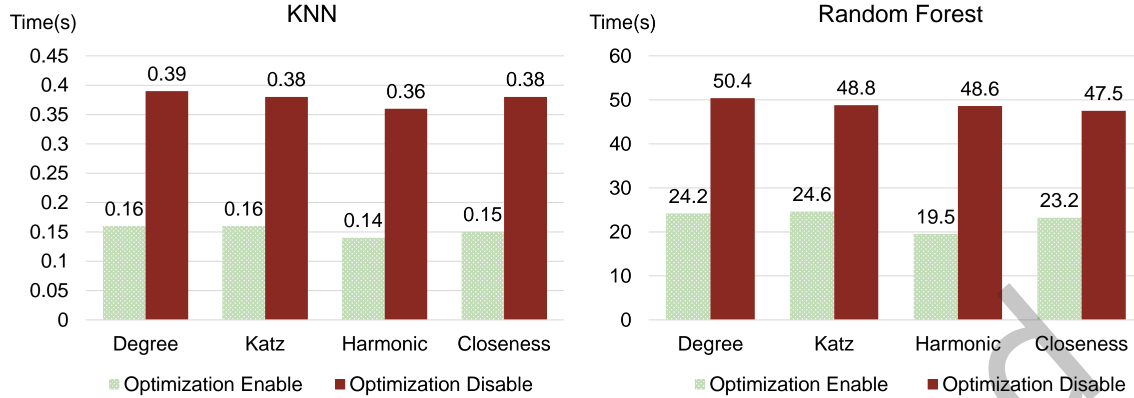


Fig. 11. Training time of different machine learning algorithms with different configurations on dataset Mal-15000.

4.4 Classification Effectiveness

To evaluate the effect of our optimized method on malware family classification, we conduct experiments on the MalwareBazaar dataset. Specifically, we evaluate the performance of our work using nine machine learning algorithms (including deep learning) for malware family classification. Subsequently, we compare our work with nine state-of-the-art malware classification works under the same experimental conditions.

We first perform five sets of classification experiments using machine learning algorithms KNN(1-NN, 3-NN), SVM, Decision Tree, and Random Forest combining different centralities (degree/katz/harmonic/closeness as formulas 1-4). To provide more convincing and comprehensive results, we also employ various deep learning models, including MLP, VGG-16, ResNet-50 and Inception-V3, for classification. In particular, to prepare the feature vector for input into the CNN models, we reshape the one-dimensional feature vector generated by previous analysis to match the width of the CNN input. Since centralities are often combined to measure the importance of nodes in a network, we add two additional experiments by artificially constructing the integrated centrality (average centrality/concatenate centrality), as shown in formulas 5, 6. In addition, to minimize the randomness of the experiments, we perform 10-fold cross-validation and use F1, Accuracy, Recall, and Precision as evaluation metrics.

The experimental results are shown in Table 3 and we can see that according to the selected centrality, the classification ability of *MalSensor* is different. The classification based on closeness centrality can achieve the best effect of using individual centrality on most of the learning algorithms, and this may be due to the strong similarity in the topological positions of key functions on the function call graph of the same family malware. The effect of concatenate centrality is usually better than that of single centrality, showing an F1 value of 98.35% in the experiment, because it combines the focuses of various centralities, which makes the features display more comprehensive.

In addition, classification performance varies depending on the learning algorithms. For instance, when using concatenate centrality, the Decision Tree shows the worst performance with an F1 value of 97.14%, while the Random Forest achieves the best performance with an F1 value of 98.35%. This is mainly because Random Forest is based on ensemble learning. It combines multiple decision trees into a strong classifier to improve classification performance. Compared to a single decision tree or individual classifiers, the voting mechanism of multiple decision trees can better capture complex relationships and nonlinear features in the data. This effectively reduces the influence of noise and the risk of overfitting. As some malware samples are very small, it may only be possible

Just Accepted

Table 3. Performance of Different Learning Algorithms with Different Centralities.

ML Algorithm	Centrality	Classification Performance(%)				
		A	*P*	*R*	*F1*	*FPR*
1-NN	Degree	97.70	97.88	97.98	97.93	2.08
	Katz	97.85	97.95	98.08	98.01	2.05
	Harmonic	97.58	97.89	97.78	97.83	2.10
	Closeness	97.88	98.05	98.05	98.04	1.89
	Average	97.79	97.92	97.99	97.96	2.03
	Concatenate	98.05	98.11	98.14	98.11	1.85
3-NN	Degree	97.05	97.27	97.41	97.34	2.66
	Katz	97.00	97.14	97.26	97.20	2.79
	Harmonic	96.97	97.09	97.23	97.16	2.85
	Closeness	97.38	97.67	97.68	97.67	2.43
	Average	97.10	97.33	97.42	97.35	2.61
	Concatenate	97.48	97.74	97.82	97.79	2.12
Random Forest	Degree	98.01	98.21	98.13	98.18	1.69
	Katz	97.98	98.10	97.94	98.02	1.71
	Harmonic	97.80	97.93	97.86	97.88	2.00
	Closeness	98.13	98.32	98.24	98.25	1.55
	Average	97.96	98.13	98.06	98.10	1.88
	Concatenate	98.26	98.41	98.31	**98.35**	1.39
Decision Tree	Degree	96.53	96.21	96.73	96.68	3.77
	Katz	96.77	96.90	96.94	96.92	3.01
	Harmonic	96.91	96.82	96.76	96.88	3.09
	Closeness	97.13	96.82	96.94	97.02	3.09
	Average	97.06	96.93	96.96	97.00	3.03
	Concatenate	97.09	97.07	97.21	97.14	2.91
SVM	Degree	97.80	97.98	98.01	97.98	1.99
	Katz	97.86	97.92	98.02	97.93	2.01
	Harmonic	97.03	97.30	97.25	97.22	2.67
	Closeness	97.06	97.52	97.24	97.34	2.42
	Average	97.65	97.70	97.82	97.79	2.29
	Concatenate	97.96	98.01	98.07	98.05	1.91
MLP	Degree	97.87	98.10	98.07	98.03	1.88
	Katz	97.87	98.23	98.12	98.10	1.75
	Harmonic	97.89	97.99	98.02	97.97	1.98
	Closeness	97.97	98.30	98.14	98.21	1.67
	Average	98.14	98.23	98.17	98.19	1.56
	Concatenate	98.30	98.39	98.16	98.33	1.59
VGG-16	Degree	97.71	97.92	97.91	97.88	2.03
	Katz	97.79	98.09	98.02	98.00	1.89
	Harmonic	97.82	97.96	98.05	97.95	2.00
	Closeness	98.00	98.16	98.07	98.10	1.85
	Average	97.94	98.11	98.03	98.04	1.79
	Concatenate	98.21	98.28	98.19	98.24	1.68
ResNet-50	Degree	97.94	98.14	98.10	98.07	1.83
	Katz	97.92	98.20	98.13	98.09	1.77
	Harmonic	97.90	97.99	98.12	98.03	1.97
	Closeness	98.02	98.32	98.18	98.19	1.68
	Average	97.96	98.17	98.14	98.11	1.79
	Concatenate	98.39	98.43	98.20	98.31	1.55
Inception-V3	Degree	98.08	98.22	98.19	98.16	1.77
	Katz	97.98	98.29	98.18	98.21	1.68
	Harmonic	98.03	98.11	98.15	98.07	1.88
	Closeness	98.18	98.41	98.18	98.29	1.56
	Average	98.11	98.31	98.18	98.19	1.69
	Concatenate	98.24	98.32	98.18	98.26	1.71

Just Accepted

to extract a limited number of API features, which can interfere with the judgment of classifiers. Compared to other machine learning methods, Random Forest have better robustness to outliers and missing values, thus it can exhibit better performance. Additionally, the deep learning algorithms perform well, with F1 values above 98.24%. However, they do not surpass the performance of the Random Forest. A main reason is that since the sensitive function features extracted from FCGs do not have the logical relationships in the feature space context, the feature processing of the VGG-16, ResNet-50, and Inception-V3 models introduce non-existing spatial correlations, which may negatively impact their performance. This is also the reason why relatively simple MLP performs better than other deep learning models.

Table 4. Performance of all methods on MalwareBazaar.

Category	**Model**	**Classification(%)**				**Train**	**Reasource Overhead**		
		A	*P*	*R*	*F1*	**min)**	Mem(GB)	GPU.Mem(GB)	GPU(%)
Image	ResNet-50	96.68	96.91	96.75	96.83	8.4	17.4	10.8	95
	VGG-16	96.35	96.58	96.54	96.56	44.0	18.43	10.8	97
	Inception-V3	95.83	95.67	95.79	95.73	6.4	12.3	10.8	96
	IMCFN	97.38	97.53	97.41	97.47	18.8	22.5	10.8	92
Binary	CBOW+MLP	97.81	97.92	98.08	98.00	0.8	52.0	10.4	34
	MalConv	95.92	96.04	96.43	96.20	65.4	246.8	10.8	60
Disassembly	MAGIC	92.82	88.03	87.36	87.45	246.0	114	10.6	81
	Word2Vec+KNN	95.64	93.34	94.29	93.79	<0.1	5.5	-	-
	MCSC	96.80	94.97	94.51	94.70	1.1	45.3	10.8	33
	MalSensor	98.26	98.41	98.31	**98.35**	0.3	4.8	-	-

To evaluate the performance of our work more extensively, we observe the experimental results of all existing state-of-the-art works under the same conditions, as shown in Table 4. In particular, we choose the Random Forest algorithm for *MalSensor* to conduct the rest of the comparative experiments. From the experimental results, we can see that among all the existing state-of-the-art works, IMCFN achieves the best F1 performance of 97.47% in image-based methods. As for the binary-based methods, CBOW+MLP has 98% F1, also the best among all existing methods. Disassembly-based methods perform poorly, 3%-11% lower than the methods of other categories. One possible reason is that, unlike image-based and binary-based methods that directly analyze raw file features, disassembly-based methods are limited by the analysis strategies and capabilities of static analysis tools, and the method of feature extraction also greatly affects the results.

In contrast, *MalSensor*, which is also based on disassembly, can have a well and stable performance no matter which centrality is used. Its F1 is 3-11 percent higher than other disassembly-based methods and this is mainly due to our finer program analysis method and more suitable feature processing. Looking at all methods, only the performance of CBOW+MLP comes close to *MalSensor*, however, it is time-consuming, which will be discussed later. Overall, *MalSensor* outperforms **all existing state-of-the-art methods** in accurately classifying malware, and this benefits from our more accurate and efficient characterization of program semantic.

4.5 Running Time Overhead

In addition to effectiveness, another significant factor affecting the practicality of the PE malware classification is the runtime overhead and the requirement of computing resources. This is because some application scenarios are on devices with limited computing resources. Therefore, in this section, we demonstrate that *MalSensor* not only outperforms other methods in effectiveness but also has a huge advantage in resource overhead. We compare

Just Accepted

the runtime overhead of *MalSensor* with the other representative methods at *Training time*, *Preprocessing time*, and *Prediction time* based on the MalwareBazaar dataset. In particular, the prediction times based on different centralities are almost the same. And the resource consumption of all methods in the three stages is shown in Table 4, 5.

Table 5. The average runtime overhead for one sample

Category	Model	Time	
		Pre-process(s)	Predict(ms)
Image	ResNet-50	0.7	2.6
	VGG-16		2.2
	Inception-V3		2.0
	IMCFN		2.2
Binary	CBOW+MLP	1.1	5441.0
	MalConv	0.3	14.0
Disassembly	MAGIC	17.8	23.6
	Word2Vec+KNN	17.7	95243.3
	MCSC	4.5	3477.8
	MalSensor	0.7	0.2

Training. During this stage, the runtime overhead will vary greatly depending on the category of method and the learning algorithm used. As Table 4 shows, the graph-based methods ResNet-50, VGG-16, Inception-V3, and IMCFN require 8.4, 44.0, 6.4, and 18.8 minutes of training time, respectively. This is mainly because complex deep learning model requires longer training time and computational resource, and their preprocessing is too simplistic in information sifting. The binary-based method CBOW+MLP takes 0.8 minutes for model training, which is much less than 65.4 minutes for the similar method MalConv. This is because CBOW+MLP filters the byte stream of the raw binary file in the preprocessing stage, removes meaningless bytes (which is why it takes longer preprocessing time), and uses a more efficient vector embedding method (Word2Vec).

As for *MalSensor*, it only needs at most 0.3 minutes (depending on the centrality) to complete the model training, which is several times to hundreds of times faster than the other methods except Word2Vec+KNN (its $F1$ is 4.5% lower than *MalSensor*). KNN-based *MalSensor* can also achieve the same training speed as Word2Vec+KNN, and the accuracy is 4.3% higher than it. This is because the features extracted by *MalSensor* are concise and effective, and the learning method used is simple but useful.

Preprocessing. At the preprocessing stage, the image-based methods take 0.7s to convert the binary of the raw file into an image, and the binary-based methods take 1.1s and 0.3s, respectively, depending on the preprocessing granularity. However, the disassembly-based methods require 4.5s, 17.7s, and 17.8s, respectively, according to the extracted features, which are much higher than the previous two categories. This is mainly because the preprocessing of the image or binary-based method is very simple, which only involves the process of mapping and converting the file binary, while the disassembly-based method requires a more complex analysis.

However, as a disassembly-based method, *MalSensor* only takes 0.7s of preprocessing time, which is 6-25 times faster than similar methods and is close to the runtime overhead of image and binary-based methods using simple processing. This is because the static analysis module cuts out all unnecessary steps and only extracts concise key function information. It is worth mentioning that the time overhead of computing centrality for each sample is in milliseconds.

Just Accepted

Prediction. At the prediction stage, for each sample, the image-based methods have excellent performance and take 2ms to predict. However, the prediction time of CBOW+MLP, MCSC, and Word2Vec+KNN explodes, with 5,441ms, 3,477ms, and 95,243ms, respectively. Their single sample prediction time is too long, which greatly hinders practical application. The performance of *MalSensor* at this stage is 0.2ms, which is 118-476,216 times faster than similar methods. Compared with other categories methods, the prediction speed of *MalSensor* is improved by at least 10 times as a result of its efficient and strong specificity feature extraction.

Overall, compared to all representative works, *MalSensor* has a great advantage in running time overhead, and this makes the practical meaning of the method even more strong. Furthermore, *MalSensor* performs well without GPUs or a large memory runtime environment, and thus, it is lower than most methods in research [39] in terms of hardware resource requirements.

4.6 Performance under Concept Drift

Malware is known to evolve rapidly over time, and concept drift (i.e., statistical properties of objects change in unforeseen ways) has become a rather challenging problem in malware classification [28]. Concept drift can be understood as changes in the code composition and file attributes of malware as the malware family evolves or drifts. The ability of one method to adapt concept drift scenarios determines whether it has the ability to deal with newer malware and the possibility of wide application. Therefore, it is important to evaluate the performance of one method in the application scenario of concept drift.

To evaluate the concept drift scenario, we use the dataset MalwareDrift for the experiment. In order to show the generalization ability of a model trained on an older dataset to a newer dataset, we train and perform cross-validation testing on the pre-drift data for each classifier, and the experimental results are shown in the "pre-pre" row of Table 6. Subsequently, we load the models trained on the pre-drift data and directly test them on the post-drift data. The experimental results for each classifier in this scenario are shown in the "pre-post" row of Table 6. This allows us to observe the performance of the classifiers trained on early data with newer data. In Table 6, the depth of red represents the degree to which the method is affected by concept drift. The darker the red, the more performance degradation. In particular, IMCFN is chosen as the representation of the image-based methods as it has the best overall performance in previous experiments.

We can see from the table that the other 9 representative works have at least a 27.07% drop in F1 score when facing concept drift in the real environment, and the maximum drop even reaches 69.62%. In contrast, the F1 drop ratio of *MalSensor* is 20.51%, which is lower than all other methods. This shows that *MalSensor* trained on older samples has stronger adaptability to newer samples than the other works, which benefits from its deep understanding of program semantics.

4.7 Practicability

In this section, we will further discuss the applicability of *MalSensor* in the real world. Attackers in the real world often use evasion techniques such as obfuscation and packing to interfere with malware analysis. For example, they may alter the internal structure of malware by adding irrelevant code and increasing control flow complexity, making it appear different from other malware in the same family (i.e., polymorphism). To evaluate the impact of various evasion techniques, we process the MalwareBazaar dataset using mature obfuscation and packing tool, Virbox Protector [4] and binary editor 010editor [1]. We apply 9 typical obfuscation techniques and packing techniques to the malicious samples, as shown in Table 7.

We evaluate *MalSensor* on datasets processed with these evasion techniques and compare the results with those on the original *MalwareBazaar* dataset in Table 8. It can be seen that *MalSensor* exhibits good performance when faced with evasion techniques such as Constant Values Encryption, Resource Obfuscation, and Section Appending. This is because *MalSensor* focuses on the calling behavior of key APIs, unaffected by changes in

Just Accepted

Table 6. Impact of concept drift on performance

Category	Model	Test Strategy	Classification Performance (%)			
			A	*P*	*R*	*F1*
Image	IMCFN	pre-pre	85.21	85.23	83.00	83.91
		pre-post	49.90	52.74	44.42	42.10
		decrease	35.31	32.49	38.58	41.81
Binary	CBOW+MLP	pre-pre	81.69	83.36	78.58	80.50
		pre-post	17.44	11.52	14.45	10.88
		decrease	64.25	71.84	64.13	69.62
	MalConv	pre-pre	77.43	78.49	74.97	76.19
		pre-post	46.50	39.19	39.49	35.51
		decrease	30.93	39.30	35.48	40.68
Disassembly	MAGIC	pre-pre	80.85	81.29	77.77	79.30
		pre-post	42.15	34.69	33.07	28.30
		decrease	38.70	46.60	44.70	51.00
	Word2Vec+KNN	pre-pre	81.85	80.77	79.79	80.12
		pre-post	49.18	48.73	51.80	43.87
		decrease	32.67	32.04	27.99	36.25
	MCSC	pre-pre	74.31	69.44	73.49	70.89
		pre-post	50.58	46.97	48.25	43.82
		decrease	23.73	22.47	25.24	27.07
	MalSensor	pre-pre	80.62	80.94	78.69	80.32
		pre-post	63.25	56.38	59.08	58.46
		decrease	17.37	24.56	19.61	21.86

Table 7. Impact of concept drift on performance

Obfuscation/Packing	Descriptions
Function Name Encryption	Encrypt function names
Constant Values Encryption	Encrypt constant strings
Resource Obfuscation	Modify the resources section of PE files
Goto	Modify the control-flow graph by adding two new nodes
NOP	Insert random nop instructions within every method implementation
Call Indirection	Modify the control-flow graph without changing the code semantics
Section Rename	Rename the section names of PE files
OEP Modification	Modify the original entry point
Section Appending	Add additional sections to PE files

irrelevant code. However, when dealing with Function Name Encryption, *MalSensor* shows a slight performance decrease. This negative effect is because Function Name Encryption makes function names less understandable, thus affecting the function name normalization process in *MalSensor*. Moreover, *MalSensor* also faces performance degradation when dealing with control flow obfuscation like Goto, Nop, and Call Indirection. To ensure normal program operation, control flow obfuscation does not introduce significant interference with key function calls significantly. However, these obfuscation introduces new function nodes and alters the topological structure of the program function call graph. This indirectly affects the centrality weight of key API functions, thereby impacting the classification performance.

Just Accepted

Table 8. Impact of concept drift on performance

Obfuscation/Packing	Classification Performance (%)			
	A	*P*	*R*	*F1*
Function Name Encryption	97.65	97.74	97.41	97.57
Constant Values Encryption	98.11	98.31	98.09	98.19
Resource Obfuscation	98.19	98.29	98.18	98.22
Goto	97.69	97.06	97.68	97.31
NOP	97.80	97.79	97.98	97.91
Call Indirection	97.77	95.66	96.58	96.34
Section Rename	98.21	98.16	98.01	98.08
OEP Modification	98.03	98.11	97.98	98.07
Section Appending	98.24	98.34	98.22	98.31
Clean	98.26	98.41	98.31	**98.35**

5 DISCUSSION

Limitation. Static analysis has some inherent limitations. Since static analysis does not actually execute the target program, its analysis results are susceptible to various obfuscation methods [26]. Our work is based on static analysis and therefore is also subject to the inherent limitations of static analysis. For example, attackers can load a constant value into a register without the static analyzer knowing its value by using opaque constants. This mechanism allows attackers to perform a number of transformations that obfuscate the control flow, data locations, and data usage of a malicious program, without changing its actual behavior [43]. In this case many jumps with constant addresses operands will not be analyzed correctly. When function names are completely randomized, MalSensor cannot construct the necessary features for family classification based on function call relationships, because the same function may have completely different names in different programs. Therefore, it is necessary to use dynamic analysis, fuzzing testing, and other methods to recover function names through inference. In addition, adding fake functions is also a way to interfere with our analysis. Since our work is based on FCG, adding fake functions to a malicious program that do not affect its behavior can alter the FCG of the program, thereby affecting the classification accuracy. There are also ways to resist static analysis by program packing or hiding externally imported functions of the program. There are many mature commercial packing tools that can create complex shells, such as multi-layer and virtualization shells. These shells are very difficult even for manual unpacking and require dynamic analysis. In such cases, since we cannot obtain useful information by static analysis, it is nearly impossible to make an effective judgment. Therefore, it is challenging for our work to serve as an end-to-end processing method, and we need to apply various anti-obfuscation techniques before inputting the original program.

Future Work. In our future work, we will further refine the feature extraction process of *MalSensor* and train the classifier with more artificial obfuscated samples to achieve higher accuracy and robustness. Furthermore, we will try more social centrality and deep learning models for experiments. We will also attempt to incorporate various anti-obfuscation preprocessing techniques into MalSensor, such as automatic unpacking modules, to alleviate the limitations imposed by static analysis. We also intend to explore the potential benefits of hybrid analysis, combining both static and dynamic analysis approaches. This approach will effectively compensate for any potential misjudgments in program behavior that may arise from static analysis alone. Additionally, we find that the dataset used in the concept drift experiment can not be overly outdated. This is because there may be samples running on an outdated version of Windows, or using outdated API and library functions, which have a certain negative impact on disassembly analysis. All of these aspects will serve as guidance for our future work.

Just Accepted

Future Trend. We believe that work in the field of static analysis-based malware classification should be closer to the needs of industry. Current learning-based static malware classification methods are not lightweight or accurate enough for large-scale analysis, especially on resource-critical devices, such as network gateway devices with limited computing resources [39]. The current work often does not pay enough attention to the issues of resource consumption, which leads to the difficulty of practical application. In addition, it is necessary to propose more effective feature extraction methods to deal with concept drift scenarios in the real world.

6 RELATED WORK

The PE malware classification based on static analysis and machine learning is divided into three categories: methods based on binary graph transformations; methods based on byte stream sequences; methods based on disassembly.

Image-Based techniques first transform the malware binary files into gray scale images, and then adopt image classification models for malware classification. Nataraj et al. [45] first combine malware analysis with image analysis techniques to convert binary files into gray scale images, where each byte corresponds to a pixel in the image with a gray scale value and then the image is classified using KNN. Kancherla et al. [30] further refined the image processing method and used the SVM algorithm for classification in combination with Gabor filter extraction features. Afterwards, Ahmadi et al. [6] extract new features such as Haralick and Local Binary Pattern for classifying malware using boosting tree classifiers. However, these methods are usually inefficient due to the high overhead of extracting complex texture features. With the development of deep learning in the field of image classification, image-based malware analysis has also become more mature. For example, Convolutional Neural Network (CNN) has been widely used in various malware classification work [20, 24, 29, 55, 56], and has excellent performance. Although image-based methods do not require specific expertise and there are many well-established image classification models available, binaries transforming malware into images introduce new hyper-parameters (e.g., image width) and might impose non-existing spatial correlations between pixels, which may be wrong [19]. Compared to our work, image-based methods require significant computational resources, such as GPUs, which can not meet the lightweight requirements of the industrial scenarios with limited resources.

Binary-Based approaches treat the binary content of malware as sequential information and process it using sequential models(such as models in the NLP domain). Moskovitch et al. [44] first propose a malware classification method based on text classification technology. They extract n-grams from the training data, select the top 5,500 as features according to a custom Document Frequency score, and then use learning algorithms for classification. Jain et al. [25] use a technique called classwise document frequency to reduce the feature space to improve the analysis method and Fuyong et al. [59] use information gain to filter K n-grams as features, but they still have the disadvantage that the computational cost increases exponentially as the value of n increases. Raff et al. [50] take the entire binary file as input for malware classification using an end-to-end shallow CNN model. However, its processing power is limited due to the large amount of memory required. Qiao et al. [48] convert byte values of PE files into 256 words and combine the Word2Vec model to represent the malware as a word embedding matrix, and then utilize the multilayer perceptron (MLP) for classification. The binary-based methods consider the contextual information of a program to a certain extent, but because the program often has instruction jumps and function calls, the byte contextual information is not always relevant, and thus, this interferes with the classification. Moreover, Binary-based methods also need to consume high system resources, because the size of the malware byte sequences may reach several million-time steps by treating each byte as a unit in a byte sequence [50]. As our work is based on function call analysis representing program semantic information, we do not encounter the issue of lost instruction jumps. Furthermore, the computation complexity of employing social network analysis for feature enhancement and vector embedding is significantly lower than that of byte-stream analysis models.

Just Accepted

Disassembly-Based methods disassemble the raw PE file and extract information from the disassembly result for malware classification. Opcodes, frequency of API function calls, Function Call Graph (FCG), Control Flow Graph (CFG), etc. are often used as features for classification. Hu et al. [23] present a classification approach based on opcode N-gram features which are extracted from the disassembly result of malware, and Ahmadi et al. [6] use the frequency of a subset of API calls, extracted from an analysis on 500 K malware samples, to build a multimodal system to classify malware into families. Ficco [16] combines multiple features, including API call frequency, and utilizes a hybrid classifier for joint decision-making. This combination achieves promising results. Awad et al. [7] treat the opcode sequence of each disassembly file as a document and generate a computational representation of the document using Word2Vec. Then, they classify these documents with the Word Mover's Distance (WMD) [35]metric and K-Nearest Neighbors. SimHash [46] and MinHash [54] utilize hash projection to convert opcode sequences into vectors, which are then visualized as images for classification. However, none of them fully reflect the semantics of the program due to the lack of program execution logic information. Kinable et al. [32] first use graph matching techniques to calculate the similarity score between two program FCGs and regard it as a distance metric for malware clustering. Afterwards, Kong et al. [33] abstract malware into an AFCG, and then learn malware distance metrics to distinguish different AFCGs. Unfortunately, the above methods are computationally intensive and cannot generalize well. Hassen et al. [22] adopt Minhash [11] to cluster similar FCGs, and then use a deep learning model to represent the graphs as vectors for classification. Similarly, Yan et al. [58] utilize Deep Graph Convolution Neural Network (DGCNN) [60] to aggregate the attributes of the AFCGs extracted from disassembly files and perform well. Disassembly can better capture program semantics, but they require domain knowledge, such as assembly language and disassembly analysis methods. Moreover, the classification effectiveness is greatly affected by the manually selected feature vectors from the disassembly results. In addition, some of the methods require a large amount of computation when calculating the similarity between graphs, which will bring huge performance overhead. Compared to these disassembly methods, our work can extract more function calls that are easily overlooked by conventional approaches. Additionally, we streamline the disassembly information through normalization, effectively reducing redundancy of the feature information. These optimizations not only enhance the classification capability but also improve the analysis speed of our method.

7 CONCLUSION

In this paper, we present an efficient and lightweight PE malware classification method based on program semantics and implement the prototype system *MalSensor* with a custom static analysis module and centrality analysis. The characteristics of *MalSensor* are very consistent with the development trend of future malware classification proposed by ENISA [15] and the research [39], that is, to pay more attention to program semantic features, to be lighter and more accurate. Through experimental comparison with various existing PE malware static classification methods, we have verified that *MalSensor* not only outperforms other methods in terms of accuracy and robustness but also significantly reduces resource overhead. These show that the application value and scalability of *MalSensor* are higher than that of most existing methods.

ACKNOWLEDGMENTS

The work is supported by the National Natural Science Foundation of China (62172168).

REFERENCES

[1] 2023. 010editor. https://www.sweetscape.com/010editor/.
[2] 2023. IDA7.0. https://www.hex-rays.com/products/ida/news/.
[3] 2023. MalwareBazaar Homepage. https://bazaar.abuse.ch/.
[4] 2023. Virbox Protector. https://shell.virbox.com/.

Just Accepted

[5] 2023. VirusShare. https://virusshare.com/.
[6] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. 2016. Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, Elisa Bertino, Ravi S. Sandhu, and Alexander Pretschner (Eds.). ACM, 183–194. https://doi.org/10.1145/2857705.2857713
[7] Yara Awad, Mohamed Nassar, and Haïdar Safa. 2018. Modeling Malware as a Language. In *2018 IEEE International Conference on Communications, ICC 2018, Kansas City, MO, USA, May 20-24, 2018*. IEEE, 1–6. https://doi.org/10.1109/ICC.2018.8422083
[8] Neil Balram, George Hsieh, and Christian McFall. 2019. Static malware analysis using machine learning algorithms on apt1 dataset with string and pe header features. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 90–95.
[9] Tamy Beppler, Marcus Botacin, Fabricio Ceschin, Luiz E. S. Oliveira, and André Grégio. 2019. L(a)ying in (Test)Bed - How Biased Datasets Produce Impractical Results for Actual Malware Families' Classification. In *Information Security - 22nd International Conference, ISC 2019, New York City, NY, USA, September 16-18, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11723)*, Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis (Eds.). Springer, 381–401. https://doi.org/10.1007/978-3-030-30215-3_19
[10] Niket Bhodia, Pratikkumar Prajapati, Fabio Di Troia, and Mark Stamp. 2019. Transfer Learning for Image-based Malware Classification. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy, ICISSP 2019, Prague, Czech Republic, February 23-25, 2019*, Paolo Mori, Steven Furnell, and Olivier Camp (Eds.). SciTePress, 719–726. https://doi.org/10.5220/0007701407190726
[11] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings*, Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro, and James A. Storer (Eds.). IEEE, 21–29.
[12] Aniket Chandak, Wendy Lee, and Mark Stamp. 2021. A Comparison of Word2Vec, HMM2Vec, and PCA2Vec for Malware Classification. *CoRR* abs/2103.05763 (2021). arXiv:2103.05763 https://arxiv.org/abs/2103.05763
[13] Lingwei Chen, William Hardy, Yanfang Ye, and Tao Li. 2015. Analyzing File-to-File Relation Network in Malware Detection. In *Web Information Systems Engineering - WISE 2015 - 16th International Conference, Miami, FL, USA, November 1-3, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9418)*, Jianyong Wang, Wojciech Cellary, Dingding Wang, Hua Wang, Shu-Ching Chen, Tao Li, and Yanchun Zhang (Eds.). Springer, 415–430. https://doi.org/10.1007/978-3-319-26190-4_28
[14] Nigel Coles. 2001. It's not what you know—it's who you know that counts. Analysing serious crime groups as social networks. *British Journal of Criminology* 41, 4 (2001), 580–594.
[15] ENISA. 2021. ENISA Threat Landscape 2021. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2021.
[16] Massimo Ficco. 2022. Malware Analysis by Combining Multiple Detectors and Observation Windows. *IEEE Trans. Computers* 71, 6 (2022), 1276–1290. https://doi.org/10.1109/TC.2021.3082002
[17] Linton C Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.
[18] Linton C Freeman et al. 2002. Centrality in social networks: Conceptual clarification. *Social network: critical concepts in sociology. Londres: Routledge* 1 (2002), 238–263.
[19] Daniel Gibert, Carles Mateu, and Jordi Planes. 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *J. Netw. Comput. Appl.* 153 (2020), 102526. https://doi.org/10.1016/J.JNCA.2019.102526
[20] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. 2019. Using convolutional neural networks for classification of malware represented as images. *J. Comput. Virol. Hacking Tech.* 15, 1 (2019), 15–28. https://doi.org/10.1007/S11416-018-0323-0
[21] Roger Guimera, Stefano Mossa, Adrian Turtschi, and LA Nunes Amaral. 2005. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences* 102, 22 (2005), 7794–7799.
[22] Mehadi Hassen and Philip K. Chan. 2017. Scalable Function Call Graph-based Malware Classification. In *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita (Eds.). ACM, 239–248.
[23] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. 2013. MutantX-S: Scalable Malware Clustering Based on Static Features. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, Andrew Birrell and Emin Gün Sirer (Eds.). USENIX Association, 187–198.
[24] Mugdha Jain, William Andreopoulos, and Mark Stamp. 2020. Convolutional neural networks and extreme learning machines for malware classification. *J. Comput. Virol. Hacking Tech.* 16, 3 (2020), 229–244. https://doi.org/10.1007/s11416-020-00354-y
[25] Sachin Jain and Yogesh Kumar Meena. 2011. Byte level n–gram analysis for malware detection. In *International Conference on Information Processing*. Springer, 51–59.
[26] Tiantian Ji, Binxing Fang, Xiang Cui, Zhongru Wang, Peng Liao, and Shouyou Song. 2023. Framework for understanding intention-unbreakable malware. *Sci. China Inf. Sci.* 66, 4 (2023). https://doi.org/10.1007/S11432-021-3567-Y
[27] Qingshan Jiang, Nancheng Liu, and Wei Zhang. 2013. A feature representation method of social graph for malware detection. In *2013 Fourth Global Congress on Intelligent Systems*. IEEE, 139–143.

Just Accepted

[28] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 625–642. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jordaney

[29] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil D. B. Bruce, Yang Wang, and Farkhund Iqbal. 2018. Malware Classification with Deep Convolutional Neural Networks. In *9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018*. IEEE, 1–5. https://doi.org/10.1109/NTMS.2018.8328749

[30] Kesav Kancherla and Srinivas Mukkamala. 2013. Image visualization based malware detection. In *Proceedings of the 2013 IEEE Symposium on Computational Intelligence in Cyber Security, CICS 2013, IEEE Symposium Series on Computational Intelligence (SSCI), 16-19 April 2013, Singapore*. IEEE, 40–44.

[31] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[32] Joris Kinable and Orestis Kostakis. 2011. Malware classification based on call graph clustering. *J. Comput. Virol.* 7, 4 (2011), 233–245. https://doi.org/10.1007/s11416-011-0151-y

[33] Deguang Kong and Guanhua Yan. 2013. Discriminant malware distance learning on structural information for automated malware classification. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy (Eds.). ACM, 1357–1365.

[34] Marek Krcál, Ondrej Svec, Martin Bálek, and Otakar Jasek. 2018. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HkHrmM1PM

[35] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings To Document Distances. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*, Francis R. Bach and David M. Blei (Eds.). JMLR.org, 957–966. http://proceedings.mlr.press/v37/kusnerb15.html

[36] Young Man Kwon, Jae-Ju An, Myung-Jae Lim, Seongsoo Cho, and Won-Mo Gal. 2020. Malware Classification Using Simhash Encoding and PCA (MCSP). *Symmetry* 12, 5 (2020), 830. https://doi.org/10.3390/sym12050830

[37] Xiaoming Liu, Johan Bollen, Michael L. Nelson, and Herbert Van de Sompel. 2005. Co-authorship networks in the digital library research community. *Inf. Process. Manag.* 41, 6 (2005), 1462–1480.

[38] Joe Security LLC. 2022. Joe Security. https://www.joesecurity.org/.

[39] Yixuan Ma, Shuang Liu, Jiajun Jiang, Guanhong Chen, and Keqiu Li. 2021. A comprehensive study on learning-based PE malware family classification methods. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1314–1325. https://doi.org/10.1145/3468264.3473925

[40] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 141–150. https://doi.org/10.1145/1242572.1242592

[41] Massimo Marchiori and Vito Latora. 2000. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications* 285, 3-4 (2000), 539–546.

[42] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[43] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. IEEE Computer Society, 421–430. https://doi.org/10.1109/ACSAC.2007.21

[44] Robert Moskovitch, Dima Stopel, Clint Feher, Nir Nissim, and Yuval Elovici. 2008. Unknown malcode detection via text categorization and the imbalance problem. In *IEEE International Conference on Intelligence and Security Informatics, ISI 2008, Taipei, Taiwan, June 17-20, 2008, Proceedings*. IEEE, 156–161.

[45] Lakshmanan Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. 2011. Malware images: visualization and automatic classification. In *8th International Symposium on Visualization for Cyber Security, VizSec 2011, Pittsburgh, PA, USA, July 20, 2011*. ACM, 4.

[46] Sang Ni, Quan Qian, and Rui Zhang. 2018. Malware identification using visualization images and deep learning. *Comput. Secur.* 77 (2018), 871–885. https://doi.org/10.1016/j.cose.2018.04.005

[47] nickcano. 2018. PyPackerDetect. https://github.com/cylance/PyPackerDetect.

[48] Yanchen Qiao, Bin Zhang, and Weizhe Zhang. 2020. Malware Classification Method Based on Word Vector of Bytes and Multilayer Perception. In *2020 IEEE International Conference on Communications, ICC 2020, Dublin, Ireland, June 7-11, 2020*. IEEE, 1–6. https://doi.org/10.1109/ICC40277.2020.9149143

Just Accepted

[49] Dima Rabadi and Sin G. Teo. 2020. Advanced Windows Methods on Malware Detection and Classification. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 54–68. https://doi.org/10.1145/3427228.3427242

[50] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. 2018. Malware Detection by Eating a Whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018 (AAAI Technical Report, Vol. WS-18)*. AAAI Press, 268–276. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422

[51] Edmar R. S. De Rezende, Guilherme C. S. Ruppert, Tiago Carvalho, Fabio Ramos, and Paulo L. de Geus. 2017. Malicious Software Classification Using Transfer Learning of ResNet-50 Deep Neural Network. In *16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017, Cancun, Mexico, December 18-21, 2017*, Xuewen Chen, Bo Luo, Feng Luo, Vasile Palade, and M. Arif Wani (Eds.). IEEE, 1011–1014. https://doi.org/10.1109/ICMLA.2017.00-19

[52] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVclass: A Tool for Massive Malware Labeling. In *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9854)*, Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquín García-Alfaro (Eds.). Springer, 230–253.

[53] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. 2018. Machine Learning Aided Static Malware Analysis: A Survey and Tutorial. *CoRR* abs/1808.01201 (2018). arXiv:1808.01201 http://arxiv.org/abs/1808.01201

[54] Guosong Sun and Quan Qian. 2021. Deep Learning and Visualization for Identifying Malware Families. *IEEE Trans. Dependable Secur. Comput.* 18, 1 (2021), 283–295. https://doi.org/10.1109/TDSC.2018.2884928

[55] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Zheng Qin. 2020. IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Comput. Networks* 171 (2020), 107138. https://doi.org/10.1016/J.COMNET.2020.107138

[56] Danish Vasan, Mamoun Alazab, Sobia Wassan, Babak Safaei, and Qin Zheng. 2020. Image-Based malware classification using ensemble of CNN architectures (IMCEC). *Comput. Secur.* 92 (2020), 101748. https://doi.org/10.1016/j.cose.2020.101748

[57] Mayuri Wadkar, Fabio Di Troia, and Mark Stamp. 2020. Detecting malware evolution using support vector machines. *Expert Syst. Appl.* 143 (2020).

[58] Jiaqi Yan, Guanhua Yan, and Dong Jin. 2019. Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 52–63. https://doi.org/10.1109/DSN.2019.00020

[59] FuYong Zhang and Tiezhu Zhao. 2017. Malware Detection and Classification Based on N-Grams Attribute Similarity. In *2017 IEEE International Conference on Computational Science and Engineering, CSE 2017, and IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2017, Guangzhou, China, July 21-24, 2017, Volume 1*. IEEE Computer Society, 793–796.

[60] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 4438–4445. https://doi.org/10.1609/AAAI.V32I1.11782

Just Accepted