Computers & Security 125 (2023) 103060

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

IoT malware classification based on reinterpreted function-call graphs

Chia-Yi Wu[a], Tao Ban[b,*], Shin-Ming Cheng[a,c], Takeshi Takahashi[b], Daisuke Inoue[b]

[a] Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan
[b] National Institute of Information and Communications Technology, Koganei, 184-8795, Tokyo, Japan
[c] Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan

ARTICLE INFO

Article history:
Received 2 September 2022
Revised 11 November 2022
Accepted 10 December 2022
Available online 13 December 2022

Keywords:
Cybersecurity
IoT malware analysis
Machine learning
Static analysis
Graph embedding

ABSTRACT

Various malware and cyberattacks have arisen along with the proliferation of IoT devices. The evolving malware targeting IoT devices calls forth effective and efficient solutions to protect vulnerable IoT devices from being compromised. In this paper, we investigate the feasibility of a state-of-the-art graph embedding method, *graph2vec*, for performing family classification for IoT malware, with promising results reported. To further improve the generalization performance of the classifiers based on *graph2vec*-extracted features, we propose two new mechanisms to improve the quality of feature representation. First, we unify user-defined function calls by reinterpreting the opcode sequences therein to better capture the semantics of the function-call relationship in malware binaries. Then, we integrate literal information into the *graph2vec* embedding of the function call graph to achieve better discriminant ability. To prove the effectiveness of the proposed scheme, we carried out performance comparison on a large-scale dataset containing more than 108K malware binaries collected from seven CPU architectures. The accuracy rates obtained by five widely adopted classifiers on malware family classification are improved by 2%, on average, by adopting the two proposed mechanisms. Specifically, when combined with the proposed approach, the support vector machine classifier obtained an accuracy rate of 98.88% on malware family classification, outperforming known function-call-graph (FCG)-based methods and previous work on static malware analysis.

1. Introduction

While the popularity of Internet of Things (IoT) devices has facilitated digital life, it has also attracted various malware attacks, casting a shadow on digital security. In particular, the source code release of malware programs such as Mirai has led to numerous malware variants, rendering the protection of IoT devices more challenging (Chaganti et al., 2022; Costin and Zaddach, 2018; Galal et al., 2015). A pressing need exists to devise effective and efficient mechanisms to detect and categorize IoT malware to achieve reliable protection for IoT devices (Kuang et al., 2020; Kumar et al., 2022; Wazzan et al., 2021).

Recent research suggests that when combined with static analysis, graph-feature-based approaches can successfully model malware behavior and achieve good prediction performance (Alasmary et al., 2019; Muzaffar et al., 2022). The syntactic and semantic structure of malware binaries can be represented as graphs at different levels, e.g., function call graphs (FCGs) and control flow graphs (CFGs). Representations of malware programs as graphs usually yield complicated structures with overwhelming capacity, which call forth an efficient transformation before it can be taken as input to a learning algorithm (Vinayaka and Jaidhar, 2021). Meanwhile, significant research progress has been made on the so-called graph embedding techniques in the past few years. Graph embedding algorithms (Xu, 2021) can effectively convert high-dimensional sparse graphs into low-dimensional, dense, and continuous vector spaces while preserving the semantics presented in the graph structures. With the learned vector representation, node similarity in the original complex graph space can be easily quantified in the embedded vector space using standard metrics and then can be effectively exploited by subsequent learning algorithms. Nevertheless, a systematic assessment of the feasibility of applying state-of-the-art graph embedding methods for IoT malware analysis has yet to be performed.

In this paper, we propose to apply the state-of-the-art method known as *graph2vec* (Narayanan et al., 2017a) to perform graph embedding for FCGs extracted from malware binaries. We propose two mechanisms to enhance the discriminating information captured by *graph2vec* during graph embedding. First, we unify user-defined function calls by reinterpreting the opcode sequences in the malware binaries. This operation helps reduce the graph's size

* Corresponding author.
E-mail address: bantao@nict.go.jp (T. Ban).

https://doi.org/10.1016/j.cose.2022.103060
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

and enables more precise modeling of the function-call relationship. Second, we realize a new implementation of *graph2vec* that uses function names to identify the vertices in the graph. Integrating literal information into the graph embedding can capture more discriminant information in the data and improve the conditional models' generalization performance.

To evaluate the proposed malware classification scheme, we implement five widely adopted classifiers, namely, random forest (RF) (Ho, 1998), *k*-nearest neighbors (*k*NN) (Hastie et al., 2009), support vector machine (SVM) (Cortes and Vapnik, 1995), multilayer perceptron (MLP) (Haykin, 1999), and logistic regression (LR) (Hosmer and Lemeshow, 2000). We carry out performance evaluation on a large-scale dataset consisting of 108,616 malware samples compiled on seven different CPU architectures. The experimental results show that by adopting the proposed mechanisms, the accuracy rates of the evaluated classifiers on malware family classification are improved by 2% on average. Among the five evaluated classifiers, SVM demonstrates the best performance and reached an accuracy of 98.88% when evaluated by 5-fold stratified cross-validation. It outperforms known FCG-based methods and previous work of IoT malware analysis based on static features.

The main contributions of this paper are summarized as follows.

- We present a feasibility study and performance evaluation of applying graph embedding methods to analyze IoT malware.
- We propose a process to reinterpret UDFs corresponding to identical opcode sequences to enhance the semantics of FCGs.
- We develop a new implementation of *graph2vec* to account for the literal information of function names in the learning process.
- We demonstrate the effectiveness of the proposed scheme using extensive numerical studies on a large-scale benchmark dataset.

The remainder of this paper is organized as follows. Section 2 reviews previous works related to malware analysis. Then, Section 3 introduces the motivation behind our research. Then, Section 4 elaborates the proposed scheme. In addition, Section 5 evaluates the performance of the proposal. Section 6 discusses the limitations of the proposed scheme. Finally, Section 7 concludes the paper.

2. Background and related work

In this section, we introduce the background of IoT malware and review previous work on IoT malware detection and classification.

2.1. IoT malware

IoT devices provide considerable convenience to today's digital life. However, due to the resource constraint feature and the lack of user security awareness, IoT devices have become the most captivating target of malware attacks. Among the numerous attacks that harm IoT devices, IoT botnet is the most notorious. An IoT botnet is a network of hijacked IoT devices infected by a botnet tool that allows hackers to take control. Botnets can be abused to send out spam or conduct attacks such as distributed denial of service (DDoS) attacks.

The most famous attack of IoT malware was the DDoS attack caused by Mirai (Antonakakis et al., 2017; Marzano et al., 2018) in 2016, which targeted systems operated by the domain name system (DNS) provider Dyn. The provider was hit on 21 October and remained under sustained assault for most of the day, bringing down sites including Twitter, Reddit, GitHub, Amazon.com, Netflix, and many others in Europe and the US (Guardian, 2022). Another well-known malware family is Hajime (Herwig et al., 2019), which searches for systems to infect by scanning the Internet for systems running telnet on port 23/TCP and then tries to log in with default accounts and passwords. Once logged in, it takes control of the device and uses peer-to-peer connections for command and control. Other well-known IoT malware families include Dofloo and Xorddos, which have also launched many large-scale DDoS attacks in recent years.

2.2. Previous work on malware analysis

We surveyed related research on malware detection and family classification to devise an effective and efficient scheme to protect IoT devices from malware infection. Note that most reviewed approaches were originally designed for malware protection on other platforms, such as Windows or Android. While some ideas can be generalized to IoT malware analysis, others are not directly applicable because of the CPU architecture diversity and resource constraints enjoyed by IoT devices. For example, dynamic analysis – executing binary programs in a sandbox environment to monitor their run-time behavior – has been the most effective means for Windows/Android malware analysis. However, in regard to IoT malware, dynamic analysis tends to suffer difficulties in performing consistent analysis on varying CPU architectures. Moreover, resource constraints on IoT devices tend to render on-device dynamic analysis infeasible. Therefore, we focus on surveying the static-analysis-based research on malware analysis.

Static analysis consists of the analysis of a program without executing it. According to the characterizing features used in the study, the related work can be classified as binary-based, opcode-based, graph-based, API-based, and others. Table 1 summarizes the related work focusing on efficient static analysis.

2.2.1. Binary-based methods

Executable files, a.k.a. binaries, are binary files containing machine code for the computer to execute. As the form in which malware is distributed, a binary carries all necessary information that enables the program's execution in the target environment. For malware analysis, binary files are often treated as sequences of bytes.

Raff et al. (2018) proposed the MalConv model, a convolutional neural network (CNN) that takes the sequences of bytes in binaries as a whole for portable executable (PE) malware detection. With a model trained over 40K training samples, they reported 94.0% accuracy on a testing set containing more than 77K Windows PEs.

Su et al. (2018) proposed extracting one-channel grayscale images that are converted from binaries and then utilizing a lightweight CNN to classify IoT malware. They reported that the proposed system could achieve 94.0% accuracy for detecting DDoS malware from benignware and 81.8% accuracy for classifying benignware and two major malware families.

Wan et al. (2020) devised an *N*-gram-based method to explore the discriminating information stored in the byte sequences at the entry points of executable programs. They reported 99.96% accuracy for malware detection and 98.47% accuracy for malware family classification on a dataset consisting of 111K benignware and 111K malware samples.

The related work listed above takes in byte sequences as input features for the learning models. It usually results in very fast feature extraction, which can, in turn, enable fast malware protection. On the other hand, the byte sequences in the malware binaries are subject to simple obfuscation techniques such as dead-code insertion and instruction substitution (You and Yim, 2010). They may yield degenerated generalization performance for evolving IoT malware. Compared with binary features, the FCG features adopted in

Table 1
Previous work on malware analysis based on static analysis.

Author	Platform		Dataset		Task		Accuracy(%)	Feature type		Algorithm(s)	
			Benignware	Malware							
Raff et al. (2018)		Windows	57,349	60,000		Detection	94.0		Binary		Deep learning (CNN)
Su et al. (2018)	✓	IoT	122	243	✓	Classification	81.8		Binary		Deep learning (CNN)
			122	243		Detection	94.0		Binary		Deep learning (CNN)
Wan et al. (2020)	✓	IoT	111K	111K	✓	Classification	98.47		Byte sequence	✓	SVM
			111K	111K		Detection	99.96		Byte sequence	✓	SVM
Kang et al. (2016)		Android	-	1260	✓	Classification	98 %(F1-measure)		Opcode	✓	SVM
			1260	1260		Detection	98% (F1-measure)		Opcode	✓	SVM
Ban et al. (2019)	✓	IoT	-	9085	✓	Classification	Up to 100.00		Opcode	✓	SVM, *k*NN
Gülmez and Sogukpinar (2021)		Windows	7500	7500		Detection	99.0		Opcode graph	✓	RF
Alasmary et al. (2019)	✓	IoT	2,999	2962	✓	Classification	99.32		Control flow graph		Deep learning(CNN)
			2,999	2962		Detection	99.66		Control flow graph		Deep learning(CNN)
Nguyen et al. (2020)	✓	IoT	6,165	3845		Detection	98.7		Printable string graph		Deep learning(CNN)
Ou and Xu (2022)		Android	82,010	50,123		Detection	97.71% (F1-measure)	✓	Function fall graph	✓	RF
Xiao et al. (2020)		Android	880	3520	✓	Classification	95.27	✓	Function call graph	✓	RF
			880	3520		Detection	99.75	✓	Function call graph	✓	RF
Zhang et al. (2020)		Windows	-	10,260	✓	Classification	99.57	✓	Function call graph	✓	RF
Ban et al. (2016)		Android	52,251	26,398		Detection	94.09		API call, permission, app category	✓	SVM
Onwuzurike et al. (2017)		Android	8.5K	35.5K		Detection	99% (F1-measure)		API call	✓	RF, *k*NN, SVM
Shahzad and Farooq (2012)	✓	IoT	734	709	✓	Classification	99.8		ELF header		Decision tree
Lee et al. (2020)	✓	IoT	-	120K	✓	Classification	98.36		Printable string	✓	SVM
Wu et al. (2021)	✓	IoT	89K	108K		Detection	99.71	✓	Function call graph	✓	SVM
Proposed approach	✓	IoT	-	108K	✓	Classification	98.88	✓	Function call graph	✓	RF, *k*NN, SVM, MLP, LR

Note: A leading checkmark in columns, including platform, task, and feature type, indicates that the corresponding related work has the same setting as the proposed approach. In the algorithm column, the leading marker indicates that the corresponding method is implemented in the experiments.

this paper carry high-level behavioral information about the malware binary and are considered more robust against code obfuscation (Naseer et al., 2021).

2.2.2. *Opcode-based methods*

An opcode, abbreviated from operation code, is the portion of a machine language instruction specifying the operation to be performed. Opcode sequences, as the output of many reverse engineering tools, carry fine-grained information about the execution logic of the program and have been widely adopted as the basis for further analysis in related works.

Kang et al. (2016) proposed a method based on *N*-grams over opcode sequences for Android malware detection and family classification. Their approach supports automated feature coverage and eliminates the need for domain knowledge to define the discriminating features. On a dataset of 2520 samples, SVM classifiers yielded a maximum F1-measure of 98% in both malware detection and malware classification with $N = 3$ and $N = 4$, respectively.

Ban et al. (2019) proposed a multimodal analytical approach to characterize IoT malware. They showed that opcode sequences obtained from static analysis and API sequences obtained by dynamic analysis provide sufficient discriminant information to classify IoT malware with near-optimal accuracy. Their method achieved detection accuracy of up to 100% for CPU-specific analysis on a dataset containing 9085 IoT malware samples collected from a honeypot system.

Gülmez and Sogukpinar (2021) proposed a method based on the opcode graphs of executables. They extracted the node degrees of self-connecting subgraphs of an opcode graph as the feature representations of the sample. Their method achieved a detection accuracy of up to 98% on a dataset of PE files containing 15K packed and unpacked samples.

According to the survey in Naseer et al. (2021), the opcode is the most widely adopted feature type for static analysis because of its low cost and strong discriminating nature. However, due to the extremely fine granularity of opcode, a graph representation yielded by opcode may show a size rendering fast analysis impossible. In the proposed scheme, the opcode sequences are organized as function calls – blocks of opcodes that realize particular functionalities. Integrating opcode as function calls can not only significantly reduce the size for graph representation but also result in improved modeling of the malware behavior at a higher level.

2.2.3. *Graph-based methods*

The malicious behavior of malware can be characterized by certain components in some representative structures obtained from malware binaries that reflect the execution logic of the program. In the literature, CFGs and FCGs are among the most widely adopted representative structures to serve this purpose.

Alasmary et al. (2019) proposed deploying deep learning to the structural features extracted from CFGs to train malware detection models. They reported 99.66% accuracy for malware detection and 99.32% for family classification on a dataset comprising 6K IoT samples. Structural features, including the number of nodes and edges, density, centrality, and shortest path, were inputs to the CNN model.

Nguyen et al. (2020) proposed extracting information from printable strings such as IP addresses, URLs, usernames, and passwords presented in an FCG and generated a printable string information graph (PSI graph). They then used a CNN to analyze the PSI graph for detecting malware. They reported 98.7% accuracy on a dataset of 10,010 ELF IoT samples.

Ou and Xu (2022) proposed a method called S3Feature, which extends a function call graph by tagging sensitive nodes based on sensitivity evaluation. They reported an F1-measure of 97.71% for malware detection when S3Feature is combined with other features.

Xiao et al. (2020) proposed a graph re-partition algorithm based on an *N*-order subgraph to capture appropriate vibration behavior. They applied an improved term frequency-inverse document frequency measure and information gain to learn the significant *N*-order subgraphs to represent crucial malware behavior. They reported 99.75% accuracy for malware detection and 95.27% accuracy for malware family classification on a dataset of 4400 samples.

Zhang et al. (2020) proposed combining features obtained from FCG vectorization and other nongraph features for better generalization performance. They reported 99.57% accuracy on a Windows malware dataset of 10,260 malware samples provided by Microsoft at Kaggle.

Built upon the FCG representation of IoT malware samples, the proposed approach is closely related to the related work. The major difference from previous work resides in the following facts: By extracting the vector representations of the semantics in the graph through graph embedding, we can capture essential discriminating information from malware samples. The proposed methods can be applied to most graph representations that can model the malware behavior in syntactic or semantic means.

2.2.4. API-based methods

Application programming interface (API), a software interface that offers a service to other pieces of software, is used in high-level programming to invoke system calls at the low level. It provides a cost-effective and comprehensive middle layer to model the attack behavior of malware samples. Many successful approaches to identifying and extracting malicious behavior are reported in the literature, ranging from techniques based on API call frequency analysis (Natani and Vidyarthi, 2013) to more sophisticated detection schemes based on exploiting the probabilities of transitioning from API invocations (D'Angelo et al., 2021).

Ban et al. (2016) explored the potential of multimodal features to enhance the detection accuracy of Android malware. The examined features included permissions, API calls, and meta-features such as the category information and application package (APK) descriptions. Using a linear SVM classifier, they reported an accuracy of 94.09% on an Android dataset composed of 78,649 apps using the API call and app category features.

The MaMaDroid detector introduced in Onwuzurike et al. (2017) abstracted the API calls performed by an Android app to their package names or families and built a model from the sequences obtained from the call-graph Markov chains. Modeling the malware behavior using the transition probabilities between API calls improved robustness against evasion techniques. MaMaDroid reported an effective detection rate (up to 99% F1-measure) on a dataset of 8.5K benign Android apps and 35.5K malicious Android apps collected over six years. It maintained its detection capabilities for long periods.

As reported by the literature survey in Naseer et al. (2021), as far as dynamic analyses are concerned, API sequences captured during the program's execution time are reported as the most significant data type that facilitates malware analysis. However, API calls are not as preferable as opcode and derived data in regard to static analysis. This is partially because capturing activity indicators finer grained than API, e.g., opcodes, are extremely expensive in dynamic analysis but comparatively affordable in static analysis. As a special type of high-level function call, APIs together with low level function calls such as system calls and UDFs, constitute to the graph representation of FCGs analyzed in the proposed approach.

2.2.5. Other static features

Many other types of information can be obtained from static analysis and serve as behavioral indicators of malware.

Shahzad and Farooq (2012) proposed composing a feature set from the header of Linux executable and linkable format (ELF) files. Using information gain as preprocessing filters, they selected 383 attributes with high classification potential and performed the data evaluation. They reported that the classical rule-based machine learning algorithms and bio-inspired classifiers can reach more than 99% detection accuracy on a Linux ELF dataset consisting of 1443 samples.

Lee et al. (2020) proposed a method based on printable strings extracted from the bodies of binaries. They reported that an SVM classifier affiliated with feature selection could yield an accuracy of 98.36% on a large-scale dataset consisting of 120K IoT samples. They also demonstrated the effectiveness of printable strings as an effective feature for cross-platform IoT malware classification.

Most of the related work surveyed in this section creates vector representations for malware to be further analyzed by learning algorithms. These vector representations can be combined with the embedding vectors yielded by the proposed scheme for possible performance gain when affordable. In our experiment, we integrated a subset of the structural features introduced in Alasmary et al. (2019) as a reinforcement to the embedding vector yielded by the proposed approach for better malware classification performance.

3. Motivation

In this section, we introduce the motivation behind our research focusing on selecting the most appropriate graph representation for malware and improving the semantics capturing capability for *graph2vec*.

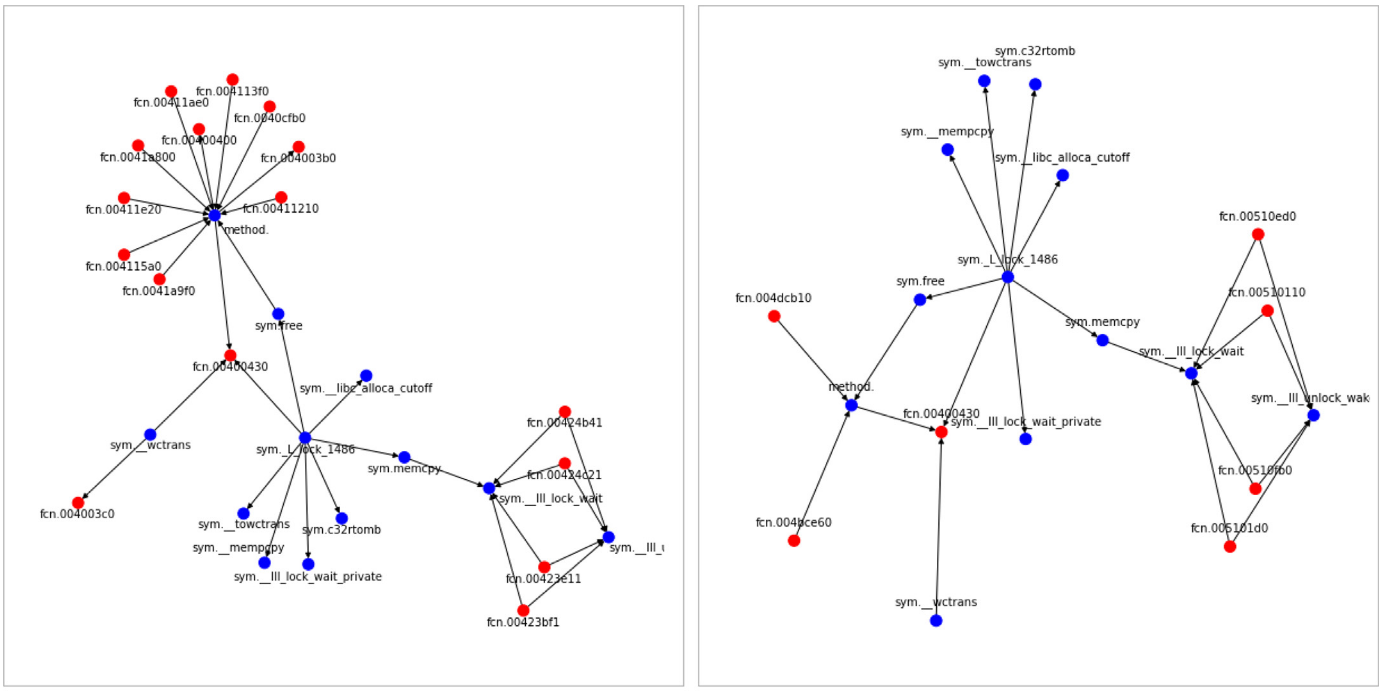
3.1. Graph representation for malware samples

Existing graph-based malware analysis approaches typically involve a step to select a suitable graphical representation of the binaries as the basis for further analysis. CFGs (Ngo et al., 2020; Qiang et al., 2022), FCGs (Kawasoe et al., 2021; Li et al., 2021), and PSI graphs (Nguyen et al., 2020) are among the most widely adopted graphical representations.

For an accurate representation of the flow inside a program unit, CFGs capture the interaction of low-level operations during the program's execution. A drawback of CFGs is that even binaries with moderate size yield extraordinarily large graphs, rendering the processing and analysis very time-consuming. On the other hand, FCGs record only calling relationships between subroutines during program execution. By ignoring the low-level operations, FCGs can yield a more concise graphical representation while maintaining the essential information about the execution logic of the program.

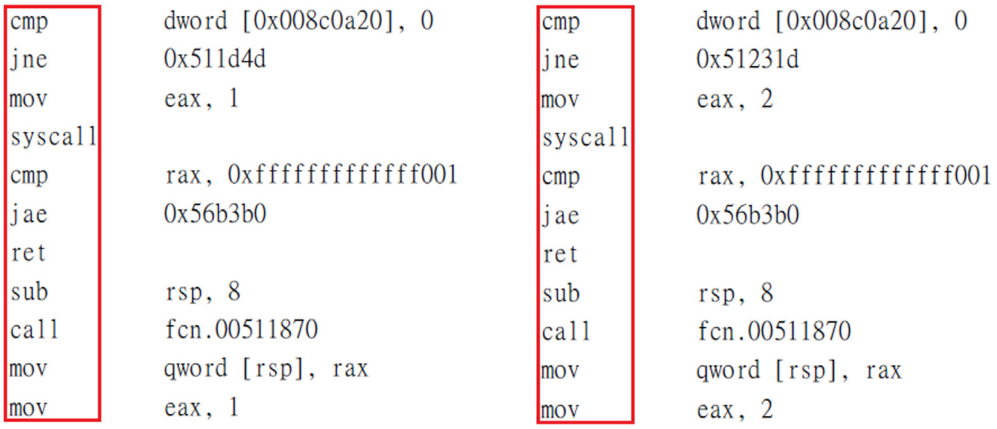
PSI graphs extract information from printable strings such as IP addresses, URLs, usernames, and passwords presented in an FCG. As most of these printable strings are composed at the programming phase, PSI graphs can serve as strong descriptors for binaries compiled from the same source. Compared with FCGs, PSI graphs require additional resources and time to extract and reorganize printable strings. On the other hand, information other than printable strings in FCGs is ignored by PSI graphs, which may result in unwanted information loss.

For the above given reasons, an FCG is chosen as our study's graphical representation for IoT malware. Fig. 2 shows an example of an FCG extracted from a malware program. In the graph, the vertices represent functions, and the edges correspond to the caller-callee relationship between them. According to Li et al. (2021), by representing the flow at the pertinent abstract level of function calls, FCGs can avoid obfuscation at the instruction level and byte level. Moreover, in terms of IoT malware,



(a) An FCG with original UDFs (b) The FCG after UDF reinterpretation

Fig. 1. Function-call graphs (partial) of a Mirai sample before and after the interpretation of user-defined functions.


```
cmp       dword [0x008c0a20], 0
jne       0x511d4d
mov       eax, 1
syscall
cmp       rax, 0xfffffffffffff001
jae       0x56b3b0
ret
sub       rsp, 8
call      fcn.00511870
mov       qword [rsp], rax
mov       eax, 1
```

(a) User-defined function example 1

```
cmp       dword [0x008c0a20], 0
jne       0x51231d
mov       eax, 2
syscall
cmp       rax, 0xfffffffffffff001
jae       0x56b3b0
ret
sub       rsp, 8
call      fcn.00511870
mov       qword [rsp], rax
mov       eax, 2
```

(b) User-defined function example 2

Fig. 2. Two similar user-defined functions in a Mirai sample.

FCGs can capture CPU-independent semantics of the same malware. When a source file is compiled for different CPU architectures, the obtained binary files will be completely different because of the distinct instruction sets, but the FCGs may track their similarity at the function-call level. This information is beneficial for malware analysis across different CPU architectures.

3.2. Improvement to *graph2vec*

We encountered two problems that may negatively impact the analysis when applying *graph2vec* (Narayanan et al., 2017b) to extract graphic embedding features from FCGs for IoT malware.

User-defined functions (UDFs) are the first problem for FCGs. While API and system calls are essential components of an FCG, UDFs also contribute. Determined by the convention of the compiler, UDFs are often assigned temporary identifiers for each running instance on an ad hoc basis. Take the two UDFs shown in Fig. 1 as an example. The same UDF is assigned two different identifiers based on memory location. Duplicated UDFs with distinguishing names induce unnecessary computation costs and lead to ill-formed FCGs with erroneous semantic information. To solve this problem, we propose a reinterpretation process for UDFs: UDFs are unified by investigating the opcode sequences representing the flow of the detailed operation executed therein. Then, the UDFs corresponding to the same opcode sequence are assigned an identical universal identifier (UUID) so that the reuse of the same UDFs will be accounted for in the analysis. Unifying the UDF names by reinterpreting opcode sequences therein is expected to reduce the FCG size and improve the prediction accuracy.

The second problem is associated with the implementation of *graph2vec*. Originally, *graph2vec* was designed to handle abstract graphs so that no identifying information is stored in the graph vertices. Therefore, common implementations use the degree, i.e., the number of edges that are incident to the vertex, as identifiers

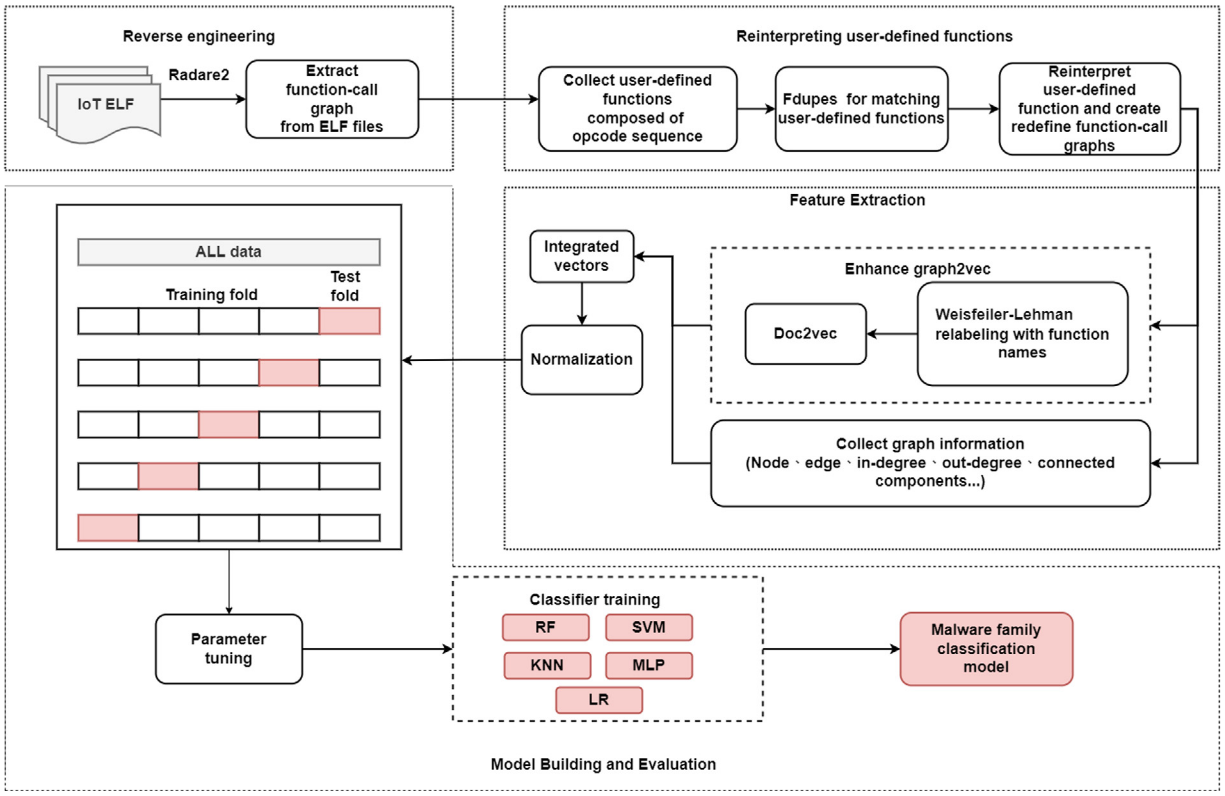

Reverse engineering
IoT ELF
Radare2
Extract function-call graph from ELF files
Reinterpreting user-defined functions
Collect user-defined functions composed of opcode sequence
Fdupes for matching user-defined functions
Reinterpret user-defined function and create redefine function-call graphs
Feature Extraction
Integrated vectors
Normalization
Enhance graph2vec
Doc2vec
Weisfeiler-Lehman relabeling with function names
Collect graph information (Node、edge、in-degree、out-degree、connected components...)
ALL data
Training fold
Test fold
Parameter tuning
Classifier training
RF
SVM
KNN
MLP
LR
Malware family classification model
Model Building and Evaluation

Fig. 3. Overview of the proposed IoT malware family classification scheme

in the graph. Such implementations render a significant loss of information in regard to the FCGs. In this paper, we implement a new version of *graph2vec* that uses function names to identify the vertices in the graph. Preserving semantics of function-call relationships is expected to capture more discriminant information in the data and improve the prediction performance.

4. Methodology

In this section, we elaborate the proposed approach for IoT malware analysis based on reinterpreted FCG obtained from static analysis. As shown in Fig. 3, the investigation is pursued in four steps: reverse engineering, reinterpreting UDFs, feature extraction, and model building and evaluation. In the reverse engineering step, we use *radare*2 (Radare2) to perform static analysis on the input binaries and create FCGs. In the UDF reinterpretation step, we match user functions by their opcode sequences and assign UUIDs to unique UDFs accordingly. In the feature extraction step, we perform feature extraction on the reinterpreted FCGs. Two types of features, namely, graph embedding features from an enhanced *graph2vec* (Wu et al., 2021) and structural features of graphs, are combined for better generalization performance. In the model building and evaluation step, we choose widely adopted classifiers to formulate prediction models for malware family classification. We use 5-fold stratified cross-validation to evaluate the performance of the models.

4.1. Reverse engineering

As Linux is the most popular operating system installed on IoT devices, most malware arrives at victim devices in the form of executable and linkable format (ELF) files. To disassemble the ELF binaries, we adopt *radare*2, a complete framework for reverse engineering. After the static analysis, *radare*2 outputs a CFG, which can

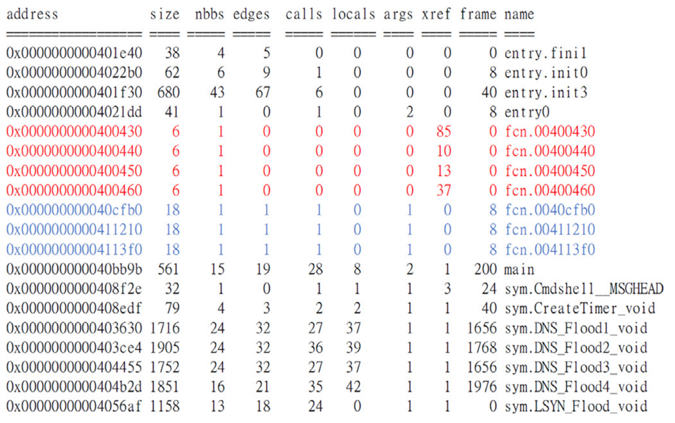

```
address              size  nbbs  edges  calls  locals  args  xref  frame  name
===================  ====  ====  =====  =====  ======  ====  ====  =====  ====
0x0000000000401e40     38     4      5      0       0     0     0      0  entry.fini1
0x00000000004022b0     62     6      9      1       0     0     0      8  entry.init0
0x0000000000401f30    680    43     67      6       0     0     0     40  entry.init3
0x00000000004021dd     41     1      0      1       0     2     0      8  entry0
0x0000000000400430      6     1      0      0       0     0    85      0  fcn.00400430
0x0000000000400440      6     1      0      0       0     0    10      0  fcn.00400440
0x0000000000400450      6     1      0      0       0     0    13      0  fcn.00400450
0x0000000000400460      6     1      0      0       0     0    37      0  fcn.00400460
0x000000000040cfb0     18     1      1      1       0     1     0      8  fcn.0040cfb0
0x0000000000411210     18     1      1      1       0     1     0      8  fcn.00411210
0x00000000004113f0     18     1      1      1       0     1     0      8  fcn.004113f0
0x000000000040bb9b    561    15     19     28       8     2     1    200  main
0x0000000000408f2e     32     1      0      1       1     1     3     24  sym.Cmdshell__MSGHEAD
0x0000000000408edf     79     4      3      2       2     1     1     40  sym.CreateTimer_void
0x0000000000403630   1716    24     32     27      37     1     1   1656  sym.DNS_Flood1_void
0x0000000000403ce4   1905    24     32     36      39     1     1   1768  sym.DNS_Flood2_void
0x0000000000404455   1752    24     32     27      37     1     1   1656  sym.DNS_Flood3_void
0x0000000000404b2d   1851    16     21     35      42     1     1   1976  sym.DNS_Flood4_void
0x00000000004056af   1158    13     18     24       0     1     1      0  sym.LSYN_Flood_void
```

Fig. 4. A partial list of function calls extracted from a Mirai sample.

be interpreted as an FCG representing the calling relationship between subroutines in the input binary. See Fig. 2(a) for an example of a part of the FCG returned by *radare*2.

4.2. Reinterpreting user-defined functions

During the analysis, we found that UDFs constitute a substantial portion of the vertices in the FCGs. Fig. 4 shows a fragmented snapshot of the subroutines found in a Mirai sample. Based on the convention of the compiler, UDFs are often assigned identifiers based on the address where they are stored. Subroutine names starting with “fcn” correspond to the UDFs found in the binary. These UDFs form groups (printed with colored fonts in the figure for better readability) that share strong similarities regarding numerical indicators such as size and arguments. A closer look into the opcode of the UDFs in the same group reveals that they are

Table 2
Comparison of the size of FCGs before and after UDF reinterpretation.

		Mirai	Tsunami	Malware Dofloo	Family Bashlite	Xorddos	Android	Average
# Avg. vertices	Before	163.19	249.95	315.90	241.37	1153.21	468.27	431.98
	After	126.85	238.30	143.90	235.57	999.30	302.09	341.00
	Reduction (%)	22.27	4.66	54.45	2.40	13.35	35.49	21.06
# Avg. edges	Before	418.74	616.98	671.70	554.36	3012.21	1236.18	1085.03
	After	313.98	591.86	353.70	540.47	2591.11	738.12	854.87
	Reduction (%)	25.02	4.07	47.34	2.51	13.98	40.29	21.21

associated with the same opcode sequence. Therefore, the grouped UDFs are identical subroutines but are assigned unique names by the compiler.

Fig. 1 shows snapshots of two of these subroutines in the red group of Fig. 4. The opcode sequences highlighted in the red text boxes are the same for the two subroutines. In the example, except for the parameters after the command "jne" – equivalent to the "jump" command on common CPU architectures – and "mov", the parameters for the commands match exactly.

This discordance in naming UDFs negatively impacts algorithms that take function names as discriminating attributes for malware analysis. We perform the following reinterpretation procedure for UDF names to solve the problem. First, we collect all the UDFs in the dataset to form a set. Then, we obtain the opcode sequence for each of the UDFs. In this step, arguments of the opcodes are ignored not only to cover the special case for jump-like commands but also to support free parameters in function calls. Then, using the opcode sequences as signatures, each of the unique UDFs in the set is assigned a UUID. Finally, the names of the UDFs in the dataset are replaced by their UUIDs in the later analysis.

This reinterpretation procedure to treat UDFs as the same subroutine if they perform the same operation in the program can benefit the analysis in the following aspects. First, it can solve the ambiguous naming issue caused by the compiler. Second, it can unify the same functions occasionally assigned different names in the source code. Then, it can result in an improved representation of semantics in the FCG, enhancing the generalization performance. Finally, it can lead to a reduction in the scale of the graph, resulting in improved learning and prediction efficiency. Take the UDFs in Fig. 2 as an example. The FCG in (b) obtained after reinterpretation is much simpler than the FCG before reinterpretation in (a).

Table 2 compares the FCGs built from the dataset introduced in Section IV-A before and after UDF reinterpretation. The table shows that the reduction in FCG size achieved via UDF reinterpretation is subject to variation according to the malware family. A maximum of a 54.45% reduction in vertex numbers and 47.34% reduction in edge numbers is achieved for Dofloo malware. The minimum of a 2.40% reduction in vertex numbers and 2.51% reduction in edge numbers is achieved for Bashlite malware. The macro-average over all examined malware families amounts to a 21.06% reduction in vertex number and 21.21% reduction in edge number. The experiment section will further investigate the impact of UDF reinterpretation on the effectiveness and efficiency of subsequent analysis.

4.3. Feature extraction

The feature extraction step takes the reinterpreted FCGs as input and returns vector representations of the graph as results. Following the idea in Wu et al. (2021), we create two types of features from the graphs, namely, structural features and graph embedding features, to facilitate further analysis.

4.3.1. Graph structure features

Graph properties that capture the structural information of an FCG can serve as characterizing features for the corresponding binary. Features defined on nodes and edges of the graph are easy to obtain and are inherently obtained in vector form. We adopt the features suggested by Alasmary et al. (2019).

Let an FCG, denoted as $G = \{V, E\}$, be a structure amounting to a set of vertices, V, connected by a set of directed edges, E. Each vertex, $v_i \in V$, represents a function found in the binary. Each directed edge running from function v_i to function v_j, denoted as $e_{i,j}$, indicates that v_i calls v_j. Based on the above definition, we compute the following structural properties for all input FCGs.

Definition 1. (**Numbers of vertices and edges**) The number of nodes and edges, denoted as $v = |V|$ and $e = |E|$, respectively, are general characteristics used to describe the scale of an FCG.

Definition 2. (**Number of connected components**) A connected component (CC) of graph G, denoted as S, is a subgraph in which the vertices are connected by the edges in E. The number of CC is the cardinality of a set that contains all CCs of G, i.e., $s = |\{S_i\}|$.

Definition 3. (**Density**) The density, d, of a directed graph G is defined as the closeness of all its edges to the number of edges for a fully connected graph with the same vertex set. Formally,

$$d(G) = \frac{e}{(v-1)^2}. \quad (1)$$

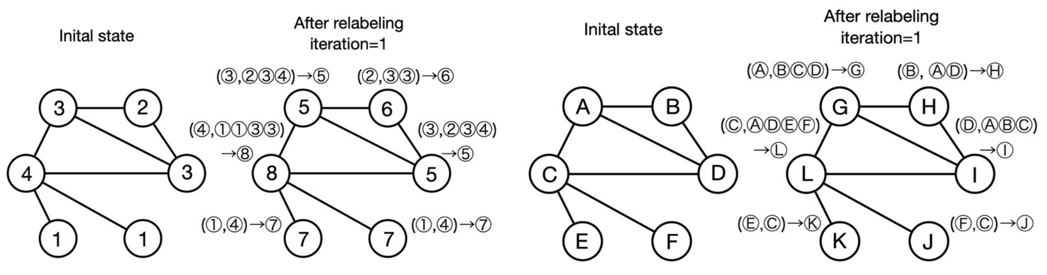
4.3.2. Graph embedding features

The conventional graph properties listed above provide consolidated structural information of an FCG. On the other hand, the semantics in the graph, i.e., the call relationships between subroutines, are not reflected in these features. We resort to a graph embedding method, namely, *graph2vec*, to explore the semantics in the graph.

Graph2vec, devised as a neural embedding framework to learn data-driven distributed representations of arbitrary-sized graphs, enjoys greater scalability, computing adaptability, and effectiveness than conventional graph-based algorithms. Because its embeddings are learned in an unsupervised and task-agnostic manner, *graph2vec* can provide vector representations of graphs for downstream tasks such as data clustering and pattern analysis. The effectiveness and efficiency of *graph2vec* for IoT malware analysis is investigated in this study.

Graph2vec is inspired by *doc2vec* (Le and Mikolov, 2014), which uses a specialized *skipgram* model to learn representations of word sequences of arbitrary length as vectors of a predefined dimension. It then extends document embedding models to obtain graph embeddings. Analogous to *doc2vec*, *graph2vec* treats graphs as documents composed of rooted subgraphs[1], which, in turn, are treated as words. As *graph2vec* was originally designed to treat abstract graphs that do not carry identifying information on the vertices, available implementations use vertex degree – the number of

[1] A rooted subgraph at vertex v_i originates from v_i and encompasses its neighborhood of a certain order.



(a) Initialization and relabeling in *graph2vec*

(b) Initialization and relabeling in enhanced *graph2vec*

Fig. 5. Enhancing *graph2vec* by integrating literal information as vertex labels.

edges that are incident to a vertex – to differentiate different vertices. Such an implementation leads to an unexpected loss of information with respect to FCGs. Fig. 5(a) shows an example of a graph where the vertex degree (numbers on the vertices) is used as the identifier of the vertices. The left figure shows the initial states of the graph when the vertex degree is assigned to the vertices as the initial label. To obtain the order-1 rooted subgraph of a vertex, *graph2vec* performs a breadth-first search starting from the vertex and records the neighbors it has traversed. Take vertex ③ at the top-left corner as an example. Its neighborhood consists of three vertices, (②,③,④), sorted in ascending order. Then, the label of the vertex and its listed neighbors are used as the new label of the order-1 subgraph. The right figure shows the result of order-1 subgraph generation, where the unique labels are replaced by labeling numbers that have not been used for label compression, e.g., a vertex with label (③,②③④) is relabeled ⑤.

For an FCG, the function names of the subroutines carry essential behavioral information of the binary. The above implementation could result in a significant loss of semantics information from the FCG. In our proposed implementation, we use literal information of the subroutines, namely, function names, to label the vertices in the FCG. Fig. 5(b) shows how *graph2vec* is performed on the same FCG as in Fig. 5(a) but with the new labeling scheme. To clarify the difference, we use circled letters to indicate the function names serving as vertex labels on the FCG. The left figure shows the initial states of the graph when function names are given to the vertices as initial labels. Note that vertices Ⓐ, Ⓒ, and Ⓓ are assigned different labels, although they have the same degree of 3. After the first relabeling iteration, each vertex is assigned a new label that encodes the identifying information at the vertex and the function-call relationship within its order-1 neighborhood. Note that the two vertices labeled ⑤ in Fig. 5(a) are now assigned two different labels, Ⓖ and Ⓘ, as they carry different semantic information.

Graph2vec repeats the above relabeling process for m iterations to obtain the order-m rooted subgraphs around each vertex. Here, m determines how many consecutive subroutines are taken as the context for each vertex. As a very large m value may result in overfitting, we set $m = 2$ in the experiments following the recommendation in Narayanan et al. (2017b). Then, *graph2vec* takes the set of all rooted subgraphs as its vocabulary and follows the *doc2vec* skipgram training process to learn a D-dimensional vector representation of each graph in the dataset. Finally, the graph embedding and structural features are combined to form a $(D+4)$-dimensional vector as input for the subsequent classifiers.

4.4. Classification methods

The vector representations yielded by *graph2vec* can facilitate most available machine learning methods. In this section, we select five widely adopted algorithms that can handle large-scale classification tasks efficiently: RF, *k*NN, SVM, MLP, and LR. We construct predictive models for malware family classification. The selected classifiers are briefly defined as follows.

- RF is an integrated learning method for classification or regression that operates by constructing a large number of decision trees during training. For classification tasks, the output of the RF is the class chosen by the majority of trees.
- *k*NN is a learning-by-example approach that learns relevant features via local approximation. A binary is classified by a majority vote of its k nearest neighbors. The results of *k*NN indicate how much discriminant information can be captured in the vector representation.
- SVM is a supervised learning model that finds a hyperplane with maximized margin to distinguish samples from two classes. The SVM meets the needs for the malware detection task where samples are divided into two categories, i.e., benign and malicious. For the multi-class problem, as in malware family classification, we follow the one-against-all convention. For an M-class problem, we first construct M binary SVM classifiers, each of which separates one class from the rest. Then, we decide the predicted label by majority voting of all the classifiers.
- MLP is a class of feed forward artificial neural networks trained using a supervised learning technique called back-propagation. MLP is good for differentiating data that are not linearly separable.
- LR is one of the most popular machine learning algorithms for binary classification because of its simplicity and good performance on a wide range of problems. LR models the probability an event occurring by viewing the logarithm of the odds for the event as a linear combination of one or more independent variables. LR can be generalized to multinomial logistic regression for multi-class classification (Greene, 2012).

5. Experiments

We evaluate the performance of the proposed approach combined with the selected classifiers on malware family classification to predict the category of malware binaries into known families. Our experiments are based on scikit-learn (Pedregosa et al., 2011) implemented with Python 3.7. All the experiments are implemented on a desktop PC with Ubuntu 16.04 LTS with the following specifications: x86 64 Intel(R) Core(TM) i7-7820X CPU @3.60 GHz, 8 core, 128 GB DDR4 Memory.

5.1. Evaluation dataset

As shown in Table 3, we collected 108,616 malware binaries from VirusTotal and labeled them with malware family names based on the majority voting of the detection reports of major

Table 3
Sample distribution among different categories and CPU architectures.

	ARM	MIPS	X86	SPARC	X86-64	PPC	UNKNOWN	Total
Mirai	19,537	10,224	7824	5048	1083	4859	4815	53,390
Tsunami	424	375	1079	62	205	106	214	2465
Dofloo	958	107	213	0	37	0	0	1315
Bashlite	13,466	8731	8240	3163	3839	3465	5136	46,040
Xorddos	2	0	485	0	5	0	0	492
Android	3061	14	1425	0	413	1	0	4914
Total	37,448	19,451	19,266	8273	5582	8431	10,165	108,616

anti-virus vendors. These samples belong to six malware families, namely, Mirai, Tsunami (Kaiten), Dofloo, Bashlite (Gafgyt), Xorddos, and Android. VirusTotal reported that these malware samples were collected from a variety of CPU architectures, including X86, MIPS, ARM, SPARC, X86-64, and PowerPC (PPC). Samples without CPU architecture information are assigned to an UNKNOWN group.

5.2. Visualization

To understand the data distribution, we use uniform manifold approximation and projection (UMAP) (McInnes et al., 2018) to visualize the data in 2D embedding space. UMAP is a nonparametric graph-based dimensionality reduction algorithm that consists of two steps: (1) calculating the graph representation of the dataset and (2) optimizing the low-dimensional embedding of the graph via stochastic gradient descent. This approach is computationally efficient and can handle large-scale high-dimensional datasets. Fig. 6 shows the 2D visualization results for a subset of 1,200 malware samples with 512 dimensions. Malware exhibits good separability across families in the 2D embedding space yielded by UMAP, which implies good prediction performance in malware family classification using the same feature representation.

5.3. Parameter tuning

The generalization performance of machine learning algorithms relies heavily on the hyperparameters used to train the models. Parameter tuning plays an important role in constructing models with robust prediction performance. For the selected algorithms, we performed a grid search on the parameters using 5-fold stratified cross-validation on the training set of each run. The parameter setting that yielded optimal cross-validation performance was used to train a prediction model using the full training set.

Fig. 7 shows an example of the parameter tuning procedure on dimension D of the FCG features. The blue line depicts the highest cross-validation accuracy obtained by SVM with varying D parameters; the red line represents the required time for 5-fold cross-validation to be completed. The figure shows that the cross-validation accuracy increases with D from 128 to 512 but starts to decrease after $D = 512$. Meanwhile, the training time to obtain the prediction models increases along with D. This result suggests that $D = 512$ is optimal to provide good prediction performance with a reasonable time cost.

Table 4 lists all the parameters subjected to a grid search process for parameter tuning. For RF, the number of decision trees involved in learning, T, impacts the performance and efficiency of the trained model. A larger T generally leads to better prediction performance but increased computation time.

For kNN, the number of nearest neighbors, k, determines the neighborhood size considered in the prediction. A larger k reduces the effect of noisy samples but blurs the class boundaries.

For SVM, there are two performance-critical parameters: penalty coefficient, C, and width parameter, γ. C determines the tolerance for training errors in the decision function. The higher C is, the fewer training errors that can be tolerated, resulting in increased training accuracy but easier overfitting. The smaller the C value is, the more training errors can be tolerated, and the easier it is to obtain an underfitting classifier. The width parameter γ comes with the most widely adopted radial-basis function (RBF) kernel function. It implicitly determines the data distribution in the new feature space induced by the RBF kernel. The choice of γ affects the number of support vectors in the decision function, which in turn impacts the speed of training and prediction. We use a grid search to evaluate the effect of C and γ independent of each other. This approach also enables fast evaluation of multiple parameter settings using a simple parallel implementation.

For MLP, we adjust the hidden layer size, S, and the number of iterations, I. In a neural network, S determines the complexity of decision functions can be implemented and has a considerable impact on performance. I provides a trade-off between training error and the overfitting.

Finally, for LR, a regularization term in the adopted implementation, the sum of the squared coefficients multiplied by a parameter $\lambda \in \mathcal{R}^+$ is added to the objective function. A suitable regularization coefficient λ can help to reduce the generalization error without affecting the training error. The stopping criterion, τ, can be adjusted to find the optimal point to stop training to obtain both good accuracy and reasonable time cost. The examined grid values for all the tuned parameters are listed in the right-most column in Table 4.

5.4. Evaluation metrics

In the evaluation phase, we adopt common evaluation metrics, namely, accuracy, recall, precision, and F1-measure, to assess the performance of our proposed scheme. These metrics are defined based on the following intermediate measures.

- True positive (TP): samples correctly classified as positive.
- False positive (FP): samples incorrectly classified as positive.
- True negative (TN): samples correctly classified as negative.
- False negative (FN): samples incorrectly classified as positive.

Accuracy refers to the proportion of correct judgments:

$$Accuracy = \frac{TP + TN}{n}, \tag{2}$$

where n is the total number of samples used for evaluation.

Precision is the probability that predicted positives are correctly classified:

$$Precision = \frac{TP}{TP + FP}. \tag{3}$$

Recall is the probability of the samples in the positive class being classified correctly:

$$Recall = \frac{TP}{TP + FN}. \tag{4}$$

The F1-measure is the weighted average of precision and recall:

$$F1\_measure = \frac{2 \times (Recall \times Precision)}{Recall + Precision}. \tag{5}$$

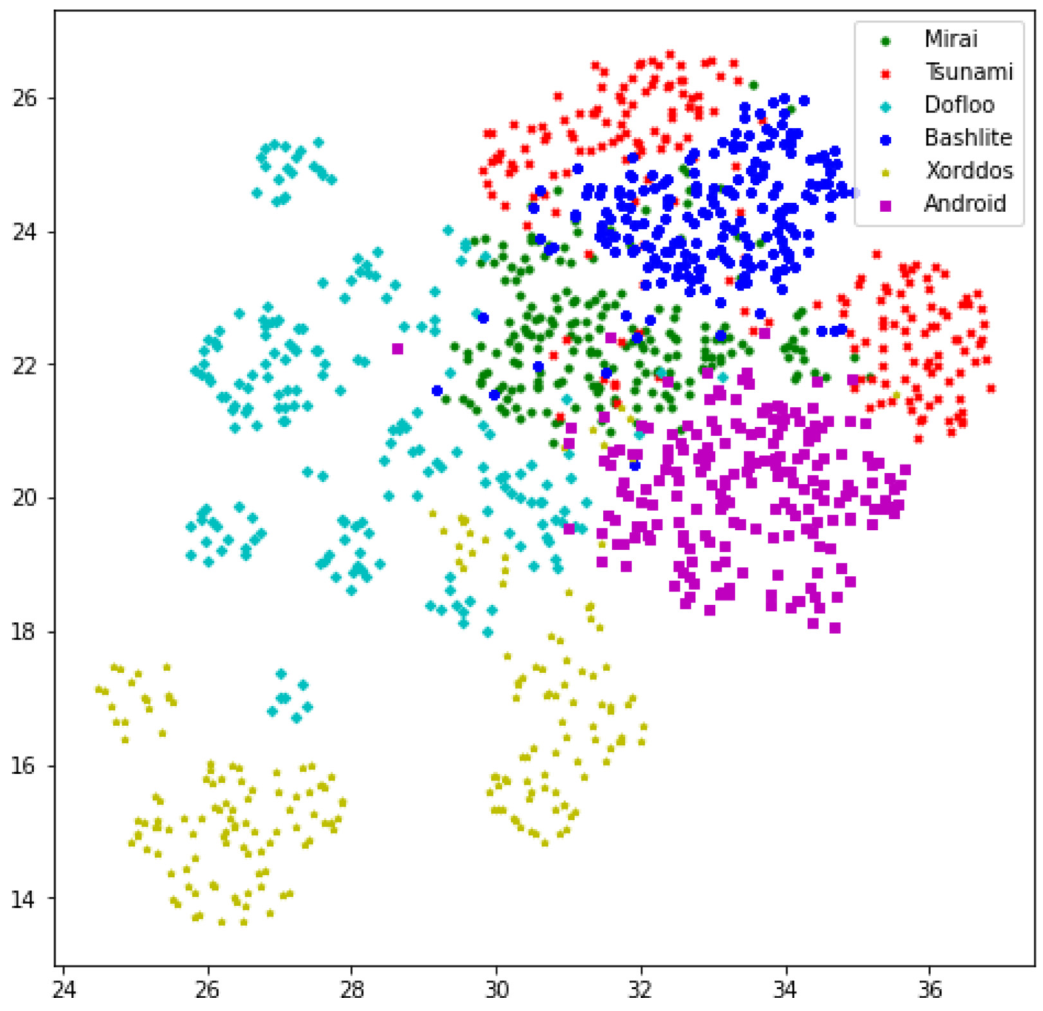


Fig. 6. Visualization of malware distribution using UMAP.

Table 4
Parameter tuning settings for classification algorithms.

Parameter	Classifier	Physical Meaning	Grid Value
D	All classifiers	Feature dimension	{128,256, ...,1024}
T	RF	Number of trees	{50,100, ...,500}
k	kNN	Number of nearest neighbors	{1,2, ...,9}
C	SVM	Penalty parameter	{10,100,1000}
γ	SVM	The width of the RBF kernel	{0.0001,0.001,0.01}
S	MLP	Size of hidden layer	{10,20, ...,100}
I	MLP	Maximum number of epochs	{10,20, ...,100}
λ	LR	Regularization coefficient	{0.1,1,10,100,1000}
τ	LR	Tolerance as stopping criteria	{0.0001,0.0001,0.001,0.01,1}

5.5. Performance evaluation

In this section, we report the results of four experiments to evaluate the performance of the proposed scheme. We use graph embedding features, the dimension determined via parameter tuning, and four graph structural features as the input attributes for the analysis. The first experiment verifies the feasibility of the reinterpreted FCG and enhanced *graph2vec* on malware family classification. In the second experiment, the dataset is divided into subsets based on the CPU architecture, and a performance comparison is conducted on CPU-specific subsets. The third experiment compares the performance of the proposed feature representation with the features introduced in related work. The last experiment compares the time efficiency of different approaches. All the reported results are obtained from 5-fold stratified cross-validation.

5.5.1. Malware family classification

The first experiment aims to verify the feasibility of the proposed approach of classifying IoT malware samples into known malware families. Classifying malware with high accuracy can enable prevention, such as malware quarantining or user alerting, soon after the binary is downloaded to the device. To demonstrate the feasibility of the FCG with reinterpreted UDFs and the implementation of *graph2vec* enhanced with literal information, we test the performance of the following settings.

1. FCG combined with *graph2vec* labeled with vertex degree (FCG×G2V),
2. FCG combined with *graph2vec* labeled with literal information (FCG×LiG2V), and

Table 5
Performance evaluation on IoT malware family classification.

Classifier	Accuracy(%)	Precision(%)	Recall(%)	F1-measure(%)	Training time(s)	Testing time(s)	Parameter setting
SVM	**98.88**	99.06	**98.57**	**98.81**	335.81	71.06	D=516, c=10, γ=0.001
MLP	98.70	98.65	98.46	98.55	66.62	0.07	D=900, size=100, iterations=50
RF	98.61	**99.12**	97.24	98.15	122.51	0.34	D=900, n=250
*k*NN	98.60	98.56	98.46	98.55	0.03	36.72	D=516, k=1
LR	98.04	98.41	97.04	97.67	80.81	0.03	D=772, λ=0.01, tol=0.0001

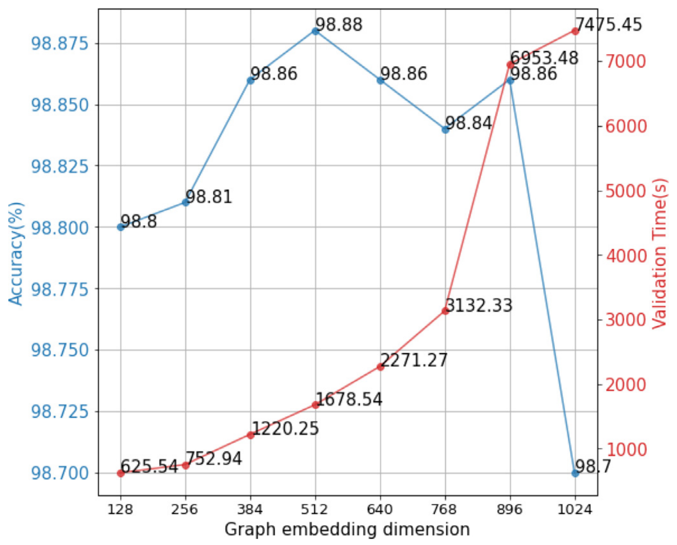


Fig. 7. Tuning parameter *D* using 5-fold cross-validation for SVM. *D* is selected from {128,256, ...,1024}.

3. Reinterpreted FCG with *graph2vec* labeled with literal information (RFCG×LiG2V).

Fig. 8 shows the experimental results obtained under these three settings. Among the charts in Fig. 8, (a) to (e) show how the classification accuracy varies with the feature dimension for the five selected classifiers. Chart (f) summarizes the results of the SVM classifiers in chart (d), which yielded the best performance. As the results obtained from the 5 selected classifiers show similar trends, we take the results from the SVM classifiers, as shown in (d) and (f), as an example for discussion. (d) shows that the ranking order of the three settings remains the same while the dimension parameter changes from 132 to 1,028. RFCG×LiG2V (red line) maintains the top ranking for all dimensions, followed by FCG×LiG2V (blue line) and FCG×G2V (green line) in descending order of classification accuracy. FCG×LiG2V outperforms FCG×G2V by a large margin. When $D = 516$, FCG×LiG2V improves the accuracy of FCG×G2V from 97.48% to 98.57%, indicating that integrating literal information into learning can yield substantial improvement in the discriminating ability for the embedding features. RFCG×LiG2V further improves upon the classification performance of FCG×LiG2V by a substantial margin. When $D = 516$, RFCG×LiG2V yields a classification accuracy of 98.88%, suggesting that reinterpreting UDFs in the graph helps to enhance the semantic information captured by *graph2vec*. Furthermore, (d) indicates that RFCG×LiG2V and FCG×LiG2V yield high accuracy (98.80% and 98.46% at $D = 132$, respectively) with comparable lower dimensional embeddings, and the performance appears to be very stable with varying dimensions. On the other hand, FCG×G2V requires comparatively high dimensional embeddings to achieve its best performance (97.68% accuracy obtained at $D = 1,028$). Similar observations can be made about the results in (a), (b), (c), and (e) obtained from the other four selected classifiers.

Table 5 summarizes the evaluation results of all examined classifiers, and the selected parameter settings are shown in the rightmost column. All the classifiers yielded near-optimal results with an accuracy greater than 98.00%. SVM achieved the highest accuracy of 98.88%, while RF, *k*NN, and MLP had slightly lower accuracy. LR yielded an accuracy of 98.04%, the weakest of the five classifiers. Because RFCG×LiG2V outperformed the other two settings by a large margin, all the results reported in the table are obtained using RFCG×LiG2V.

As a summary of the first experiment on IoT malware classification for all examined classifiers, we can summarize the feasibility study's results as follows.

1. Integrating literal information of the subroutine names in the *graph2vec* model substantially improve the prediction accuracy.
2. Reinterpreting UDFs in the FCGs leads to further improvement in the prediction accuracy for FCG×LiG2V.

5.5.2. *CPU-specific performance evaluation*

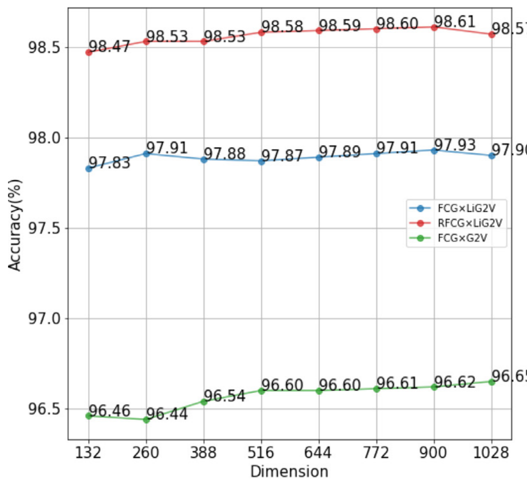
To reinterpret the UDFs, we used the associated opcode sequence to identify subroutines with the same functionality. Because different CPU architectures adopt different instruction sets, UDF reinterpretation works only for IoT malware compiled on the same CPU architecture. Therefore, better classification performance is expected if IoT malware family classification is performed on a dataset with all samples collected from a uniform CPU architecture. In the second experiment, we divided the dataset into seven subsets containing malware samples compiled on the same CPU architecture. Then, an analysis following the steps in the first experiment was performed on these seven subsets (Table 5).

Table 6 shows the results for IoT malware family classification for each CPU architecture. We report only the results obtained by RF and SVM for better readability: *k*NN, MLP, and LR showed slightly inferior generalization performance compared to RF and SVM. For IoT malware family classification, SVM yielded accuracy greater than 98.92% on MIPS, X86, and SPARC at $D = 516$. For two of the other four remaining CPU architectures, ARM and PPC, SVM also achieved accuracy close to 98.92%. SVM yielded inferior performance only on X86-64 and UNKNOWN. Although slightly lower than that of SVM, RF also achieved an accuracy greater than 98.64% on 5 of the 7 CPU architectures.

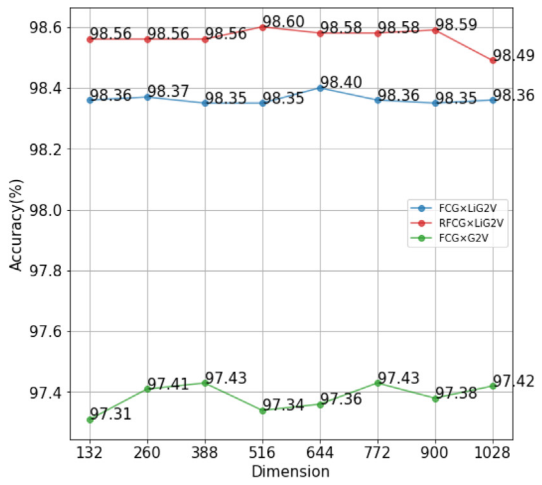
We investigated why the classifiers showed comparatively low accuracy on X86-64 and UNKNOWN. A simple explanation in the case of X86-64 is that samples collected on this CPU architecture complied on a much wider range of Linux distributions. The increased sample variance renders the classification more difficult. On the other hand, the subset collected on UNKNOWN contained a variety of CPU architectures, but the sample size is reduced considerably compared with that of the full dataset. Moreover, the distinct instruction sets on different CPU architectures also cast a shadow on the efficacy of FCG reinterpretation. These two factors rendered the classification tasks more difficult for the UNKNOWN subset.

5.5.3. *Performance comparison with related work*

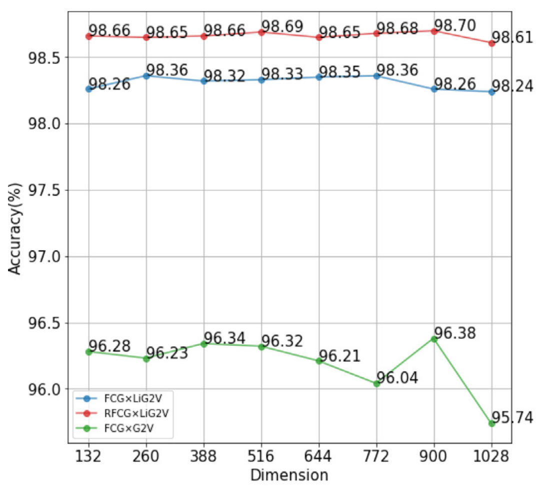
In the third experiment, we compare the proposed approach with selected approaches in related work. As reviewed in Section 2,



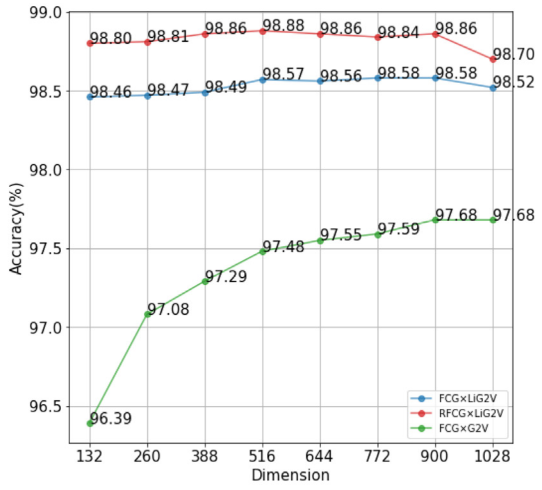
(a) RF



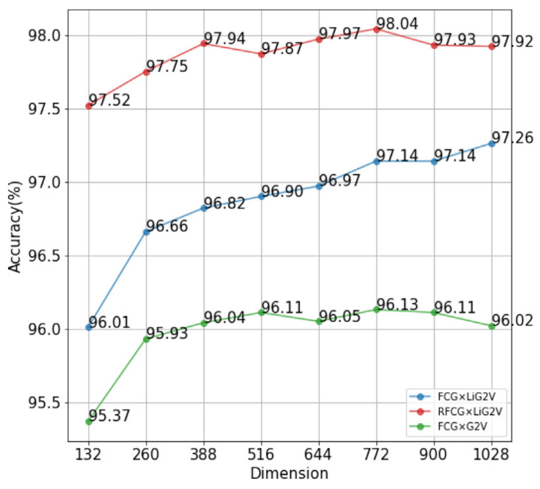
(b) kNN



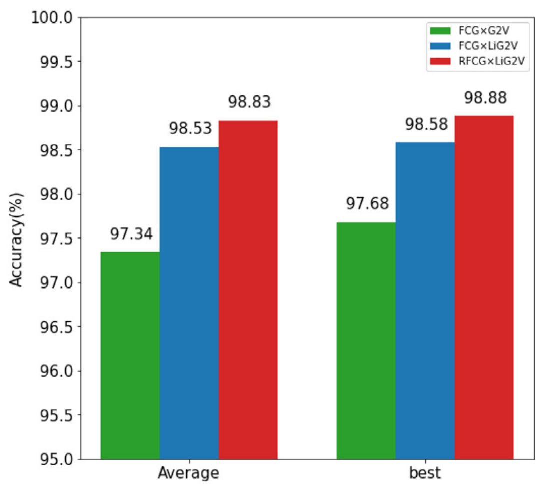
(c) MLP



(d) SVM



(e) LR



(f) Summary of SVM accuracy

Fig. 8. Performance comparison between RFCG×G2V, FCG×LiG2V, and FCG×G2V.

Table 6
Performance evaluation on CPU-specific malware family classification.

CPU-architecture	Classifier	Accuracy(%)	Precision(%)	Recall(%)	F1-measure(%)
ARM	RF	98.64	92.31	89.90	91.02
	SVM	**98.85**	**98.85**	**98.32**	**98.57**
MIPS	RF	99.03	87.14	84.19	85.31
	SVM	**99.10**	**98.80**	**98.72**	**98.75**
X86	RF	98.89	**99.36**	98.90	**99.13**
	SVM	**98.92**	99.17	**99.05**	99.10
SPARC	RF	99.08	98.70	92.39	95.07
	SVM	**99.12**	**98.84**	**94.76**	**96.47**
X86-	RF	97.61	**94.17**	89.39	**91.41**
64	SVM	**97.85**	93.52	**92.73**	93.05
PPC	RF	98.69	**93.72**	89.66	91.49
	SVM	**98.80**	93.43	**90.62**	**91.77**
UNKNOWN	RF	97.26	97.57	90.61	93.59
	SVM	**97.91**	**97.71**	**95.95**	**96.75**

different types of features associated with static malware analysis are studied in the literature. We select approaches that are easily adaptable for IoT malware classification. Specifically, we implement the graph-based features introduced by Alasmary et al. (2019), the features based on binary header information presented by Shahzad and Farooq (2012), the *N*-gram features obtained from opcode sequences introduced by Kang et al. (2016), and the graph features based on opcode sequences presented by Gülmez and Sogukpinar (2021). Note that this experiment aims to evaluate the discriminating ability of a certain type of static feature not to search for the classification model that gives optimal prediction performance. Therefore, we choose to implement the five widely adopted classifiers introduced in Section 4.4 based on these features to conduct a fair and general comparison. More specialized classification algorithms could work extraordinarily well on one or more feature types; searching for such optimal combinations is beyond the scope of this work. In this experiment, we used a shuffled subset of 10K malware examples for training and a shuffled subset of 10K for testing. The reported results are obtained via 5-fold stratified cross-validation.

Table 7 shows the results obtained from each type of feature for all the selected classifiers. The prediction performance is measured in terms of accuracy, precision, recall, and F1-measure. Note that the samples are subject to a severely skewed distribution among different malware families. The reduction in sample size of the testing set casts a shadow on criteria other than accuracy because the macro average over the six classes favors results in small classes. In the following, we use accuracy as the main performance criterion. Although other measures are subject to significant variation compared with the previous experiments, the comparison results adhere to those obtained from accuracy. As shown in the table, the features based on graph theory from Alasmary et al. (2019) yielded greater than 96.64% accuracy for 4 of 5 classifiers; they produced a comparatively low accuracy of 91.61% for LR. The features extracted from ELF headers from Shahzad and Farooq (2012) and the opcode-based features from Kang et al. (2016) yielded stable accuracy above 95.13% for all five classifiers. The features learned from opcode graphs from Gülmez and Sogukpinar (2021) obtained a very high accuracy of 98.42% with RF. Meanwhile, it yielded a relatively low accuracy of 88.82% with LR and an extremely low accuracy of 47.85% with MLP. Thus, the performance of opcode-graph-based features is not always stable and may not be suitable for certain classifiers.

Under the RFCG$\times$LiG2V setting, the proposed graph embedding features yielded the best performance in IoT malware classification (98.89% accuracy with SVM, 85.0% precision with RF, 84.76% recall with MLP, and 84.83% F1-measure with SVM). This outstanding performance compared with that of existing methods proves that reinterpreted FCG with *graph2vec* labeled with literal information captures essential discrimination information for IoT malware family classification.

5.5.4. Time efficiency comparison

In this subsection, we compare the time efficiency of the proposed method with related work. As summarized in Table 8, we divide the malware classification process into reverse engineering, feature extraction, (model) training, and prediction. The time used for reverse engineering accounts for applying *radare*2 to perform static analysis on a malware binary. The time used for feature extraction accounts for all the time spent transforming the static analysis log into a numerical vector ready to be input to the classifier. In particular, for the proposed scheme, the time to obtain the graph representation of an FCG, reinterpret the UDFs, and perform graph embedding is integrated into the feature extraction time. The training time constitutes the time cost for building a classifier model with all training samples. We report the average time used to build the classifiers at all steps of the 5-fold cross-validation as training time. Note that as the mechanism for parameter tuning is not directly comparable for different classification algorithms, we exclude the time spent on parameter tuning from the training time. The testing time is the time it takes to apply a model to predict the class label for a test instance based on the vector representation of the sample.

The second column in Table 8 reports the time spent on reverse engineering for all compared methods. Because the features introduced in Shahzad and Farooq (2012) can be directly extracted from ELF headers, it does not need a reverse engineering step. On the other hand, advanced feature types are generated based on opcode sequences for the other four approaches. Therefore, for these approaches, reverse engineering is required before feature extraction. On the evaluation dataset, the average time to perform reverse engineering on a single file using *radare*2 is 1.8623 s, a substantial computational overhead for malware classification.

The third column in Table 8 reports the time spent on feature extraction. We can see that the features extracted from the ELF headers as introduced in Shahzad and Farooq (2012) are very time efficient, with an average time cost of 0.0620 s per file. The *N*-gram extracted from the opcode sequence as introduced in Kang et al. (2016) also works efficiently on IoT malware with an average time cost of 0.0128 s per file. The feature introduced in Gülmez and Sogukpinar (2021) is extracted from a condensed graph presentation of the opcode sequences, yielding an average time cost of 0.1417 s per file. Because computation-intensive features such as the shortest path length of the CFGs are considered, the structural features introduced in Alasmary et al. (2019) require an average feature extraction time of 1.1566 s per file. The aver-

Table 7
Performance comparison with previous work based on static features.

Related work	Feature	Classifier	Accuracy(%)	Precision(%)	Recall(%)	F1-measure(%)
Shahzad and Farooq (2012)	ELF header	SVM	97.39	81.54	81.38	81.42
		MLP	96.94	81.35	79.54	80.32
		RF	97.55	83.55	80.73	81.91
		*k*NN	95.13	81.00	78.08	79.19
		LR	95.53	81.64	77.00	78.64
		Avg.	96.51	81.42	79.35	80.30
Alasmary et al. (2019)	Graph theory	SVM	97.78	83.95	82.56	83.24
		MLP	96.64	81.38	82.50	81.89
		RF	97.50	83.95	82.01	82.91
		*k*NN	97.61	83.25	82.22	82.72
		LR	91.61	76.96	71.55	73.68
		Avg.	96.23	81.90	80.17	80.89
Kang et al. (2016)	Opcode	SVM	96.78	59.51	60.22	59.47
		MLP	96.59	59.68	60.08	59.48
		RF	97.02	59.58	60.68	59.81
		*k*NN	96.50	58.61	60.16	59.08
		LR	96.31	62.01	60.05	60.88
		Avg.	96.64	59.88	60.24	59.74
Gülmez and Sogukpinar (2021)	Opcode graph	SVM	94.34	83.91	77.58	80.27
		MLP	47.85	6.84	14.29	9.25
		RF	98.42	84.28	82.63	83.43
		*k*NN	96.82	81.68	81.31	81.47
		LR	88.82	61.07	64.26	62.12
		Avg.	85.25	63.56	64.01	63.31
RFCG×LiG2V	Graph embedding	SVM	**98.91**	**85.00**	84.66	**84.83**
		MLP	98.78	84.86	**84.76**	84.81
		RF	98.59	**85.00**	83.13	84.04
		*k*NN	98.59	84.69	84.34	84.50
		LR	98.40	84.30	84.70	84.49
		Avg.	98.65	84.77	84.32	84.53

Table 8
Time efficiency comparison with previous work based on static features.

Related work	Rev. eng. (per file)	Feat. extr. (per file)	Classifier	Training	Prediction (per file)	MTTD (per file)
Shahzad and Farooq (2012)	-	0.0620	SVM	5.6182	2.22×10^{-3}	0.0642
			MLP	12.6789	1.43×10^{-4}	0.0621
			RF	0.2219	2.06×10^{-4}	0.0622
			*k*NN	0.0050	3.26×10^{-3}	0.0653
			LR	5.8607	1.39×10^{-4}	0.0620
Alasmary et al. (2019)	1.8623	1.1566	SVM	1.1986	8.50×10^{-4}	3.0197
			MLP	16.4680	9.96×10^{-5}	3.0190
			RF	0.9083	2.55×10^{-4}	3.0191
			*k*NN	0.0024	2.84×10^{-3}	3.022
			LR	0.9842	8.22×10^{-5}	3.0190
Kang et al. (2016)	1.8623	0.0128	SVM	9.4033	6.86×10^{-3}	1.8820
			MLP	147.5743	3.21×10^{-4}	1.8754
			RF	6.4611	3.49×10^{-4}	1.8754
			*k*NN	0.0490	4.69×10^{-3}	1.8798
			LR	6.1600	1.57×10^{-4}	1.8753
Gülmez and Sogukpinar (2021)	1.8623	0.1417	SVM	477.4096	1.75×10^{-1}	2.179
			MLP	11.7092	1.17×10^{-2}	2.0157
			RF	2.6422	2.14×10^{-4}	2.0042
			*k*NN	0.2682	4.76×10^{-3}	2.0088
			LR	69.9998	1.30×10^{-4}	2.0041
RFCG×LiG2V	1.8623	0.6196	SVM	1.3124	1.14×10^{-3}	2.4830
			MLP	2.1390	5.57×10^{-5}	2.4820
			RF	2.9778	2.40×10^{-4}	2.4821
			*k*NN	0.0032	2.73×10^{-3}	2.4846
			LR	2.8083	5.27×10^{-5}	2.4820

Note: All numbers are in seconds.

age feature extraction cost for RFCG×LiG2V is 0.6196 s per file. Considering that it integrates a subset of the features introduced in Alasmary et al. (2019) together with the graph embedding features extracted by *graph2vec*, the implementation is computationally efficient.

The training time to build a classification model is determined by factors such as the dimension of the feature vectors, the data's separability, and the algorithm's optimization strategy. As shown in the table, for the sake of employing an ensemble of fast linear classifiers to perform the classification, RF performs fast training for all feature settings. All RF models are built within less than 10 s. SVM, MLP, and LR are subject to strong variation from a few seconds to a few hundred seconds, mainly due to the feature representation's change in dimension. As a special case, *k*NN adopts the so-called lazy learning strategy that does not require training a discriminate function from the training data. This lazy learning scheme comes with a relatively expensive time cost for prediction and a large space cost to keep all the training samples in memory. Based on the vector representation obtained by RFCG×LiG2V, all five checked classification algorithms reported a fast training time of less than 3 s. This is attributed to the strong capability of *graph2vec* to capture essential discriminant information with a very low feature dimension.

As a key performance indicator for incident management, the mean time to detect (MTTD) is defined as the average time the detector takes to identify the threat successfully. The rightmost column in Table 8 reports the average MTTD, which is the time spent performing reverse engineering, feature extraction, and prediction for all evaluated approaches. A shorter MTTD indicates that users suffer from disruptions for less time than with a longer MTTD. As seen from the table, the time spent on reverse engineering and feature extraction constitutes the major part of MTTD. Compared with other methods, the approach introduced in Shahzad and Farooq (2012) yielded a very short MTTD. The other four approaches, which involve a reverse engineering step to obtain the static analysis results, show comparable performance in terms of MTTD. In particular, the proposed approach reported an average MTTD of approximately 2.5 s for all evaluated classification algorithms.

6. Discussion

In this section, we discuss the limitations of the proposed scheme and the application scenarios that can benefit from its high prediction performance.

6.1. Limitations of static analysis

The proposed approach is based on reverse engineering IoT malware binaries to obtain FCGs to understand their behavior. Code obfuscation is frequently employed by malware attackers to hinder such analysis (Schrittwieser and Katzenbeisser, 2011). Executable computation, commonly with the help of software tools known as runtime packers, is a process that compresses an executable file and combines the compressed data with decompression code into a single executable. When the compressed executable is executed, the decompression code recreates the original code from the compressed data and then passes the control. As reported in Aghakhani et al. (2020), for the case of Windows portable executables, packing is common not only in malware samples (75%) but also in benign samples (more than 50%). The good news for IoT malware is that very few cases of obfuscated IoT malware have been observed thus far (Wan et al., 2020). Therefore, we believe the proposed approach can be an efficient solution for IoT malware protection for the time being. Future work will cover an extended study on the countermeasures of obfuscation techniques on IoT malware.

6.2. Application scenarios

Note that the proposed scheme works on FCGs and the output of advanced reverse engineering tools such as *radare*2. Moreover, obtaining the embeddings using reinterpreted FCGs may incur a prohibitive computation cost on a resource-constrained device. Therefore, the application of the proposed scheme is possible only on IoT devices with comparatively abundant resources, e.g., AI speakers and home routers. Another plausible application scenario of the proposed scheme is on a smart home cybersecurity hub that can provide overall protection of various internet-connected devices from attacks such as malware, stolen passwords, and identity theft. Finally, the proposed scheme can be applied at security operation centers of enterprises and research laboratories of security vendors. In these scenarios, extensive analysis of collected malware binaries must be performed with high accuracy to enable effective countermeasure policies.

7. Conclusion

In this paper, we propose a new scheme to apply the well-known graph embedding approach, *graph2vec*, to analyze IoT malware. To improve the generalization performance of the scheme, we first devise a preprocessing step that reinterprets the UDFs using their associated opcode sequences to obtain refined semantics from the FCG. Then, we present an enhanced implementation of *graph2vec* that effectively integrates the literal information of the subroutine names in FCGs into the learned embedding. Finally, the graph embedding features from the proposed scheme are combined with graph structural features to facilitate training models for IoT malware family classification. The effectiveness and efficiency of the proposed scheme are evaluated by experiments conducted on a large-scale benchmark dataset consisting of more than 108K IoT malware samples. The experimental results show that integrating literal information of the subroutine names into the *graph2vec* model can yield malware classification models with high classification accuracy. Reinterpreting UDFs in the FCGs leads to further improvement, resulting in an SVM classifier with a near-optimal accuracy of 98.88% for IoT malware family classification. We believe that promising solutions for IoT security can be developed based on the findings presented in this paper.

Declaration of Competing Interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Tao Ban reports financial support was provided by The Ministry of Education, Science, Sports, and Culture, Japan.

CRediT authorship contribution statement

Chia-Yi Wu: Methodology, Software, Writing – original draft. **Tao Ban:** Conceptualization, Data curation, Writing – review & editing, Supervision, Funding acquisition. **Shin-Ming Cheng:** Data curation, Supervision, Funding acquisition. **Takeshi Takahashi:** Data curation, Project administration. **Daisuke Inoue:** Funding acquisition.

Data availability

Data will be made available on request.

Acknowledgments

This work was partially supported by JSPS/MEXT KAKENHI Grant Number JP22K12038.

References

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C., 2020. When malware is packin' heat; limits of machine learning classifiers based on static analysis features. NDSS.
Alasmary, H., Khormali, A., Anwar, A., Park, J., Choi, J., Abusnaina, A., Awad, A., Nyang, D., Mohaisen, A., 2019. Analyzing and detecting emerging internet of things malware: a graph-based approach. IEEE Internet Things J. 6 (5), 8977–8988.
Antonakakis, M., et al., 2017. Understanding the Mirai botnet. In: Proc. USENIX Security 2017, pp. 1093–1110.
Ban, T., Isawa, R., Huang, S., Yoshioka, K., Inoue, D., 2019. A cross-platform study on emerging malicious programs targeting IoT devices. IEICE Trans. Inform. Syst. 102-D (9), 1683–1685.

Ban, T., Takahashi, T., Guo, S., Inoue, D., Nakao, K., 2016. Integration of multi-modal features for android malware detection using linear SVM. In: 2016 11th Asia Joint Conference on Information Security (AsiaJCIS), pp. 141–146. doi:10.1109/AsiaJCIS.2016.29.

Chaganti, R., Ravi, V., Pham, T.D., 2022. Deep learning based cross architecture internet of things malware detection and classification. Comput. Secur. 120, 102779.

Cortes, C., Vapnik, V., 1995. Support-vector networks. Mach. Learn. 20 (3), 273–297.

Costin, A., Zaddach, J., 2018. IoT malware: comprehensive survey, analysis framework and case studies. In: Proc. BlackHat USA 2018.

D'Angelo, G., Ficco, M., Palmieri, F., 2021. Association rule-based malware classification using common subsequences of API calls. Appl. Soft Comput. 105, 107234. doi:10.1016/j.asoc.2021.107234.

Galal, H.S., Mahdy, Y.B., Atiea, M.A., 2015. Behavior-based features model for malware detection. J. Comput. Virol. Hacking Tech. 12. doi:10.1007/s11416-015-0244-0.

Greene, W.H., 2012. Econometric Analysis. Vol. Pearson series in economics, 7th ed., international ed. Pearson Education.

Guardian, 2022. DDoS attack that disrupted Internet was largest of its kind in history, experts say. last accessed, Jul. 10, 2022. https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

Gülmez, S., Sogukpinar, I., 2021. Graph-based malware detection using opcode sequences. In: Proc. ISDFS 2021, pp. 1–5.

Hastie, T., Tibshirani, R., Friedman, J., 2009. The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd ed. Springer.

Haykin, S., 1999. Neural Networks: A Comprehensive Foundation. Prentice Hall.

Herwig, S., Harvey, K., Hughey, G., Roberts, R., Levin, D., 2019. Measurement and analysis of Hajime, a peer-to-peer IoT botnet. Network and Distributed Systems Security (NDSS) Symposium.

Ho, T.K., 1998. The random subspace method for constructing decision forests. IEEE Trans. Pattern Anal. Mach. Intell. 20 (8), 832–844.

Hosmer, D.W., Lemeshow, S., 2000. Applied Logistic Regression. John Wiley and Sons.

Kang, B., Yerima, S.Y., McLaughlin, K., Sezer, S., 2016. N-opcode analysis for android malware classification and categorization. In: Proc. Cyber Security, pp. 1–7.

Kawasoe, R., Han, C., Isawa, R., Takahashi, T., Takahashi, J., 2021. Investigating behavioral differences between IoT malware via function call sequence graphs. In: Proc. ACM SAC, pp. 1674–1682.

Kuang, B., Fu, A., Zhou, L., Susilo, W., Zhang, Y., 2020. DO-RA: data-oriented runtime attestation for IoT devices. Comput. Secur. 97, 101945.

Kumar, A., Shridhar, M., Swaminathan, S., Lim, T.J., 2022. Machine learning-based early detection of IoT botnets using network-edge traffic. Comput. Secur. 117, 102693.

Le, Q., Mikolov, T., 2014. Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning. PMLR, Bejing, China, pp. 1188–1196.

Lee, Y.-T., Ban, T., Wan, T.-L., Cheng, S.-M., Isawa, R., Takahashi, T., Inoue, D., 2020. Cross platform IoT-malware family classification based on printable strings. In: Proc. IEEE TrustCom 2020.

Li, C., Shen, G., Sun, W., 2021. Cross-architecture internet-of-things malware detection based on graph neural network. In: Proc. IJCNN 2021, pp. 1–7.

Marzano, A., Alexander, D., Fonseca, O., Fazzion, E., Hoepers, C., Steding-Jessen, K., Chaves, M.H., Cunha, Í., Guedes, D., Meira, W., 2018. The evolution of Bashlite and Mirai IoT botnets. In: Proc. IEEE ISCC 2018, pp. 813–818.

McInnes, L., Healy, J., Melville, J., 2018. UMAP: uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426.

Muzaffar, A., Ragab Hassen, H., Lones, M.A., Zantout, H., 2022. An in-depth review of machine learning based android malware detection. Comput. Secur. 121, 102833.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S., 2017a. graph2vec: Learning distributed representations of graphs. arXiv preprint arXiv:1707.05005.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S., 2017. graph2vec: Learning distributed representations of graphs. CoRR. abs/1707.05005.

Naseer, M., Rusdi, J., Shanono, N., Salam, S., Zulkiflee, M., Abu, N., Abadi, I., 2021. Malware detection: issues and challenges. J. Phys. Conf. Ser. 1807, 012011. doi:10.1088/1742-6596/1807/1/012011.

Natani, P., Vidyarthi, D., 2013. Malware detection using API function frequency with ensemble based classifier. In: Security in Computing and Communications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 378–388.

Ngo, Q.-D., Nguyen, H.-T., Le, V.-H., Nguyen, D.-H., 2020. A survey of IoT malware and detection methods based on static features. ICT Express 6 (4), 280–286.

Nguyen, H.-T., Ngo, Q.-D., Le, V.-H., 2020. A novel graph-based approach for IoT botnet detection. IJISS 19 (5), 567–577.

Onwuzurike, L., Mariconti, E., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G., 2017. MaMaDroid: detecting android malware by building Markov chains of behavioral models (extended version). ACM Trans. Privacy Secur. 22. doi:10.1145/3313391.

Ou, F., Xu, J., 2022. S3Feature: a static sensitive subgraph-based feature for android malware detection. Comput. Secur. 112, 102513.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. scikit-learn: Machine learning in Python. J. Mach. Learn. Res. 12, 2825–2830 .

Qiang, W., Yang, L., Jin, H., 2022. Efficient and robust malware detection based on control flow traces using deep neural networks. Comput. Secur. 122, 102871.

Radare2, 2022. last accessed, Jul. 10, 2022. https://rada.re/r/.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C., 2018. Malware detection by eating a whole EXE. In: Proc. AAAI 2018.

Schrittwieser, S., Katzenbeisser, S., 2011. Code obfuscation against static and dynamic reverse engineering. In: Information Hiding. Springer Berlin Heidelberg, pp. 270–284.

Shahzad, F., Farooq, M., 2012. ELF-Miner: using structural knowledge and data mining methods to detect new Linux malicious executables. Knowl. Inf. Syst. 30 (3), 589–612.

Su, J., Vasconcellos, D.V., Prasad, S., Sgandurra, D., Feng, Y., Sakurai, K., 2018. Lightweight classification of IoT malware based on image recognition. In: Proc. IEEE COMPSAC 2018, pp. 664–669.

Vinayaka, K., Jaidhar, C., 2021. Android malware detection using function call graph with graph convolutional networks. In: Proc. ICSCCC 2021.

VirusTotal, 2022. last accessed: Jul. 10, 2022. https://www.virustotal.com/.

Wan, T.-L., Ban, T., Cheng, S.-M., Lee, Y.-T., Sun, B., Isawa, R., Takahashi, T., Inoue, D., 2020. Efficient detection and classification of internet-of-things malware based on byte sequences from executable files. IEEE Open J. Comput. Soc. 1, 262–275.

Wazzan, M., Algazzawi, D., Bamasaq, O., Albeshri, A., Cheng, L., 2021. Internet of things botnet detection approaches: analysis and recommendations for future research. Appl. Sci. 11 (12), 5713.

Wu, C.-Y., Ban, T., Cheng, S.-M., Sun, B., Takahashi, T., 2021. IoT malware detection using function-call-graph embedding. In: Proc. PST 2021, pp. 1–9.

Xiao, F., Sun, Y., Du, D., Li, X., Luo, M., 2020. A novel malware classification method based on crucial behavior. Math. Probl. Eng. 2020.

Xu, M., 2021. Understanding graph embedding methods and their applications. SIAM Rev. 63 (4), 825–853. doi:10.1137/20M1386062.

You, I., Yim, K., 2010. Malware obfuscation techniques: a brief survey. In: Proceedings - 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010, pp. 297–300. doi:10.1109/BWCCA.2010.85.

Zhang, Y., Chang, X., Lin, Y., Mišić, J., Mišić, V.B., 2020. Exploring function call graph vectorization and file statistical features in malicious PE file classification. IEEE Access 8, 44652–44660.

Chia-Yi Wu received the BE degree in computer science and information engineering from Yuan Ze University, Taoyuan, Taiwan, in 2020 and the ME degree from National Taiwan University of Science and Technology, Taipei, Taiwan, in 2022. His current research interests include security for IoT and machine learning.

Tao Ban received his BE degree from Xi'an Jiaotong University in 1999, ME degree from Tsinghua University in 2003, and PhD degree from Kobe University in 2006, respectively. He is currently a senior researcher with Cybersecurity Research Institute, National Institute of Information and Communications Technology, Tokyo, Japan. His research interest includes network security, malware analysis, machine learning, and data mining.

Shin-Ming Cheng received the B.S. and Ph.D. degrees in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 2000 and 2007, respectively. Since 2012, he has been on the faculty of the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, where he is currently a professor. Since 2017, he has been with the Research Center for Information Technology Innovation, Academia Sinica, Taipei, where he is currently a joint appointment research fellow.

Takeshi Takahashi received his Ph.D. degree in communications from Waseda University in 2005. He worked for the Tampere University of Technology from 2002 to 2004 as a researcher, Waseda University as a JSPS research fellow from 2004 to 2006, and Roland Berger Ltd. from 2006 to 2009 as a business consultant. He has been with the National Institute of Information and Communications Technology since 2009 and is currently an associate director. Further, he was a visiting research scholar at the University of California, Santa Barbara, from 2019 to 2020. His research interests include cybersecurity and machine learning.

Daisuke Inoue received his B.E. and M.E. degrees in electrical and computer engineering and Ph.D. degree in engineering from Yokohama National University in 1998, 2000 and 2003, respectively. He joined Communications Research Laboratory (CRL), in 2003. CRL was relaunched as National Institute of Information and Communications Technology (NICT) in 2004, where he is currently the director general of Cybersecurity Nexus (CYNEX) and the director of Cybersecurity Laboratory. His research interests include practical cybersecurity technologies, and security visualization.