IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 22, 2019, accepted March 19, 2019, date of current version April 29, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910268

A Consistently-Executing Graph-Based Approach for Malware Packer Identification

XINGWEI LI, ZHENG SHAN, FUDONG LIU, YIHANG CHEN, AND YIFAN HOU
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China

Corresponding authors: Zheng Shan (zzzhengming@163.com) and Fudong Liu (lwfydy@126.com)

This work was supported by the National Natural Science Foundation of China under Grant 61802435 and Grant 61802433.

ABSTRACT Packing is the most common malware obfuscation technique employed to evade detection by anti-virus systems. With the explosive growth of packed malware, packer identification has become increasingly important in current cybersecurity. Most prior studies lack resilience since they neglect the significance of preserving the semantic integrity for pursuing efficiency. Therefore, in this paper, we propose a novel approach based on consistently executing graph (CEG) mining, which can maximize semantics and facilitate the execution intents of executables for identification. We elaborate on the concepts of consistent execution and represent malware packers as structured CEGs. Then, we employ the bipartite matching algorithm built on a designed block proximity metric to extract the critical matching subgraphs. Weisfeiler–Lehman shortest path graph kernel is adopted for pairwise comparison to make the prediction. We have evaluated the efficacy extensively with several experiments on large-scale datasets containing manually packed benign apps, wild-packed malware, and wild-unpacked malware. The promising results demonstrate that our approach is valid and practical when applied to real-world packed malware analysis.

INDEX TERMS Malware analysis, packer identification, consistent execution, graph kernel.

I. INTRODUCTION

Malicious code, also known as malware (e.g. virus, worm, trojan), is one of the most severe threat in modern computer society over the past decade. Although anti-virus (AV) researchers are developing scanners to keep pace with the speed of novel malware persistently, coping with malware remains challenging with their rapid growth both on volume and complexity. Worse still, a large amount of malware adopts obfuscation techniques involving polymorphism, metamorphism and virtualization to disguise themselves. When in the propagation phase, their camouflage can assist them in evading detection and achieving their goals sophisticatedly.

Packer is one of the most widely used obfuscator. Over 80% malware in the wild appears to be obfuscated using packers, particularly in Advanced Persistent Threat (APT). It was originally a benign software program designed to compress other executables and restore the image when the packed file loaded into memory for saving storage [1]. Unfortunately, packers are despitefully used by the authors of malware for obfuscating their original malicious code so as to eliminate malign features. A packed malware typically contains an encrypted payload that has a virus decryption routine (VDR) to decrypts their encrypted virus body (EVB) and a mutation engine (ME) to change VDR during each iteration. They cause the ineffectiveness of traditional detections because their malicious intentions are transformed into random byte sequences. Based on their behaviors, we can broadly classify malware packers into four categories: compressors, crypters, protectors and new generation of virtual machine (VM) protectors (i.e. virtualizer) [2].

Although there are many significant discussions both on academia and industry, packing is still recognized as a typical analysis challenge. Therefore, similar to the motivation of malware triage [3], it is urgent to identify malware packers under the increasing volume of packed samples. Meanwhile, accurate identification for malware packers can also contribute to the unpack and retrieval for more in-depth analysis. To settle this tricky problem, *a reliable packer classification scheme should be proposed to facilitate the identification, characterize the process and reduce the cost significantly* [4]. As is known, the most significant advantage of dynamic analysis over static analysis is that dynamic analysis can represent the executing status and verify the action in practice [5].

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek.

However, the static representation can achieve higher code coverage, expressing more comprehensive and efficient. Our motivation is to combine both advantages to create an efficient construction in static analysis but able to express or excavate its possible behavior approximately. Based on the representation, we devise methodologies to further research and achieve the stated effects.

In this paper, we represent an automatic approach to identify malware packers based on consistently-executing graph (CEG) mining. The main contributions are as follows.

- we define a series of related concepts of consistent execution, develop a strategy to generate executable's CEG by slicing disassembly code and extracting semantic information.
- we newly design a novel consistent-executing block (CEB) proximity metrics to obtain node correspondence preparing to extract the critical matching subgraphs.
- we employ Weisfeiler-Lehman shortest path kernel to build a similarity measurement standard for our CEG graph classifier and achieve a fine result.
- we carry out multiple testings both on the manually-packed benign apps, real world wild-packed malware and large scale wild-unpacked malware to validate our approach and demonstrate the practicality of our system.

The remainder of the paper is organized as follows. Section II briefly overviews related studies and ideas of malware packers. Section III introduces our proposed approach in detail. Section IV describes experiments and reports evaluation results to validate our system. Section V summarizes our whole work and discusses several future directions.

II. RELATED WORK

From systematical reveal of packed malware [1], they have been well acknowledged and understood in the context of different operating systems, such as Windows [6], Linux [7], Android [8]. Besides the latest talk [9] raises three concerns to indicate that malware packers are far more complicated than researchers think. The counterwork between packing and unpacking will be conducted continuously. Equally, there are various methods or prototypes have been proposed for malware packer identification. Bat-Erdene, Munkhbayar, *et al.* [10] develop entropy analysis to classify unknown packing algorithms for detection, BE-PUM [11] and BinUnpack [12] design generic dynamic analysis systems for unpacking. Generally, solutions for packer identification can be divided into three main approaches.

1) **Signature-based approaches.** Signature-based approaches mainly rely on matching the generated signatures or regulations. It can be considered as a 01 issue. The standard tools PEiD,[1] Exeinfo,[2] DIE[3] use packers signature database containing at most around 3000 signatures of different packers for identification. In addition, yara[4] is a more heuristic method based on mining textual or binary pattern rules. Hai *et al.* design BE-PUM [11] to generate *metadata signature* for identification using symbolic execution technique.
2) **Behavior-based approaches.** Behavior-based approaches execute packed malware adopted by sandboxs [13], simulators [6] or debugging environments [14] to record packer behaviors. Bat-Erdene *et al.* [10], [15] employ time series analysis to metric the alter of entropy when executing. Ugarte-Pedrero *et al.* [6] implement a framework based on TEMU[5] to analyze packers and public release at Deep Packer Inspector.[6] Moreover, Panda [16], CoDisasm [17], BinUnpack [12], are all remarkable unpacking platforms.
3) **Feature-based approaches.** Feature-based approaches extract essential features for instance, entropy, randomness [4], file header [18], memory image [19], unpacking routine [20] or other combined characteristics [21]. Especially, artificial intelligence performs well on separating features so being applied in classification naturally. For example, Zhang, *et al.* [13] apply multiLayers neural networks to differentiate sensitive system calls sequence for packed malware variants detection.

However, all of the above approaches have particular strengths and limitations. Signature-based methods rely on the update frequency of the signature database and have poor performance in resilience and generalization. In contrast, we develop a system to generate CEG and boost itself by feedback automatically. The main weakness of behavior-based approaches are computation expensive and coverage noncomprehensive, and it can't assume real-time assignments. Besides, we believe that the primary purpose for simulation is to unpack, identify malware packers prefers a more effective way. *A malware can damage the host as soon as it starts execution. Therefore, the most effective means is to detect and block the malware before executing [22].* Under these circumstances, a graph-based representation may be a natural way. Further, we discuss the relationships in semantics and construct a more advanced structure named CEG. Feature-based approaches encounter a slight modify may cause unpredictable results because they care more about features instead of the understanding of the program itself. Besides, it requires a large number of training samples and training time. However, CEG seeks to preserve the semantic relationship and logical structure, and our mining methods still retain the mentioned information.

The closest work to our research was conducted by Cesare *et al.* who proposed "Malwise" [23], a control-flow graph (CFG) based system for polymorphic malware classification. They apply the exact and approximate

[1] https://www.aldeid.com/wiki/PEiD
[2] https://exeinfo.atwebpages.com
[3] https://github.com/horsicq/Detect-It-Easy
[4] https://virustotal.github.io/yara
[5] https://bitblaze.cs.berkeley.edu/temu
[6] https://www.packerinspector.com

matching algorithm using string edit distance and string-based signature. Mapping graphs to strings or vectors will facilitate subsequent processing yet may lose considerable diagnostic information. The idea to build CEG is similar to JACKSTRAWS [24], which automatically construct graph templates to capture the core of different kinds of command and control (C&C) activities and we believe CEG can represent the discriminative and import behavior of malware packers. Different from building program graphs merely with Application Programming Interfaces (API) calls [25], [26], we choose opcode sequences as the vital information, because Runwal *et al.* [27] has demonstrated opcode similarity outperformed under plausible scenarios. Unlike other research, e.g. Egele *et al.* [5] transform CFG into fingerprints, Saleh *et al.* [28] convert CFG to a signature, Anderson *et al.* [29] transform code graph into Markov chains, we try to obtain a discriminative specification without changing graph itself. As the prior research on CoP [30] and INNEREXE [31] that employ symbolic formulas to determine semantics equivalence basic blocks (SEBB) and use Longest Common Subsequence (LCS) algorithm for path similarity comparison to detect systematic plagiarism. Our approach designs a new method to calculate block proximity considering the contextual relationship. With the graph kernel latest to malware field [32], we also adopt Weisfeiler-Lehman [33] shortest path graph kernel [34] to approximately estimate path similarity for our identification.

III. PROPOSED APPROACH

In this section, we will introduce our approach in detail which includes the architectures of CEG system, the definition of consistent execution, the strategy to construct CEG and the novel method to calculate similarity.

A. SYSTEM DESIGN

In this paper, for solving the conundrum of packer identification, we design a system as figure 1. Following our thoughts, we first generate CEGs and then develop comparator and updater to constitute our prototype system. To automate our workflow, the system will provide feedback to guarantee the accuracy and form a closed-loop system between model training and testing. Precisely, our CEG system consists of three major components.

1) **Constructor:** Constructor automatically generates the CEG of binary executables. It automatically disassembles the binary by disassembler, then optimizes the disassembly results for eliminating common obfuscation tricks. Following the principle of consistent execution, constructor slices and extracts relationships to construct CEG.
2) **Comparator:** Given an unknown CEG of packed malware, the comparator tries to identify it by comparing with a set of CEGs in reference collections. We propose a block proximity metric which takes current perception field into account, after that combine a most-weighted graph matching algorithm to obtain a pair of matching subgraphs. A pairwise graph similarity method will be applied to obtain the similarity score for classifying and identifying malware packers.
3) **Updater:** The role of this component is updating and improving our system continually. By integrating the result from different sources, for each malware packer, updater discusses and determines the standard to generate reference collections though employing the similarity metric mentioned in the comparator. Simultaneously, reference collections are used for classifying and will be dynamically promoted by feedback from intermediate results.

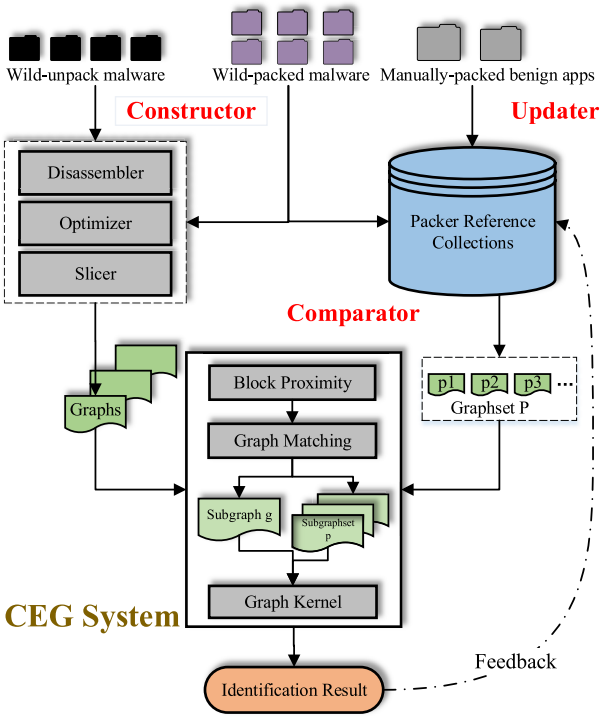

FIGURE 1. Architectures of Our CEG System.

B. CONSISTENT EXECUTION

Since we are trying to analyses the executive relationship in the graph representation, refer to the definition of program graph [22], we must preserve the functional information and semantics to facilitate unveiling executive intents. For characterizing the functions and semantics also ensuring independence and completion as far as possible, we define the consistent execution as follows.

Notations: Given an executable file F, let P be all the instructions of F, $s_i \in S$ be a set of attributes that influences the execution flow, E_{s_i} be the *executed instructions* under s_i. Therefore, we can conclude that $E_{s_i} \subseteq P$ and $\forall s_i, s_j \in S, E_{s_i} \neq E_{s_j}$.

Definition 1 (Consistent Execution): Let p_i,p_j be the instruction in P, respectively. Formally, we denote the instruction p_i and p_j is consistently executed (CE) if satisfy:

$$\forall s_i \in S, (p_i \in E_{s_i} \wedge p_j \in E_{s_i}) \vee (p_i \notin E_{s_i} \wedge p_j \notin E_{s_i}) = True$$

where s_i is the combination of factors including user inputs, program parameters, environments and other triggered elements which impact the flow in the course of execution.

Definition 2 (Consistently-Executing Block): Define a Consistently-Executing Block (CEB) B is a maximal sequences of consecutive instructions, $B = \{p_m, p_{m+1}, ..., p_{m+n}\} \subseteq P$ which satisfy the following properties:

when F runs to the instruction p_m:

1. $\forall p_i, p_j \in B \Rightarrow p_i, p_j$ is CE.
2. $\forall p_i \in P, p_i \notin B. \forall p_j \in B \Rightarrow p_i, p_j$ is not CE.

CEB has more than one entry points and exit points. Both the previous instruction of start p_{m-1} and stop instruction p_{m+n} of CEB are branch instructions.

Definition 3 (Consistently-Executing Graph): A CEG denoted by g, is a directed graph, represented as $g = (V, E, l_v, l_e)$, where V is the set of nodes and E is the set of directed links. l_v denotes a labeling function for nodes, i.e, $l_v : V \rightarrow \sum_v$ and l_e denotes another tagged function for links, i.e, $l_e : E \rightarrow \sum_e$. Where $\sum_v$ are the consistently-execution blocks, $\sum_e$ are the connecting branch instructions.

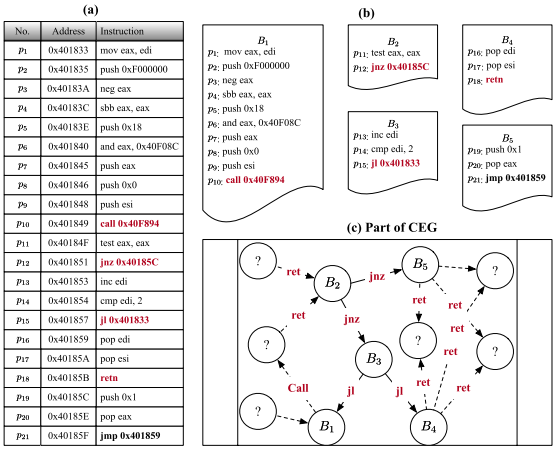

FIGURE 2. A Diagram of Consistent Execution. (a) Disassembler snippet. (b) CEB sketch.

For example, figure 2 illustrates a fragment dissembler of WananCry ransomware crypto worm which encrypted data to demand payments in bitcoin cryptocurrency. Subfigure 2(a) shows a dissembler snippet which has been compiled to x86-64 architecture and branch instructions marked red. In the case of CEB numbered as B_4 and B_5, when the execution flow jumps to the instruction p_{16}, evidently at that moment, p_{16}, p_{17}, p_{18} are pairwise CE. Equally, although *jmp 0X401859* is an unflowing instruction, it won't damage the consistency thus makes up $B_5 = \{p_{19}, p_{20}, p_{21}, p_{16}, p_{17}, p_{18}\}$ as shown in figure 2(b). Subfigure 3(c) is a part of CEG (containing five known blocks and the blocks labeled as a question mark express it doesn't appear in our case), each node stands for a single CEB which has more than one entry points and exit points, e.g. B_2 and B_3 has two exit points because conditional jump instruction *jl 0X401833* has two destinations. B_4, B_5 ends with return instruction *retn* may have more than one exit points which connect to the next instruction of different callers. Similarly, instruction *call 0X40F894* interrupts the sequential execution and program will jump to the objective procedure addressed 0X40F894, after that it will return at the next instruction p_{11} in B_2.

Algorithm 1 CEG Generating

Data: Procedures F with start address F_s
All instructions with address a and assembly i
All Branch instructions set I
Result: A CEG G with block set B and edge set E

```
C = 0 ;                          ▷ C is block number record
Function Slicer (a, cal):
    i = GetInsturction(a) ;
    cur = C ;
    if Alldone then
        return ; ▷ At the end of iterations
    end
    if Type(i) is in I then
        B ← b;
        E ← [cur, cal, i];
        C = C + 1 ;
    else
        b ← i ;                  ▷ b is a collection variable
    end
    T = CrossReference(i);
    forall the target addresses a_t in T do
        ▷ Now the cur will be next's cal
        Slicer(a_t,cur)
    end
for each f in F do
    Slicer(F_s, -1) ;    ▷ Iterate each function
end
G.Add(B,E) ;
```

C. CEG CONSTRUCTION

The concept of consistent execution is describing as above, after that a generating strategy is proposed based on rational cogitation upon a breakthrough to implement as well as utility. When given a packed malware sample in the form of executable binary, it is import to handle appropriately in the early stage so that we can extract as much semantic information as possible. First, we use the disassembler to disassemble it, then employ several heuristic methods to optimize disassembly results, including eliminating dead codes, removing unconditional jumps, modifying symmetric *nops* and so on.

Although the concept of consistent execution is standing from the perspective of behavior, there is an alternative way to construct CEG through static analysis instead of executing it. Our point is focusing on the branch instructions whose appearance lead to inconsistent on the executing flow. We traverse procedures to locate branch instructions, slice the instructions to divide boundaries, record invoking relationships to constitute flow paths.

As shown in algorithm 1, the input parameters all depend on the disassembly result of the analyzed executable file.

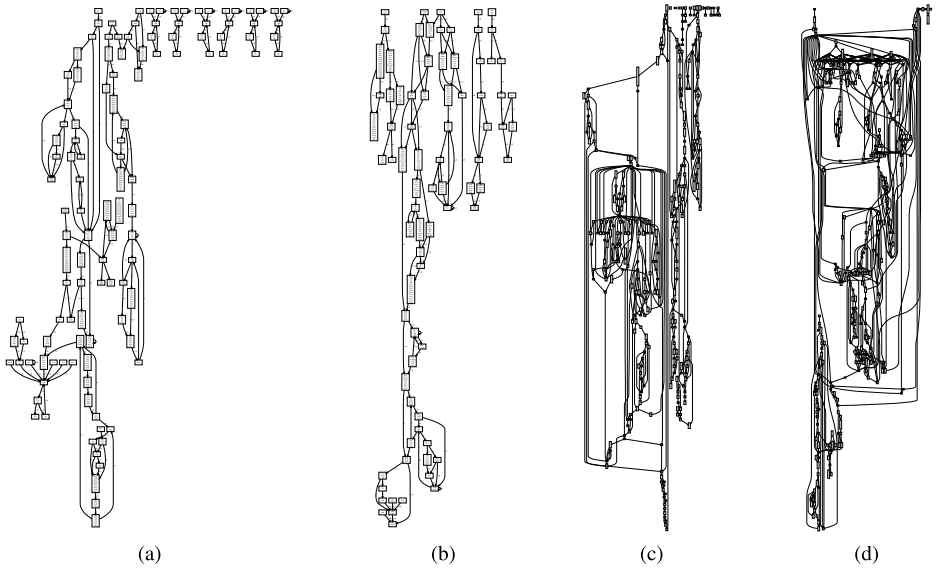

FIGURE 3. Four examples represented by CEG. (a) Nsapck. (b) Upack. (c) eXPressor. (d) PKLite32.

We recursively call the procedure **slicer** (Line 20) to obtain CEBs. In **slicer**, we traverse each instruction to determine it is a branch instruction or not (Line 8), and update blocks B (Line 9) and edges E (Line 10) at the same time. For each branch instruction, we repeat the procedure **slicer** beginning with its cross-references (Line 15) until traversal completed (Line 5). Finally, collected B and E composed CEG, and by choosing opcode sequences to express semantics, CEG can represent almost lossless semantics. As expressed in figure 3, the CEG of Nspack 3(a) and Upack 3(b) have a relatively simple structure, whereas others have extremely intricacy representations owing to their complicated packing algorithm.

D. CEG SIMILARITY COMPARISON

Given two CEGs, we intend to estimate the similarity between them. Because of the noise injected by adversaries, we shouldn't try to compute the exact similarity but consider comparing the significant constructions.

1) BLOCK PROXIMITY

Instead of a black-or-white metric, our method is estimating a proximity decimal between 0 and 1 and trying to accommodate subtle differences. Given a pair of objects which considered similar in a network, their similarity not just reflects on their close attributes in themselves, but their *neighbors* as well. Hence, the context of corresponding CEBs is supposed to be pretty similar if both try to achieve the same function. We first replace opcode sequences as labels to differentiate CEBs. Then express its sensing area in the form of the vector, and finally conclude a score using the Jaccard Similarity metric. As shown in figure 4, for corresponding nodes x, y, extracted $\vec{X}$ and $\vec{Y}$ and their ith order contextual CEBs set are X^i, Y^i. The proximity score S_{xy} (between 0 and 1) is computed

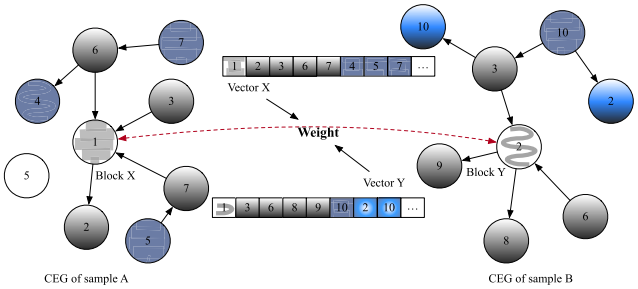

FIGURE 4. Our method to calculate block proximity.

through the following formula.

$$S_{xy} = \beta J(X^0, Y^0) + \beta^2 J(X^1, Y^1) + \ldots + \beta^{n+1} J(X^n, Y^n)$$

where J stands for jaccard similarity coefficient and β is a adjusting parameter between 0 and 1, which related to the highest order n.

2) GRAPH MATCHING

Before a pairwise comparing measurement, our intuition is trying to extract the essential structure for comparison so that we need an approximate matching method to achieve it. At the block level, we prefer to select the closest blocks (highest similarity score). However, it may be a little biased at the graph level, because the matching blocks may not be the closet one but weighting ensemble. As figure 4 shows, we treat block proximity score as the weight of edges between corresponding blocks and generate a weighted bipartite between CEGs. Hence, we employ the most-weighted bipartite matching Kuhn-Munkres algorithm to determine the structures and extract the critical matching subgraphs. Finally, we apply a series of pruning techniques to remove isolated nodes, restore associations for attenuating the impact of noise. To this

extent, a part of malware packers can be easily distinguished after subsequent optimizations of graph matching.

3) GRAPH KERNEL-BASED SIMILARITY METRIC

The suspicious behavior express on the disassembly level is executing a series of instruction sequences, which were considered as *path* in CEG. Path similarity comparison can adapt through pairwise comparing all paths to yield kernel, yet the problem is NP-hard. As an approximate algorithm, shortest-paths kernel [34] determine all shortest paths to calculate as follows.

$$K_{\text{shortestpath}}(G, G') = \sum_{s \in SD(G)} \sum_{s' \in SD(G')} k(s, s')$$

where $SD(G)$ and $SD(G')$ are shortest distances in graph G and graph G'.

Taking into account the block labels and relatively low efficiency in computation, we employ the Weisfeiler-Lehman graph kernel (WLK) for optimization. WLK computes the similarities based on the 1-dimensional WL test of graph isomorphism [33]. Let k be any graph kernel named base kernel. Then WLK with h iterations between two graphs G and G' is defined as

$$K_{WL}(G, G') = k(G_0, G'_0) + k(G_1, G'_1) + \ldots + k(G_h, G'_h)$$

where $\{G_0, G_1, \ldots, G_h\}$ and $\{G'_0, G'_1, \ldots, G'_h\}$ are the Weisfeiler-Lehman sequences of G and G'. It is clear that the shortest-path kernel with block labels can take advantage of the Weisfeiler-Lehman framework to compare graphs based on the whole Weisfeiler-Lehman sequence. Comprehensive applying both graph kernels would be capable of measuring path similarity more efficiently.

IV. EXPERIMENTS

A. DATASETS AND EXPERIMENT SETUP

Our experiments have been carried out on three different datasets named manually-packed benign app set, wild-packed malware set and wild-unpacked malware set. The first set is manually constructed by packing 10 benign executables in Windows7 Ultimate (x64), including "calc.exe", "notepad.exe", "find.exe", "dir.exe" and so on. We focus on 15 different packers, and it is noteworthy that most of them have several different versions.

All the samples used in the last two datasets prepared through free resource site VirusShare[7] and VXHeaven.[8] Each sample is scanned by VirusTotal[9] which utilises three packer scanners Command, F-PROT and PEiD. The former packed malware dataset collected the packed sample which assigned as the same packer identified by all three scanners. Similarly, The latter unpacked malware dataset collected if a packed sample is all assigned as unpacked. It is a large-scale dataset

[7]https://virusshare.com
[8]https://vxheaven.org
[9]https://virustotal.com

TABLE 1. The details of experimental datasets.

No.	Packers	Versions	Numbers
1	UPX	v1.00, v1.25, v2.00, v3.00	8035
2	Themida	1.0.0.8, 1.2.0.1, and 1.5.0.0	97
3	Aspack	2.12, 2.12b and 2.29	545
4	Nspack	2.9, 3.4, 3.7 and 4.1	889
5	FSG	1.33 and 2.0	1200
6	ORiEN	v1.03, v2.11 and v2.12	135
7	Upack	0.32, 0.36, 0.39 and 0.399	3192
8	RCtyptor	1.1, 1.5 and 1.6	116
9	tElock	0.9x and 2.00	440
10	Thinstal	v2.427 and v3.348	136
11	MoleBox	2.36 and 2.65	163
12	PEArmor	v0.74 and v0.765	106
13	PKLite32	v1.1 and v2.2	628
14	eXPressor	v1.4.5.1	177
15	BeRo	1.0	139
	Benign		380
	Packed		15998
	Unpack		39692
	Total		**56070**

contains about 40K unpacked malware samples. The details of our datasets are presented in table 1.

The front-end of our system is based on IDA Pro disassembler and all the implementations of graph kernel is based on the framework Grakel [35]. Our experiments were performed on a Linux machine with an Intel Core I7-8700K CPU (3.7GHz) and 32GB RAM. In our experiments, we select a certain number of representative CEGs from manually-packed executable to reference collections by measuring both aggregation and dispersion between packers. This estimation will continue to provide positive feedback following the progress of our experiments.

B. EVALUATION ON MANUALLY-PACKED BENIGN APPS

In this set of experiments, we evaluate the classification performance beyond different packers and different versions of the same packer and try to prove the effect of CEG graph matching.

Because the first dataset is manually packed with general packers, we can confirm the packing consequence is 100% precise. We construct CEGs and compare them to detect the performances of graph matching, inspecting the matching capability to extract critical structures. The similarity score shows in Table 2. Each column, thickened portion is the score of using graph matching method, others are the scores by directly comparing, the digitals marked red are the score of the average similarity for the same packer of different versions.

The results are shown in figure 5, the coordinates of the x-axis represents the number of packers as mentioned in table 1. We can make intuitive sense that different packers have significant discrimination in the view of CEG, and the difference between versions is subtle that can be ignored. Graph matching leads us to pay more attention to the vital structure of CEG, it grabs, and zoom on the significant subgraphs. As shown in 5(a), we plot the average, maximum and

TABLE 2. The similarly scores between different packers and different versions of the same packer.

Packers	UPX		Themida		Aspack		Nspack		FSG		ORiEN		Upack		RCryptor		tElock		Thinstal		MoleBox	
UPX	0.83	**0.97**	0.09	**0.15**	0.28	**0.27**	0.01	**0.02**	0.10	**0.08**	0.00	**0.03**	0.00	**0.00**	0.01	**0.00**	0.00	**0.00**	0.00	**0.00**	0.10	**0.09**
Themdia	0.09	**0.15**	0.99	**0.99**	0.15	**0.16**	0.01	**0.00**	0.01	**0.02**	0.00	**0.00**	0.02	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.01	**0.00**
Aspack	0.28	**0.27**	0.15	**0.16**	0.91	**0.95**	0.01	**0.07**	0.07	**0.00**	0.09	**0.02**	0.01	**0.00**	0.08	**0.01**	0.00	**0.00**	0.00	**0.00**	0.10	**0.07**
Nspack	0.01	**0.02**	0.01	**0.00**	0.01	**0.07**	0.98	**1.00**	0.03	**0.06**	0.00	**0.00**	0.01	**0.00**	0.05	**0.00**	0.00	**0.00**	0.00	**0.00**	0.10	**0.16**
FSG	0.10	**0.08**	0.01	**0.02**	0.07	**0.00**	0.03	**0.06**	0.87	**0.92**	0.04	**0.00**	0.00	**0.00**	0.01	**0.00**	0.00	**0.00**	0.00	**0.00**	0.01	**0.00**
ORiEN	0.00	**0.03**	0.00	**0.00**	0.09	**0.02**	0.00	**0.00**	0.04	**0.00**	0.99	**0.99**	0.00	**0.00**	0.01	**0.00**	0.05	**0.00**	0.00	**0.00**	0.00	**0.00**
Upack	0.00	**0.00**	0.02	**0.00**	0.01	**0.00**	0.01	**0.00**	0.00	**0.00**	0.00	**0.00**	0.89	**0.93**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.08	**0.06**
RCryptor	0.01	**0.00**	0.00	**0.00**	0.08	**0.01**	0.05	**0.00**	0.01	**0.00**	0.01	**0.00**	0.00	**0.00**	0.98	**0.98**	0.01	**0.00**	0.01	**0.00**	0.00	**0.00**
tElock	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.05	**0.00**	0.00	**0.00**	0.01	**0.00**	0.72	**0.87**	0.00	**0.00**	0.00	**0.00**
Thinstal	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.01	**0.00**	0.00	**0.00**	1.00	**1.00**	0.00	**0.00**
MoleBox	0.10	**0.09**	0.01	**0.00**	0.10	**0.07**	0.10	**0.16**	0.01	**0.00**	0.00	**0.00**	0.08	**0.06**	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**	0.94	**0.98**

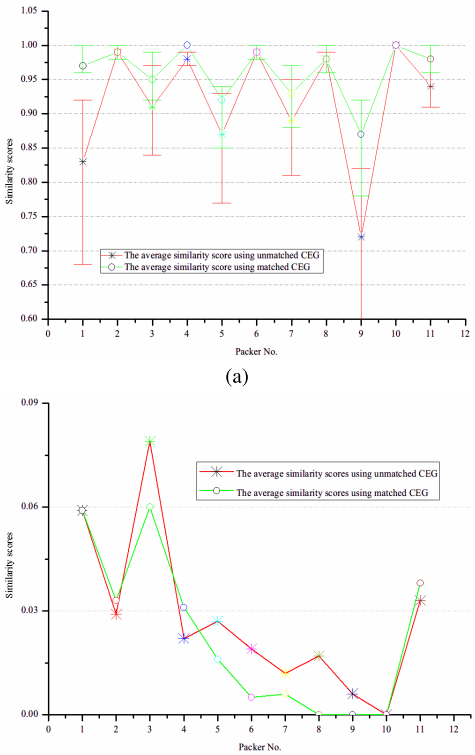

FIGURE 5. The Similarity Between Matched and Unmatched Metric. (a) The Similarity Scores Between Different Packers. (b) The Similarity Scores Between Different Versions.

minimum similarities together on Y-axis, the graph matching method performs better both on accuracy and stability. Meanwhile, focusing on the essential structure may cause slight instability because the scale of graph comparison reduced as illustrated. In figure 5(b), polyline intersections explained that graph matching might cause minor fluctuations to obtain a separately high similarity.

However, small fluctuations caused by graph matching will not affect the outcomes but significantly save time overhead because large-scale graph comparison is computation expensive. In figure 6, we counted the number of nodes and edges to represent the scales before and after graph matching. Though those obvious gaps, we can conclude that graph matching can doubly shrink the figure size which saving the overhead on time-consuming operations, like graph searching and traversing. The study demonstrates that CEG representation

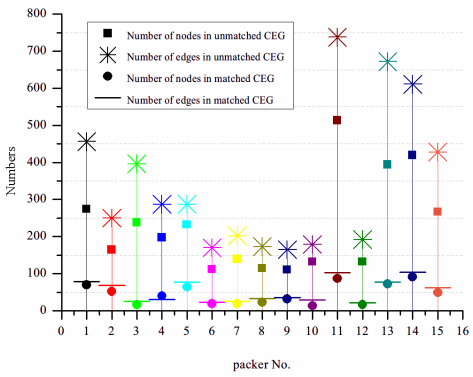

FIGURE 6. The Scale of Matched and Unmatched CEG.

can reduce the workload of feature engineering and graph matching can necessity boost the overall performance of our system.

C. EVALUATION ON WILD-PACKED SAMPLE COLLECTIONS

In this experiment, since we employ the discriminant result from VirusTotal as our ground truth, we can check out our accuracy based on the wild sample collection of real-world packed malware set.

For each malware packer identifier, it can be regarded as a simple binary classification problem, and we systematically evaluate the performance of each packer using dichotomous evaluation criteria, including Precision (*Pre*), Recall (*Rec*), Force Positive Rate (*FPR*), Accuracy (*ACC*) and $F1$. These evaluation indicators are often used to evaluate the classification model in machine learning field.

The measures are shown as follows.

$$Pre = \frac{TP}{TP + FP} \quad Rec = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN}$$

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \quad F1 = \frac{2 * Pre * Rec}{Pre + Rec}$$

where TP is the quantity of correctly identified, TN is the quantity of correctly not identified, FP is the quantity of mistakenly identified, FN is the quantity of incorrectly not identified.

From Table 3, The average precision of identification is up to impressive 96% which explain our approach is still feasible for practical use. We also further evaluate the identification performance that the accuracy of UPX, Thinstal, BeRo is pretty high, whereas FSG, eXPressor, MoleBox have

TABLE 3. Detailed accuracy using wild-packed Malware dataset.

No.	Packers	Pre	Rec	FRP	ACC	$F1$
1	UPX	99.6%	98.8%	0.32%	99.2%	0.992
2	Themida	96.0%	98.9%	0.02%	99.9%	0.975
3	Aspack	97.5%	92.3%	0.09%	99.6%	0.948
4	Nspack	98.7%	100%	0.07%	99.9%	0.994
5	FSG	89.1%	99.8%	0.91%	99.1%	0.941
6	ORiEN	97.8%	98.5%	0.01%	99.9%	0.982
7	Upack	98.7%	95.6%	0.31%	98.8%	0.972
8	RCtyptor	97.3%	93.1%	0.01%	99.9%	0.952
9	tElock	98.2%	100%	0.05%	99.9%	0.991
10	Thinstal	99.5%	100%	0.03%	99.9%	0.998
11	MoleBox	92.0%	98.8%	0.10%	99.9%	0.953
12	PEArmor	97.2%	100%	0.02%	99.9%	0.986
13	PKLite32	96.5%	100%	0.02%	99.9%	0.982
14	eXPressor	91.0%	97.2%	0.11%	99.9%	0.940
15	BeRo	99.2%	98.6%	0.01%	99.9%	0.990
	average	**96.5%**	**98.1%**	**0.13%**	**99.7%**	**0.973**

the minimum precision of about 90%. The poor result may be related to their complex packing algorithm generating a relatively large CEG, refer to figure 6. Equally, in figure 3(c) and 3(d), we confirm that the packer eXPressor and PKLite32 represented indeed a surprisingly complicated structure.

Overall, *FPR*s are all relatively low explains that the possibility of misidentification is low. However, the more information embedded in the CEG may cause more elasticity when comparing, which results in higher FP (false positive) rates. Meanwhile, less in the quantity may magnify the final consequence.

D. EVALUATION ON DISCOVERING ERROR-LABELLED OR UNLABELED PACKED MALWARE

In this experiment, we test on a large-scale unpack malware dataset contains 34692 samples. Through the result and our manually investigate, there are 59 suspicious samples which are not detected among the scanners when we set the similarity threshold up to 0.4, and the number is 13 if the threshold is 0.6. At the same time, we also found separate samples that may be misclassified or incorrect recognition result.

Our system detected some error-labelled samples, e.g. malware 28c533e84e679497af749c117db83007 which be identified as PEBundle among the scanner PEiD, VirusTotal, Exeinfo. However, the similarity is up to 0.66 when compared with UPX in our system. Therefore, we manually investigate and discovered that although the gap between their scale of CEG is vast, the blocks matched by graph matching are concentrate on a similar subgraph. As shown in figure 7 it illustrated the malware is most likely packed by UPX but not deny PEBundle. Moreover, because of its section name ''pebundle'' and several executing characteristics, we can make sure that it be multi-layer packed by both UPX and PEBundle. The result suggests that our approach can be an essential reference or supplement to the existing achievements.

In other words, our approach also detected unlabeled packed malware missed by other scanners. E.g. malware

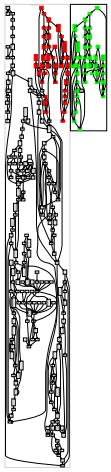

FIGURE 7. Error Labelled Example.

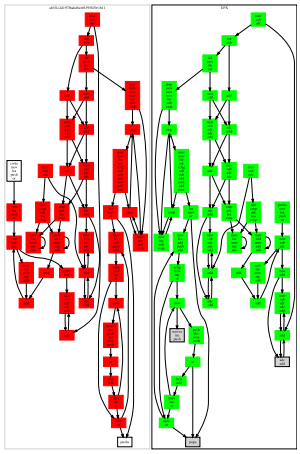

FIGURE 8. Unlabelled Example.

ab3f1c2d197fbabd8cef6395829e1b11 scores 0.73 similar to UPX while report unpacked by PEiD, VirusTotal, Exeinfo and CFF Explorer. Though our exhaustive analysis, it modified the instruction based on the common prefix of code packed, which resulted in invalid signature matching which misleads scanners. Even though the binary signature has been modified deliberately, the basic structure of packer cannot be erased, as illustrated in figure 8. In addition, advanced scanner DIE and state of art inspector Deep Packer Inspector (DPI) support our results. This experiment demonstrates our approach is robustness to the modification of conventional obfuscation techniques.

V. CONCLUSION AND FEATURE WORK

This paper proposed a new approach based on CEG mining to identify malware packers. We discoursed the concept of consistent execution and constructed CEG to characterize the behavior of packers. Moreover, we design a novel similarity schema for measuring the pairwise similarity between two CEGs. Primarily, on the foundation of node proximity metric, the usage of graph matching assists us to focus on the critical structure thus remarkably enhance the accuracy as well as

effectiveness. The evaluation of experiments is good and reflect the potential of CEG. Later we may widely apply it on malware analysis or even program analysis soon.

The dilemma of our approach is reflexively counting on the disassembly effect, unable to cope with advanced polymorphic and metamorphic techniques like virtual machine protectors. The emergence of those innovative obfuscation technologies poses greater threats and challenges to us analyzer and urge us to enrich the connotation of CEG or other methods. Furthermore, our approach is a little time consuming and the pairwise comparison is either insufficient or inapplicable in the large-scale dataset, so transform pairwise similarity problem into a similarity searching problem is worthy of further investigation.

REFERENCES

[1] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Secur. Privacy*, vol. 6, no. 5, pp. 65–69, Sep./Oct. 2008.
[2] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques," *Comput. Sci. Rev.*, vol. 32, pp. 1–23, May 2019.
[3] D. Ucci, L. Aniello, and R. Baldoni. (2017). "Survey of machine learning techniques for malware analysis." [Online]. Available: https://arxiv.org/abs/1710.08189
[4] L. Sun, S. Versteeg, S. Boztaş, and T. Yann, "Pattern recognition techniques for the classification of malware packers," in *Proc. Australas. Conf. Inf. Secur. Privacy*. Berlin, Germany: Springer, 2010, pp. 370–390.
[5] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, Feb. 2012, Art. no. 6.
[6] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 659–673.
[7] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding Linux malware," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 161–175.
[8] Y. Duan *et al.*, "Things you May not know about Android (un) packers: A systematic study based on whole-system emulation," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, Feb. 2018, pp. 18–21.
[9] G. Vigna and D. Balzarotti, "When malware is packing heat," in *Proc. ENIGMA*, 2018, pp. 1–34.
[10] M. Bat-Erdene, H. Park, H. Li, H. Lee, and M.-S. Choi, "Entropy analysis to classify unknown packing algorithms for malware detection," *Int. J. Inf. Secur.*, vol. 16, no. 3, pp. 227–248, Jun. 2017.
[11] N. M. Hai, M. Ogawa, and Q. T. Tho, "Packer identification based on metadata signature," in *Proc. 7th Softw. Secur., Protection, Reverse Eng./Softw. Secur. Protection Workshop*, Dec. 2017, Art. no. 5.
[12] B. Cheng *et al.*, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 395–411.
[13] J. Zhang, K. Zhang, Z. Qin, H. Yin, and Q. Wu, "Sensitive system calls based packed malware variants detection using principal component initialized multilayers neural networks," *Cybersecurity*, vol. 1, no. 1, p. 10, Dec. 2018.
[14] Y. Kawakoya, M. Iwamura, and M. Itoh, "Memory behavior-based automatic malware unpacking in stealth debugging environment," in *Proc. 5th Int. Conf. Malicious Unwanted Softw.*, Oct. 2010, pp. 39–46.
[15] M. Bat-Erdene, T. Kim, H. Park, and H. Lee, "Packer detection for multilayer executables using entropy analysis," *Entropy*, vol. 19, no. 3, p. 125, Mar. 2017.
[16] S. Aicardi, "Packer-complexity analysis in panda," Ph.D. dissertation, Politec. Torino, Turin, Italy, 2018.
[17] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 745–756.
[18] M. Saleh, E. P. Ratazzi, and S. Xu, "Instructions-based detection of sophisticated obfuscation and packing," in *Proc. IEEE Mil. Commun. Conf.*, Oct. 2014, pp. 1–6.
[19] K. Kancherla, J. Donahue, and S. Mukkamala, "Packer identification using byte plot and Markov plot," *J. Comput. Virology Hacking Techn.*, vol. 12, no. 2, pp. 101–111, May 2016.
[20] C. Lim, Suryadi, K. Ramli, and Y. S. Kotualubunb, "Mal-flux: Rendering hidden code of packed binary executable," *Digit. Invest.*, vol. 28, pp. 83–95, Mar. 2019.
[21] T. Ban, R. Isawa, S. Guo, D. Inoue, and K. Nakao, "Application of string kernel based support vector machine for malware packer identification," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Aug. 2013, pp. 1–8.
[22] M. S. Islam, M. R. Islam, A. S. M. Kayes, C. Liu, and I. Altas, "A survey on mining program-graph features for malware analysis," in *Proc. Int. Conf. Secur. Privacy Commun. Netw.* Cham, Switzerland: Springer, 2014, pp. 220–236.
[23] S. Cesare, Y. Xiang, and W. Zhou, "Malwise&#× an effective and efficient classification system for packed and polymorphic malware," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1193–1206, Jun. 2013.
[24] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "Jackstraws: Picking command and control connections from bot traffic," in *Proc. USENIX Secur. Symp.*, San Francisco, CA, USA, 2011, pp. 1–16.
[25] M. Eskandari and S. Hashemi, "A graph mining approach for detecting unknown malwares," *J. Vis. Lang. Comput.*, vol. 23, no. 3, pp. 154–162, Jun. 2012.
[26] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer, "Extraction of statistically significant malware behaviors," in *Proc. 29th Annu. Comput. Secur. Appl. Conf.*, Dec. 2013, pp. 69–78.
[27] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *J. Comput. Virol.*, vol. 8, nos. 1–2, pp. 37–52, 2012.
[28] M. Saleh, E. P. Ratazzi, and S. Xu, "A control flow graph-based signature for packer identification," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Oct. 2017, pp. 683–688.
[29] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *J. Comput. Virol.*, vol. 7, no. 4, pp. 247–258, 2011.
[30] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 389–400.
[31] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng. (2018). "Neural machine translation inspired binary code similarity comparison beyond function pairs." [Online]. Available: https://arxiv.org/abs/1808.04706
[32] A. Narayanan, G. Meng, L. Yang, J. Liu, and L. Chen. (2016). "Contextual weisfeiler-lehman graph kernel for malware detection." [Online]. Available: https://arxiv.org/abs/1606.06369
[33] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *J. Mach. Learn. Res.*, vol. 12, pp. 2539–2561, Sep. 2011.
[34] K. M. Borgwardt and H. P. Kriegel, "Shortest-path kernels on graphs," in *Proc. 5th IEEE Int. Conf. Data Mining (ICDM)*, Nov. 2005, p. 8.
[35] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis. (2018). "GraKeL: A graph kernel library in python." [Online]. Available: https://arxiv.org/abs/1806.02193

XINGWEI LI is currently pursuing the M.S. degree in computer science from the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His recent research interests include program analysis, cybersecurity, and machine learning.

ZHENG SHAN is the Director of the Department of High Performance Computing, State Key Laboratory of Mathematical Engineering and Advanced Computing, China, where he is also a Doctor, a Doctoral Supervisor, and an Associate Professor. His research interests include cybersecurity and high-performance computing.

FUDONG LIU is currently a Lecturer in computer science with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His recent research interests include advanced computing, machine learning, and cybersecurity.

YIHANG CHEN is currently pursuing the M.S. degree in computer science with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His recent research interests include cybersecurity and artificial intelligence.

YIFAN HOU is currently a Lecturer in computer science with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. Her recent research interests include the operating systems and high-performance computing.