IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 1, 2019, accepted March 21, 2019, date of publication May 9, 2019, date of current version June 13, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2914031

Malware Clustering Using Family Dependency Graph

BINLIN CHENG[1,2], QIANG TONG[1], JIANHONG WANG[1], AND WENHUI TIAN[1]
[1]College of Computer and Information Engineering, Hubei Normal University, Huangshi 435002, China
[2]College of Arts and Science, Hubei Normal University, Huangshi 435002, China

Corresponding author: Qiang Tong (tongqiang@hbnu.edu.cn)

This work was supported in part by the Natural Science Foundation of Hubei Province of China under Grant 2017CFB307, Grant 2018CFB550, and in part by the Talent Introduction Project of Hubei Normal University in 2017, College of Arts and Science, Hubei Normal University, under Research Project Ky201803.

ABSTRACT Malware brings a major security threat on the Internet today. It is not surprising that much research has concentrated on detecting malware. Unfortunately, the current malware detection approaches suffer from ineffective detection of new malware samples. These models effectively identify the known malware samples but not new variants. To address this issue, we propose a novel malware detection approach based on the family graph. First, we trace the API calls of the monitored application, and then we generate the dependency graph based on the dependency relationship of the API calls. At last, we construct the family dependency graph via clustering the graphs of a known malware family. In this way, we can determine whether a new sample belongs to a known malware family. The evaluation results show that our approach is effective with small overhead compared to other existing approaches.

INDEX TERMS Malware, dynamic analysis, API call.

I. INTRODUCTION

Malware is a major security problems on the Internet. Since there are several thousand new malware variants per day. The most challenging problem is how to effectively identify new malware variants.

Dynamic analysis method are promising solutions to the problem of new variants detection as they do not rely on static signature. These methods execute samples and monitor their API call semantics. behaviors [1], [2]. However, current dynamic detection approaches have significant limitation. That is, these approaches only generate the feature from one instance, making them low detection accuracy with new variants. Malware can easily evolve itself to new variant to erase its feature. For example, new variant can obfuscate its API call information through API reordering, injecting, or alternating attacks [3], [4].

Since the API call information extracted from one instance is easily tampered by evolved variants, we aim to answer the question:" *can we design a malware detection approach which does not rely on the feature of only one instance but a clustering of a family?*" If so, it can help us to improve the efficiency of malware detection and handle newly evolved variants.

In this paper, we propose CuF, a family dependency graph based malware detection approach. Unlike previous approaches that extract the feature from one instance, we cluster the dependency graph of different variants of a malware family into the family graph. Compared to previous approaches, our approach is more resilient against the new variants.

Contributions:

The main contributions of this paper are as follows:

- We present a novel, family dependency graph based approach to profile a malware family's feature. Our approach can effectively identify the new variants of a malware family.
- We have implemented a prototype of CuF;
- We have evaluated CuF on a known datasets. The results demonstrate that our approach outperforms previous approaches in terms of better effectiveness and efficiency.

The rest of the paper is organized as follows. Section II presents the goals of this paper. Section III describes the our approach. Section IV provides the implementation of our approach. Section V presents our experimental

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

results. Section VI gives the related works. Section VII concludes.

II. GOALS

CuF aims to detect new malware using family dependency graph. Our design has the following main goals:

- **Effectiveness** CuF should detect malware effectively (low false positives), while limiting the benign software (low false negatives).
- **Efficiency** CuF should have a low performance overhead. It should be feasible to deploy our approach to detect malware in practice.
- **Robustness** CuF should be robust to various attack strategies. We assume that the attacker knows about the CuF techniques and will try to avoid detection.

III. OUR APPROACH

This section provides our approach. First we introduce the concept of API call. Then we describe the API call dependency graph. Next, we propose our novel concept, that is, family dependency graph. At last, we propose a family dependency graph based approach to detect malware.

A. API CALL

API is a set of application programming interfaces available in the operating systems to interact with it.[1] The information of API call traced from an application can be used to represent it's behavior. Existing dynamic analysis methods usually use API call sequences to represent the program behavior [5]–[7].

B. API CALL DEPENDENCY GRAPH

In addition to the API call sequences, the dependency relationships between API calls also are useful to profile the targeted application. Therefore, much work use API call dependency graph instead of API call sequences to represent the semantic of application. For instance, Mihai et al. [8] define two types of API call dependency relationships in their work:

1) **Def-Use Dependency:** Mihai et al. [8] create a dependency edge between two API calls when the subsequent API call has an argument with both same type and value to one argument of the previous API call;
2) **Substring Dependency:** For each string-valued arument, Mihai et al. [8] compare its value with the successor's. If two values share a substring, then Mihai et al. create a dependency edge from the first API call to the successor.

For a program, there exists many dependency relationships between its API calls. Dependency graph can be used to represent these dependency relationships. In this graph, an edge from node x to y indicates that there is a dependency relationship from API call x to y. We borrow the definition of API call dependency graph in the Mihai et al's work [8]:

[1]https://en.wikipedia.org/wiki/Windows_API

Definition 1: **API Call Dependency Graph** = (N, E), where:

- N is the nodes set, each representing an API call;
- E is the edges set, $E \subseteq V \times V$, each representing a dependency relationship between API calls.

C. FAMILY DEPENDENCY GRAPH

We have illustrated the concept of API call dependency graph. However, the graph only profile only one malware instance but not other instance in the same malware family, which severely limits the adoption of our approach. To detect various instances from a malware family, we need to cluster multiple dependency graphs of these instances in a malware family into a common dependency graph.

Let $G = \{g_i, \ldots, g_n\}$ be a set of n graphs constructed from n variants in a malware family. To cluster these graphs, we use existing graphs clustering methods [9]. We borrow two definitions of *Minimum Common Supergraph (MinSuper)* (definition 2) and *Maximum Common Subgraph (MaxSub)* (definition 3) from bunke et al. [9].

Definition 2 (Minimum Common Supergraph, MinSuper): We use $g_i = (N_i, E_i)$ and $g_j = (N_j, E_j)$ denote two graphs. If there exist supergraph from g to g_i and from g to g_j, we call g is a *common supergraph* of g_i and g_j. In a further, if there exists no other common supergraph of g_i and g_j has fewer nodes than g, we call g a *minimum common supergraph* of g_i and g_j.

Definition 3 (Maximum Common Subgraph, MaxSub): We use $g_i = (N_i, E_i)$ and $g_j = (N_j, E_j)$ denote two graphs. If there exist subgraph from g to g_i and from g to g_j, we call g is a *common subgraph* of g_i and g_j. In a further, if there exists no other common subgraph of g_i and g_j has more nodes than g, we call g a *maximum common subgraph* of g_i and g_j.

$$\begin{aligned} MinSuper\left(g_i, g_j\right) = MaxSub\left(g_i, g_j\right) \\ \bigcup \left(g_i - MaxSub\left(g_i, g_j\right)\right) \\ \bigcup \left(g_j - MaxSub\left(g_i, g_j\right)\right) \end{aligned} \quad (1)$$

The MaxSub algorithm is defined in Bunke and Shearer [10], and the MinSuper between two dependency graphs is generated by Eq.(1) [11]. Both MinSuper and MaxSub are original defined between two graphs, but they can also be extended to more graphs. In that case, the Eq.(1) is repeatedly computed until cover all the graphs in the set. In this way, we can use the combination of MinSuper and MaxSub to represent of *family dependency graph* for a malware family.

D. FAMILY DEPENDENCY GRAPH BASED MALWARE DETECTION

Having generated the family dependency graph from a malware family, we can use it to detect the new malware variants from this family. First, we generate an API call dependency graph, called g_{new}, for an instance suspected of being malicious. The g_{new} is compared with the family dependency

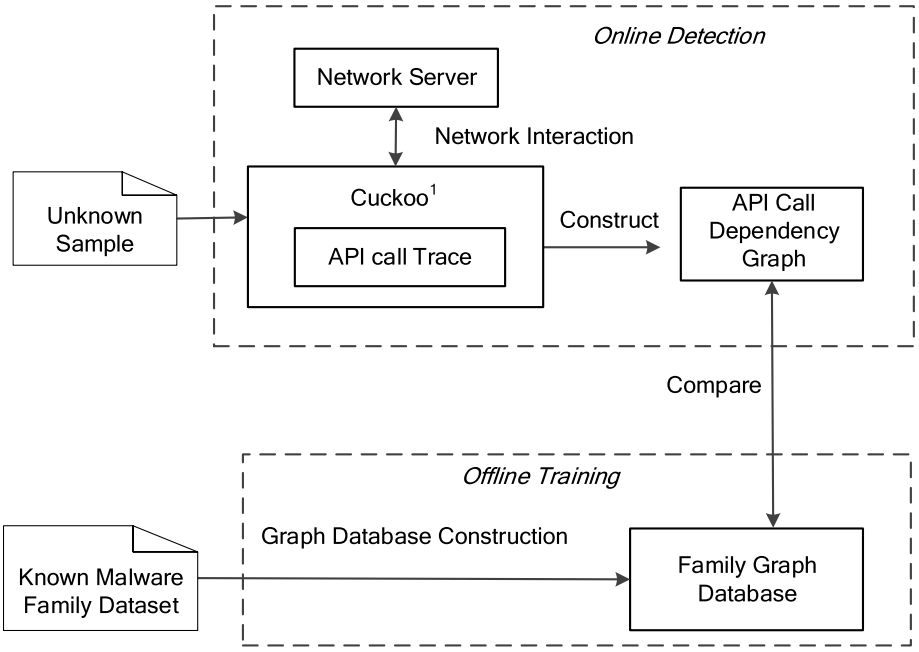

FIGURE 1. The architecture of CuF.

graph (both MaxSub and MinSuper) for each malware family. We can determine whether a new various belongs to an existing malware family according to Eq.(2). That is, the g_{new} should include the MaxSub. Meanwhile, MinSuper should include g_{new}.

$$D = MaxSub \subseteqq g_{new} \wedge g_{new} \subseteqq MinSuper \tag{2}$$

IV. IMPLEMENTATION ARCHITECTURE

As shown in Figure 1, CuF consists of offline detection stage and online training stage.

CuF's offline stage is based on GMT [12], an open source graph matching toolkit, with 1, 865 lines of code. We cannot directly use this graph matching tool because it does not support family dependency graph construction. To address this issue, we enhance the GMT to support minimum common supergraph (MinSuper) and maximum common subgraph (MaxSub) construction.

CuF's online stage is developed on the top of Cuckoo,[2] an open-source malware sandbox framework, containing 545 lines of code in C. We customize Cuckoo by replacing its API call hooking module with our module. The custom Cuckoo VirtualBox[3] virtual machine is running to trace the API calls, construct the dependency graph. After that, we compare this the dependency graph to the family graph database, and identify this new sample according to the Eq.(2) (see section III-D).

A. SELECTIVE API CALLS

CuF's online module extends Cuckoo sandbox to hook selective Windows APIs. Now CuF records 205 Windows API calls in total. The selective APIs are chosen by the following steps.

First, our goal is to construct the API call dependency graph to describe the malware behaviors, so we concentrate on the API calls which can accomplish the malicious intends to reduce the unnecessary monitor. To this end, we make an in-depth study of Windows API calls, and find that totally 204 API calls are needed to monitor.

Second, Windows network operations are all through a universal API call, that is `NtDeviceIoControlFile`. We can identify network-related operations of `NtDeviceIoControlFile` by checking its arguments. Recall that we have monitored 204 API calls in the first step, and therefore we records 205 Windows API calls in total.

B. NETWORK SERVER

Another problem is that some malware samples need to interact with network. Otherwise, they will terminate their execution. To address this issue, we use two instances of Virtualbox in our online detection phrase. The first one runs Windows, and acts as a victim machine requiring necessary network operation. The second one provide network services to the victim machine (as shown in Figure 1). In this way, CuF can monitor the behavior of malware samples with network interaction.

V. EVALUATION

In this section, we evaluate CuF. In Particular, we are very interesting with the following questions.

1) Q1: Can CuF be effective to detect new malware variant? (*effectiveness*)

[2]https://cuckoosandbox.org/
[3]https://www.virtualbox.org/

2) Q2: How many false positives does CuF incur? (*effectiveness*)
3) Q3: How much overhead does CuF introduce? (*efficiency*)

A. DATASET

As showed in Table 1, we use two data sets to evaluate our approach: Malware set and Benign set.

TABLE 1. Datasets.

Type	#Sample
Malware set	300
Benign set	3546

Malware set consists of the 6 popular malware families (shown in Table 2), which are obtained from Anubis [13]. These malware families were also used for the evaluation in the works of Kolbitsch [14] and Park [15]. Many of the samples use the obfuscation technique to generate new variants to evade existing API call based detection approaches.

TABLE 2. Malware set.

Malware Family	Type	#Samples
Allaple	Exploit-based	50
Bagle	Mass-mailing	50
Mytob	Mass-mailing	50
Agent	Trojan	50
Netsky	Mass-mailing	50
Mydoom	Mass-mailing	50
Total		300

The benign sets consist of 3546 popular applications on the Windows OS, such as Explorer, Calc, Notepad, WinRAR, QQ, Adobe, IE Browser, Firefox and so on. These samples were scanned by the Kaspersky anti-virus and confirmed security.

B. ANSWER TO Q1: DETECTION ACCURACY

We evaluate the detection accuracy of CuF with other two representative API call based detection approaches: Kolbitsch [14] and Park [15]. This evaluation is conducted on the malware dataset described in Table 2. To validate the detection accuracy of CuF, we perform our evaluation using 10-fold cross-validation [16]. For each of our six malware families, samples are randomly divided into 10 subsets. 9 subsets are used as known malware family dataset to the offline training, and the 1 subset is used as unknown variants to the online detection (see Figure1). Next, we compare the the API call dependency graph from the online detection and family graph database in the offline training. Once the Eq.(2) is matched, we can determine the new variants belongs to a known malware family.

The results are shown in Table 3. We can see that the detection accuracy of CuF is highest with an average value 0.92.

TABLE 3. Comparative evaluation of detection accuracy.

Malware Family	Detection Accuracy		
	Kolbitsch [14]	Park [15]	CuF
Allaple	0.90	0.90	**0.98**
Bagle	0.60	0.80	**0.92**
Mytob	0.72	0.92	**0.91**
Agent	0.10	0.38	**0.91**
Netsky	0.54	0.77	**0.88**
Mydoom	0.90	0.90	**0.96**
Average	0.65	0.87	**0.92**

TABLE 4. False positive rates.

Malware Family	False Positive Rates
Allaple	0
Bagle	0
Mytob	0
Agent	0
Netsky	0
Mydoom	0
Average	0

These results indicate that with the same data set our approach achieves higher detection accuracy than the previous works.

Answer to Q1: CuF can detect new malware variants, and outperforms API call based detection approaches in terms of better effectiveness.

C. ANSWER TO Q2: FALSE POSITIVES

To evaluate false positive rates of CuF, we generate the API call dependency graph for each application in the benign set. And then we evaluate this new graph with each family dependency graph from the malware dataset. If the Eq.(2) is matched, we consider it as a false positive sample. Table 4 shows that the false positives of the begin sample for each malware family are 0. This indicates that our approach can distinguish the begin sample from the malware family well.
Answer to Q2: CuF produces 0 false positives.

D. ANSWER TO Q3: PERFORMANCE OVERHEAD

CuF aims to introduces an acceptable performance overhead. To evaluate the performance impact of our approach, we use several popular commencer application, including 7-zip, IE Browser, and VS Compiler. We perform the evaluation on a computer with Intel Core i7-8550 processor (quad-core, 1.8GHz) and 16GB memory.

First, we conduct evaluation three times for the 7-zip: 1) using a command line option for 7-zip as a simple benchmark; 2) compressing a folder that contains 1 GB of data; 3) archiving three copies of the same folder. Next, we measure the number of pages per second that IE Browser loading a benchmark web site. At last, we measure the time required to compile CuF with VS Compiler.

TABLE 5. Performance overhead.

Test	CuF			Kolbitsch [14]
	Disable	Enable	Relative Overhead	Relative Overhead
7-zip (benchmark)	84sec	85sec	1.2%	2.4%
7-zip (compress)	218sec	224sec	2.8%	4.7%
7-zip (archive)	183sec	195sec	6.6%	8.4%
IE Browser	0.31pages/s	0.32pages/s	3.2%	4.4%
VS Compiler	84sec	96sec	14.3%	39.8%

For each test, we evaluate the overhead of utility with CuF disable and enable respectively, and then we calculate the relative overhead. Also, we compare our performance overhead with Kolbitsch [14]. As shown in Table 5, CuF has a lower overhead compared to the existing work. The worst performance comes from the compilation benchmark (14.3%). However, even in this worst case, our approach still introduce outperforms Kolbitsch (39.8%) [14].

Recall that CuF reduces the number of API call sets which need to monitor, concentrating on the API calls which can implement the malicious behaviors (see Section IV-A). Therefore, the performance overhead of our approach is less than the existing approach.

Answer to Q3: CuF introduces an acceptable performance overhead, which outperforms existing approach in terms of better efficiency.

VI. RELATED WORK

The API call based analysis methods utilize the API call semantics to analyze the malware behavior. These methods can be divided into three classes: control-flow based analysis, data-flow based analysis and graph-based analysis. The details of these classes are provided as follows.

A. CONTROL-FLOW BASED ANALYSIS

Control-flow based analysis mainly uses the API call sequences to represent the semantics of malware [5]–[7]. For example, Sekar et al. [17] leverage the FSA[4] to extract the relations of API calls. Gao et al. [18] combine the white box and black box information to represent the program semantics.

All of existing control-flow based analysis methods utilizes the API call information but the arguments of them is missing, making the evasion attacks possible [19], [20].

B. DATA-FLOW BASED ANALYSIS

Data-flow based analysis uses API call information as well as their arguments to analyze the malware [21]–[23]. For example, Tandon and Chan [22] use the value set allowed for each argument of API calls to improve their system. Bhatkar et al. [23] study the relationship of API call arguments from the data-flow information.

[4]FSA: Finite State Automaton

C. GRAPH-BASED ANALYSIS

Graph-based analysis is another direction of API call based malware analysis scheme. Hu et al. [24] use graphs indexing to implement large-scale malware function call querying in databases. Kolbitsch et al. [14] use dynamic slices analysis to extract data-flow between API calls, and then they leverage model checking of slices to identify unknown malware. Fredrikson et al. [25] present an automated technique for extracting optimally discriminative specifications which uniquely identify a class of program, such as a malware family. Caselden et al. [26] combine data-flow and control-flow graph to generate a API call graph. Gascon et al. [27] extend graph-based malware analysis to the Android environment.

Existing graph-based analysis methods have a common limitation. They only generate graph from a single malware instance, not clustering the graphs from the same malware family. In comparison, our method rely on not a single instance but the clustering of whole malware family, which is a novel approach more robust than existing methods with the new malware variants, resulting in high accuracy of malware detection.

VII. CONCLUSION

In this paper, we proposed CuF, a novel malware clustering approach based on the family graph. Our experiments demonstrate that our approach is both effective and efficient. It can accurately recognize new family variants from known-family with a small overhead. In the future, we plan to study how to apply our idea to cluster malware on other platform such as Android.

REFERENCES

[1] T. Wüchner, M. Ochoa, and A. Pretschner, "Robust and effective malware detection through quantitative data flow graph metrics," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, Jun. 2015, pp. 98–118.

[2] A. Mohaisen, O. Alrawi, and M. Mohaisen, "AMAL: High-fidelity, behavior-based automated malware analysis and classification," *Comput. Secur.*, vol. 52, pp. 251–266, Jul. 2015.

[3] S. Banescu, T. Wüchner, A. Salem, M. Guggenmos, M. Ochoa, and A. Pretschner, "A framework for empirical evaluation of malware detection resilience against behavior obfuscation," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 40–47.

[4] G. Russello, A. B. Jimenez, and H. Naderi, "Poster: Firedroid: Hardening security in Android with system call interposition," in *Proc. IEEE Symp. Secur. Privacy*, 2013.

[5] C. Dietrich, M. Hoffmann, and D. Lohmann, "Cross-kernel control-flow–graph analysis for event-driven real-time systems," *ACM SIGPLAN Notices*, vol. 50, no. 5, Jun. 2015, Art. no. 6.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. USENIX Secur. Symp.*, Aug. 2015, pp. 161–176.
[7] Y. Cao *et al.*, "Edgeminer: Automatically detecting implicit control flow transitions through the Android framework," in *Proc. NDSS*, 2015, pp. 1–15.
[8] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2005, pp. 32–46.
[9] H. Bunke, P. Foggia, C. Guidobaldi, and M. Vento, "Graph clustering using the weighted minimum common supergraph," in *Proc. Int. Workshop Graph-Based Represent. Pattern Recognit.* Berlin, Germany: Springer, 2003, pp. 235–246.
[10] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognit. Lett.*, vol. 19, nos. 3–4, pp. 255–259, Mar. 1998.
[11] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Softw., Pract. Exper.*, vol. 12, no. 1, pp. 23–34, Jan. 1982.
[12] K. Riesen, S. Emmenegger, and H. Bunke, "A novel software toolkit for graph edit distance computation," in *Proc. Int. Workshop Graph-Based Represent. Pattern Recognit.* Berlin, Germany: Springer, 2013, pp. 142–151.
[13] Lastline. (2017). *Lastline Breach Defender*. [Online]. Available: https://www.lastline.com/
[14] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-Y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proc. USENIX Secur. Symp.*, Aug. 2009, vol. 4, no. 1, pp. 351–366.
[15] Y. Park, D. S. Reeves, and M. Stamp, "Deriving common malware behavior through graph clustering," *Comput. Secur.*, vol. 39, pp. 419–430, Nov. 2013.
[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explor. Newslett.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.
[17] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE Symp. Secur. Privacy. (SP)*, May 2001, pp. 144–155.
[18] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, 2004.
[19] E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer, "Taxonomy and survey of collaborative intrusion detection," *ACM Comput. Surv.*, vol. 47, no. 4, p. 55, Jul. 2015.
[20] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 191–206, Mar. 2015.
[21] Sufatrio and R. H. C. Yap, "Improving host-based IDS with argument abstraction to prevent mimicry attacks," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Berlin, Germany: Springer, 2005.
[22] G. Tandon and P. K. Chan, "Learning rules from system call arguments and sequences for anomaly detection," Florida Inst. Technol., Melbourne, FL, USA, Tech. Rep. CS-2003-20, 2003.
[23] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2006, p. 15 and p. 62.
[24] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 611–620.
[25] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 45–60.
[26] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "HI-CFG: Construction by binary analysis and application to attack polymorphism," in *Proc. Eur. Symp. Res. Comput. Secur.* Berlin, Germany: Springer, 2013, pp. 164–181.
[27] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, Nov. 2013, pp. 45–54.

BINLIN CHENG received the Ph.D. degree from Wuhan University, China, in 2016. He is currently a Lecturer with Hubei Normal University. His main research interest includes software security.

QIANG TONG received the master's degree from the Beijing University of Technology, China, in 2001. He is currently an Associate Professor with Hubei Normal University. His main research interest includes machine vision.

JIANHONG WANG received the B.S. degree from Hubei Normal University, China, in 2001, where he is currently a Lecturer. His main research interest includes computer software.

WENHUI TIAN received the master's degree from Central China Normal University, China, in 2010. She is currently a Lecturer with Hubei Normal University. Her main research interest includes computer software.