BinGraph: Discovering Mutant Malware using Hierarchical Semantic Signatures

Jonghoon Kwon, Heejo Lee

Div. of Computer & Communication Engineering
Korea University
Seoul, Republic of Korea
{signalnine, heejo}@korea.ac.kr

***Abstract*— Malware landscape has been dramatically elevated over the last decade. The main reason of the increase is that new malware variants can be produced easily using simple code obfuscation techniques. Once the obfuscation is applied, the malware can change their syntactics while preserving semantics, and bypass anti-virus (AV) scanners. Malware authors, thus, commonly use the code obfuscation techniques to generate metamorphic malware. Nevertheless, signature based AV techniques are limited to detect the metamorphic malware since they are commonly based on the syntactic signature matching. In this paper, we propose *BinGraph*, a new mechanism that accurately discovers metamorphic malware. *BinGraph* leverages the semantics of malware, since the mutant malware is able to manipulate their syntax only. To this end, we first extract API calls from malware and convert to a hierarchical behavior graph that represents with identical 128 nodes based on the semantics. Later, we extract unique subgraphs from the hierarchical behavior graphs as semantic signatures representing common behaviors of a specific malware family. To evaluate *BinGraph*, we analyzed a total of 827 malware samples that consist of 10 malware families with 1,202 benign binaries. Among the malware, 20% samples randomly chosen from each malware family were used for extracting semantic signatures, and rest of them were used for assessing detection accuracy. Finally, only 32 subgraphs were selected as the semantic signatures. *BinGraph* discovered malware variants with 98% of detection accuracy.**

I. Introduction

Malware is a software that implemented to damage computer systems or networks without any user's awareness [1]. Such malware like trojan, virus, worm and bot leads most kinds of cyber crimes, such as DDoS attacks, spam, click fraud and information theft. Even through there are numerical efforts to detect malware, malware threat is rampant.

The major difficulty of malware detection is the rapid increase of malware variants. Symantec and McAfee security reports state that the number of new malware signatures has shown an extreme growth by more than doubling on a year-to-year between 2006 and 2011. Moreover, new signature creation in 2011 is figured about 403 Million [2], [3]. The main reason for the dramatic growth is that the malware variants can be easily produced by adopting new techniques such as code obfuscation and modularization [4]–[7].

The code obfuscation technique supports that malware can change their instruction sequences and also even their signatures with preserving their functionalities. Thus, the code obfuscation provides the chance to evade existing signature-based AV scanners with inexpensive cost [8]. This is the why new malware having explosively increase, and AV venders need to pay the amount of cost for generating new signature. Taha and Jacob *et al.* state that, over 50% of new malware are obfuscated version of existing known malware [9], [10].

Modularization of malware is another important issue of the malware landscape [11]. Riech *et al.* and Thomas *et al.* state that Malware authors often build their codes with several modules [7], [12]. It allows them to easily create new malware. For example, malware authors, who are lack of abilities to implement their own malware, can build new malware by combining existing malware modules. In addition, the modularization also offers possible ways to evade AV scanners as well. If one of the modules is labeled as a malicious by AV vendors, all the malware authors have to do is only simple modification or substitution of the particular module.

Currently, most of AV venders have used signature scanning for malware detection. It is known to be efficient because it guarantees less time, small overhead, and low false positive rate. In spite of the advantages, the signature-based malware detection have a fatal failure. It is not able to detect brand new malware, and even their mutants that their signatures have not been updated yet. Therefore, malware analysis for signature updating is the most important task to AV venders.

When a new suspicious binary is shown up, malware experts conduct behavior analysis through several ways to determine whether or not the binary has malicious attempts. More precisely, they execute the binary on the controlled environment and monitor what the binary do, such as accessing specific registries, creating files into windows system directories, trying to kill the antivirus processes and so on. If any malicious attempts are discovered, a signature of the malware is extracted and updated. However, the signature update is not an easy task since malware analysis is commonly time consuming task, and there are too many malware to be analyzed. In the meantime, unspecified individuals are just exposed to the malware. A new technique to reduce the time for signature update is required.

To this end, we proposes *BinGraph*, a new behavior-based mechanism for analyzing and classifying metamorphic malware by automatic way. *BinGraph* is to mainly reduce the analysis time for the signature update. We utilize a system-call sequence as a binary characteristic. This is based on the assumption that the same API call sequence S is appeared in an original malware M and its new variant malware M' as well. That is, even if the M' is generated by applying code obfuscation to M and the syntax of M' is quite different from M, the semantics of M' cannot be deviated from the semantics of M.

We extract only several instructions related to the system call sequence in a binary, and represent as a form of a directed graph. After that, the graph is separated into several subgraphs depending on their functionalities, and abstracted based on behavioral semantics. We gather the subgraphs from not only several malware variants but also benign executables, and apply the graph mining to determine which subgraphs are only present in malware. Finally, the subgraphs are used for malware detection as the semantic signatures.

We implemented *BinGraph* and evaluated with a total of 827 real-world malware variants that can be classified as 10 malware families, and 1,202 benign executables including Windows system programs. The malware samples were given to us by a AV vendor who collect the binaries from submission to a public web site on May 2012. Among the malware samples, we randomly chose 20% of malware sample from each malware family. 166 binaries were chosen to extract semantic signatures, and rest of them were used for assessing whether the extracted semantic signatures can detect mutant malware binaries accurately.

In the experiments, we collected 8,673 subgraphs from the selected malware samples and 161,328 subgraphs from 1,002 benign executables. By the graph mining step, 1,897 unique subgraphs were extracted as the malicious behavior, and only 32 of them were addressed as the semantic signatures. The sematic signatures are, finally, matched with 661 malware samples as well as 200 benign binaries. *BinGraph* successfully discovered malware variants with 98% of detection accuracy using only the 32 semantic signatures, and there is no false positive.

The main contributions of this paper are threefold:

- We present a novel approach that discover malware even though the malware adopts code obfusctation. Our approach leverages a common semantic characteristic of a malware family, thus it is not influenced whether or not the malware manipulate their instructions.
- We can detect modularized malware. Our approach defines a binary with a hierarchical structure and separates into tiny pieces of code based on functionality. Therefore, we can determine whether or not a binary has a malicious attempt even though the binary contains small piece of malicious code.
- We reduce the number of malware signatures. Our experimental results exhibit that each malware families have distinct semantic signature(s), so we can detect much malware variants using few signatures. This reduction can solve the problem of extremely increasing malware. Furthermore, analysis times and storages to be needed are dramatically reduced as well.

The remainder of this paper is organized as follows. In Section 2, the background of this research and related studies are discussed. And, we describe the architecture of our mechanism, *BinGraph* in Section 3. Section 4 evaluate *BinGraph*, and finally, we conclude this paper and provide an outline of future work in Section 5.

II. Background

Malware and anti-malware warfare is endless. Malware authors adopt new techniques, such as code obfuscation techniques, which can manipulate their codes to avoid detection by defenders, since traditional anti-malware solutions are commonly based on the signature scanning. Accordingly, defenders have studied on the issue of obfuscated malware to improve the detection performance.

Malware detection has been studied with mainly two approaches, static analysis and dynamic analysis, and the approaches have advantages and disadvantages for each. Dynamic analysis [13]–[17] executes malware samples in a controlled, isolated environment such as Virtual Machine (VM), monitors malwares behaviors, and reports automatically. ANUBIS [18], CWSandbox [19] and TTAnalyze [20] are the popular dynamic analysis tools. However, dynamic analysis involves system infection, and malware authors can apply the anti-VM techniques to prevent code analysis. Most importantly, many of the current malware are operated by attacker's commands. Thus, even the malware is loaded on the analysis system, there exist a possibility that the malware does not work. This is un-ignorable limitation.

Binary pattern matching [21], data flow [22], and code flow analysis [23], [24] are the examples of static analysis. They analyze the malware without malwares execution, thus they commonly guarantee fast and safe analysis. In addition, static analysis can cover the entire malware code. Of course, static analysis also has limitations. Static analysis has difficulty to analyze code obfuscated malware. Using the code obfuscation technique, the malware can change their codes without changing functionality.

To overcome the limitations [24], code normalization [25], [26] and semantic approaches have been proposed [27], [28]. Even though the approaches are helpful, code normalizations are depending on specific obfuscating techniques, and the semantic approaches have shown a weakness to new type of code obfuscation.

Recent studies have shown that API calls can be useful for malware detection since the API calls reflect the functionalities of a program. Bai *et al.* presented a malware detection research based on critical API-calling graph(CAG) [29]. The approach

deals with matching of the CAG rather than considering all API calls. Eskandari and Hashemi proposed a metamorphic malware detection using API calls on the Control Flow Graph(CFG) [30]. They converted the resulted sparse graph to a vector and adopted few machine learning techniques. However, the works are only focused on the syntactic structure of API call sequences, not the semantics of the API calls. Therefore, their approaches cannot perform well for detecting malware variants created by replacing the APIs with other APIs or reordering the APIs while preserving the same functionalities of the program.

CodeGraph [31], our previous work, has been proposed to detect metamorphic malware using semantic signatures. As one of the static analysis, CodeGraph converts the API call sequence of the malware into a graph to extract the semantic, and converts the graph to a code graph used for the semantic signature. Finally, the semantic signature is compared with other semantic signatures to calculate similarity. Unfortunately, CodeGraph is susceptible to new types of malware, especially modularized ones, since it only concerns whole behaviors of malware as one semantic signature.

To unveil the modularized malware, the semantic signature has to be represented as the functionality level rather than the code level, and also contain unique behavioral features. To address this, our work is focused on;

1) How to build subgraphs which can be distinguished based on their functionalities,
2) How to address their semantics in the graph structure, and
3) How to extract semantic signatures which represent specific malware or malware families.

This work is basically an extension of the previous work, where the concept of building the semantic signature was proposed for detecting metamorphic malware.

III. BinGraph System

BinGraph is a system to detect metamorphic malware using semantic signatures. Fig. 1 shows the working flows of *BinGraph*, and it has two phases: semantic signature extraction and malware detection. At the first phase, we construct the semantic signatures using both malware samples and benign executables. Then, malware detection against unknown binaries is performed by matching with the semantic signatures at next phase. Our system consists of five different analysis steps, behavior graph construction, subgraph extraction, graph abstraction, graph mining, and signature matching. In this section, we describe the details of *BinGraph* from the perspective of the different analysis steps.

A. Behavior Graph Construction

Current malware detection is commonly based on the syntactic signature matching, and malware authors adopt new techniques such as code obfuscation to evade the malware

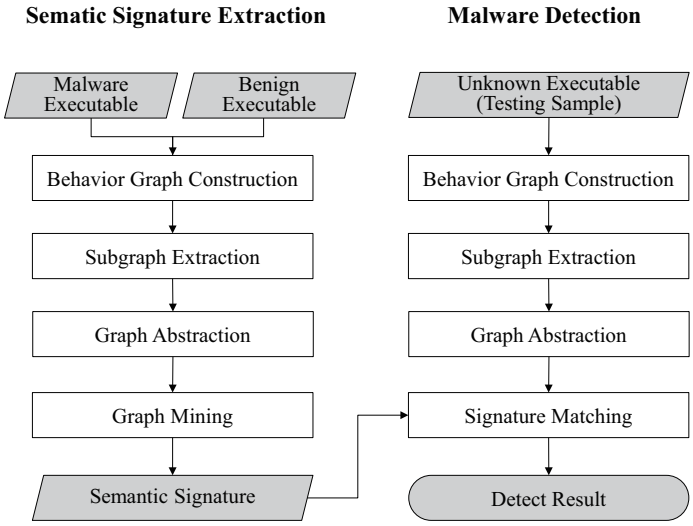


Fig. 1. *BinGraph* architecture with two phase, semantic signature extraction and malware detection.

detection system. It is surely effective since the technique changes syntactics of malware without semantics. This is the motivation that *BinGraph* is focused on semantic characteristics of malware rather than syntatic characteristics.

The system call features are critical importance for understanding and identifying the semantic of malware. In the Windows system, applications access the system resources, such as process, memory, registry, and network, through Win32 API. Malware is also able to access the system resources by using API functions to implement their tasks, and semantics of the API functions are clear. Hence, *BinGraph* utilizes the system call information, especially call sequence, as the semantic characteristics of binaries.

To represent the semantic of a binary, *BinGraph* construct a behavior graph using the API call sequence. Therefore, the extracting accurate call sequence is required. The code analyzer [31], [32], fortunately, provides the extraction from a executable. The code analyzer first extracts instructions related to the system call sequence in the binary, and represents the result into the set of nodes and edges. The result of code analyzer is a call graph, $G = (V, E)$, where V is a set of system calls selectively chosen among the system calls. And E is a set of calling relations of the system calls in V, e.g., $E = \{(v_i, v_j)\}|v_i, v_j \in V\}$, where v_i denotes the caller, and v_j denotes the callee.

We build the set of node V, which contains the system calls, through the *Import Address Table (IAT)* in a executable. Next, we generate the set of edges E for the call graph G based on the system call sequence, where v_i is the caller and v_j is the callee.

B. Subgraph Extraction

Many malware are implemented as several modules, and the modules are shared to create other malware or attached

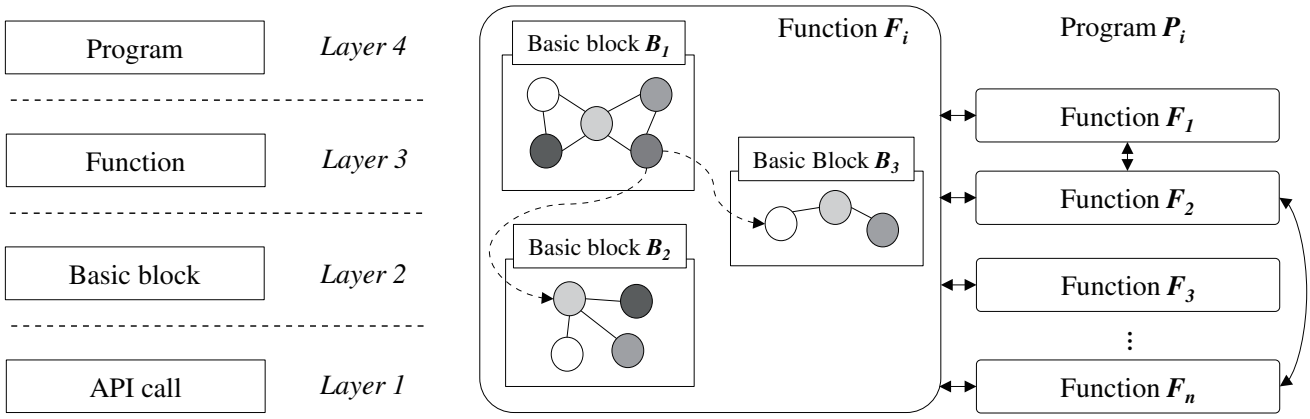


Fig. 2. A hierarchical structure of binaries with four layers.

into benign executables. To detect such kinds of malware, we need to understand the malware in terms of functionalities. Moreover, we need to understand the hierarchical architecture of malware and duplicated behaviors across the malware variants.

To do this, we classify the binary into four layers, which is in order of API calls, basic blocks, functions, and a program. Fig. 2 represents the hierarchy. A program P consists of functions F, a function F consists of basic blocks B, and a basic block B consists of API calls v. The API calls represent the basic nodes composing a behavior graph, however a single API call is not sufficient to express a specific behavior.

The basic block [33] is a portion of the code with certain desirable properties. Usually, the basic block has an entry point and an exit point, meaning there is no jump instruction and only the last instruction can cause other basic block to execute. We consider the basic block as a basic subgraph G_B, since it can be the smallest piece of code embracing a behavior with a purpose.

In few cases, the basic blocks cannot represent specific behaviors. For example, the size of all of them in a binary are too small or only small number of API call is arisen in the basic blocks. To handle this, if all the basic blocks are not sufficient to exhibit the semantic characteristics of a binary and we are not able to generate unique subgraphs for the binary, we move to the upper layer, *i.e.*, function layer, to produce subgraph G_F that expresses the function. More precisely, if a function F_i consists of basic blocks B_1, B_2 and B_3, and all the basic blocks cannot present the semantic characteristics of the binary, G_{Fi} is considered as one of the subgraphs.

G_P is a super graph that represents all behaviors in program P, and it is widely used in previous researches which applied API call based binary analysis [34]. However, in *BinGraph*, the G_P is considered as a semantic behavior graph if and only if there is no sufficient subgraphs for representing unique semantics. This hierarchical subgraph approach guarantees at least same detection performance with the previous works in worst case, such as there is no unique malicious subgraph in malware.

C. Graph Abstraction

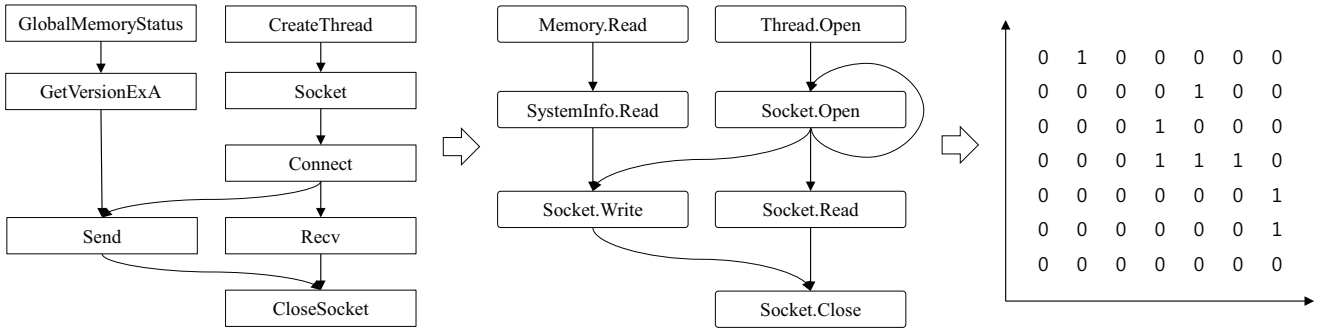
So far, we have constructed call graphs based on the API call to detect malware variants through comparing the graphs. Unfortunately, the call graphs have thousands of nodes and each nodes represents each calls, and the graph isomorphism is as commonly known as a NP-complete problem. We need to simplify the call graphs to comparable forms with preserving semantic characteristics. We call this step as the graph abstraction.

To this end, we substitute the system calls into 128 nodes based on their semantics. More precisely, we classify each calls by 32 objects of a system call, where the objects are process, register, memory, socket, and so on. And then, the calls are classified again by four behaviors of the related object, where the behaviors are open, close, read and write. For instance, *CreateProcess()* is a member of the process-open group. Note that the 32 objects are referenced from the *Microsoft Developer Network (MSDN)* in order to implement *BinGraph* on Windows system.

After the classification, the nodes within the same groups are merged and edges are relocated based on the call relationships of the nodes.

After subgraph extraction from a call graph, every subgraphs are transformed abstracted graphs. We use an adjacent matrix as a data structure to store the graph information. An adjacent matrix is the most proper data structure for graph, since the graph is a directed graph and the number of nodes is only 128. Finally, we can successfully abstract and transform the call graphs to 128×128 matrices.

Fig. 3 is a simple example of the graph abstraction. A subgraph (a) presents malicious behavior that collects system information and sends to third party through the network. Note that malware usually collects the information from infected machines, and this behavior graph is captured from one of the malware samples. The malware invokes the system calls to collect Windows version and current memory status, and sends them to attackers through a network connection. The subgraph is abstracted as forms of a semantic graph (b) with



(a) An example of API call graph that presents system information leakage.

(b) A semantic graph that is abstracted and transformed from the information leakage graph.

(c) A matrix data structure that contains the semantic graph in fixed size.

Fig. 3. Semantic abstraction with a real example of malicious behavior.

128 nodes, finally, transformed into the matrix data structure (c).

This graph abstraction offers next three advantages:

1) The abstracted graphs can express semantics of the executables,
2) The abstracted graphs can be represented as a small data structure irrespective of their sizes or hierarchy, and
3) Computation time is extremely decreased, since the size of a graph has only 128×128, thus it can be completed within finite time at any condition.

D. Graph Mining

In this step, we conduct the frequent subgraph mining for extracting semantic signatures. A frequent subgraph means that it appears simultaneously in a fraction k of all binaries. If k is greater than two and it appeared in only malware samples, it is regarded as a shared module across the malware variants. Consequently, it can be useful as a semantic signature to detect malware variants.

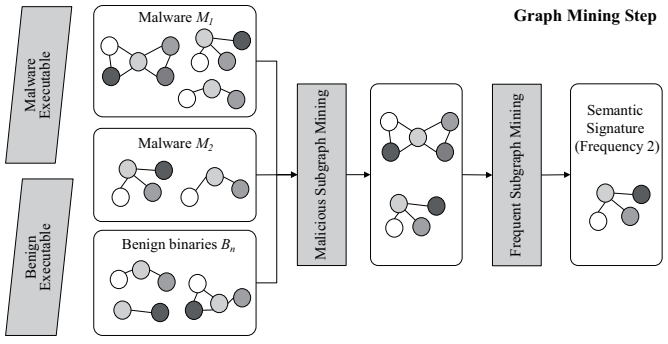


Fig. 4. A overview of graph mining process.

To mine graphs only observed in the malware, we perform the comparison between two graph sets came from malware and benign binaries respectively. To this end, the three analysis steps, behavior graph construction, subgraph extraction and graph abstraction, are equally performed to malware and benign binaries. Next, we compare the each graph to determine whether or not it appears in benign binaries. If a graph appears in a benign graph set, it is removed immediately. But, if a graph only appears in a malware graph set, it is added to the set of a candidate of semantic signatures.

Even though a graph is not appeared in the benign graphs, we do not consider the graph as a semantic signature directly. In some malware cases, there are tens of or hundreds of such graphs. If the number of samples to be analyzed is small, that is no big deal. However, analyzing millions of samples is a big deal since the time to process increases exponentially in accordance with the number of signatures, and millions of binaries are submitted to AV vendors lately. Therefore, we need to minimize the number of signatures with maximizing the coverage over malware.

The greedy strategy [35] is selected for efficient signature mining. At first, we select a graph that has the highest frequency, and exclude malware samples that can be detected by the graph from training data pool. After that, we select the most frequent graph among the data pool again. The signature mining with the greedy strategy is performed until the data pool becomes empty. The number of signature is rarely similar with the number of malware sample, and most cases, very small number of signatures is selected.

E. Semantic Signature Matching

The previous analysis adopts graph mining with sets of known malicious and benign graphs to extract semantic signatures. The semantic signature represents a behavior that only appears in particular malware or variants. This step is for detecting unknown malware by matching the semantic signature.

In order to discover unknown malware, we first extract semantic graphs to be compared with the semantic signatures. As we can see in Fig. 1, *BinGraph* applies the behavior

TABLE I

STATISTICS OF SEMANTIC SIGNATURES FOR EACH MALWARE FAMILY

Malware Name	# of Sample	# of Total Subgraph	# of Unique Subgraph	# of Malicious Subgraph	# of Semantic Signature
Conficker	8	845	422	13	3
Killav	20	69	24	5	1
Koobface	8	173	126	41	3
Nebuler	20	3246	605	413	1
OnlineGameHack18	20	271	43	21	1
Palavo3	20	2011	1632	1345	7
Qhost	20	400	20	8	1
Rustock	10	270	193	0	4[a]
TDSS3	20	69	24	10	10
Userinit	20	1319	144	41	1

[a]The semantic signatures of Rustock were composed in function layer.

graph construction, the subgraph extraction, and the graph abstraction to unknown binary. After the pre-analysis steps, matching with the semantic signature is performed.

As we mentioned in section 3.C, the graph isomorphism problem is NP-complete, thus we utilize the matrix data structure for semantic graph matching. Every semantic graphs and signatures are represented by 128×128 matrices. More in details, each matrix is stored as a 2KB bit-stream, and simple XOR operation is executed to determine whether or not the matrices of semantic signatures are matched with the matrices of semantic graphs. Whenever a match is found, the result of XOR operation is equal to zero, we define the corresponding graph as a malicious behavior, and we classify the corresponding binary as a malware.

IV. EVALUATION

We performed several experiments to evaluate accuracy and efficiency of *BinGraph*. This section describes the evaluation results and interesting features in detail.

A. Environments

BinGraph has been implemented and evaluated with a total of 827 real-world malware variants. The samples were provided from an AV company. The company obtained the samples through the public submission Web site and classified the samples upon their own analysis processes. The malware samples were provided with names of 10 malware families, and they all had distinct binary patterns.

To ensure that whether the classification of the malware families is trustworthy, we leveraged the samples with two different AV engines (*e.g.*, Kaspersky and BitDefender). Unfortunately, the AV engines showed a difficulty to detect all the malware samples. Besides, the labels of the malware conflict depending on the AV engines. One of the binaries labeled as *Generic.Malware.SP!g.E654213E*, thus we could not determine its malware family. The signature database might not be up to date at that time. Consequently, our evaluation was performed based on the initial classification. The families were *Conficker* (40), *Killav* (100), *Koobface* (37), *Nebuler* (100), *OnlineGameHack18* (100), *Palavo3* (100), *Qhost* (100), *Rustock* (50), *TDSS3* (100) and *Userinit* (100).

Total 1,202 benign binaries, Windows system program binaries on pure environment and freeware binaries, were utilized for our evaluation. We verified the benign samples with the three AV engines (Kaspersky, BitDefender and the company's AV engine) and confirmed that all the samples were not classified as malicious. The size of the samples ranges from few KB to tens of MB, and a total of 161,328 unique subgraphs were produced. The subgraphs cover the various benign behaviors, and we utilized them to eliminate benign behaviors from semantic signatures.

B. Semantic Signature Generation

Lately, detection rates of less than 20% are commonly seen on newly discovered malware [36]. It means AV vendors have only 20% of samples for new malware. Hence, we need to discover all the malware variants using information obtained from the 20% of samples. Among the provided 827 malware samples, 20% of each family (166 samples) were randomly selected for generating the semantic signatures.

Table I explains the analysis results for each malware family. A total of 8,673 subgraphs were extracted and only 3,233 subgraphs show uniqueness. Finally, we analyzed 1,897 subgraphs as malicious by comparing with benign graphs. However, all of the 1,897 malicious graphs could not be the semantic signature, since comparing thousands of signatures is not efficient as we mentioned in Section 3.D. Therefore, we needed to figure out most efficient semantic signatures.

The efficiency in malware detection means how many malware can be caught by the signature. In other words, how

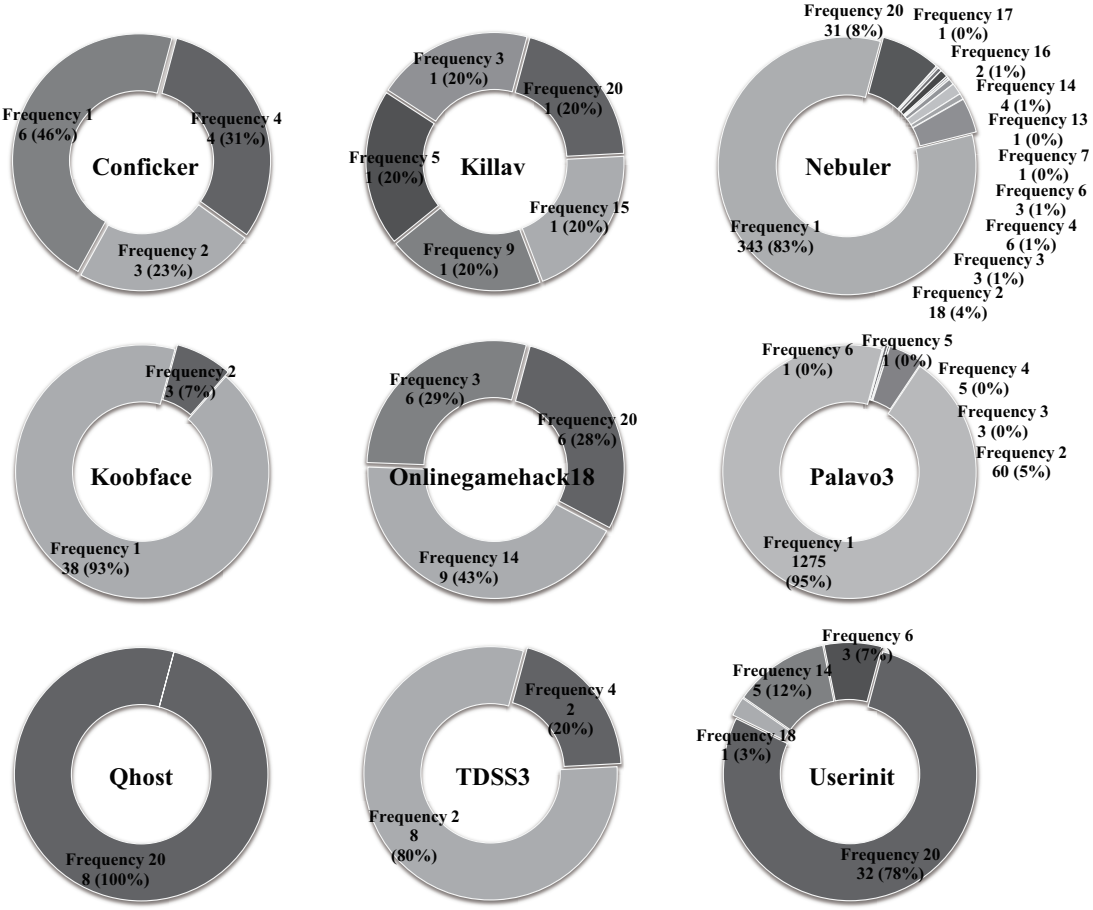


Fig. 5. Distributions of subgraph's frequency for various malware families.

often the signature is appeared across the malware variants. We define this as "Frequency". If the frequency is equal to k, the malicious graph appears in k distinct malware variants. Fig. 5 illustrates the frequency of the malicious graphs. For instance, 5 malicious graphs are discovered in 20 *Killav* variants and each of the malicious graphs had frequency 20, 15, 9, 5 and 3 respectively. That is, one of the malicious graph appears in every *Killav* variants, thus it bocomes the most efficient semantic signature.

Half of the malware samples in Fig. 5, *Killav*, *Nebuler*, *Onlinegamehack18*, *Qhost* and *Userinit*, show at least one or more subgraphs that rank maximum frequency 20. Other malware families also have frequent subgraphs ranked more than two. Even though the frequent subgraphs are not able to cover all variants in a malware family, they are undoubtedly useful since a frequent subgraph can discover more than two malware variants.

We performed the semantic signature selection by applying a greedy algorithm. As a result, only 32 semantic signatures were constructed for 166 malware variants. The number of semantic signatures decreased to only 19.28% of the number of syntactic signatures.

C. Detection Accuracy

We assessed whether the semantic signatures can accurately detect malware variants without false alarm. To this end, we performed two experiments. First, we evaluated the semantic signatures with the remain 80% of real malware samples. And then, we applied the semantic signatures to benign binaries to evaluate false positives.

To validate the detection accuracy, we performed the four analysis steps (viz., Fig. 1) against the 661 malware variant (80% of samples). First, a call graph was extracted from each malware, and second, subgraphs were extracted based on the basic blocks. Third, the subgraphs were transformed into 128×128 matrices through the graph abstraction step. Finally, the graphs were matched with semantic signatures to determine whether or not the graphs attempt malicious behavior.

Table II exhibits the detection results. Among the 661 malware samples, 649 were matched to the 32 semantic signatures. Average detection rate was 98.18%. Furthermore, we also performed matching with 200 benign binaries. 6,379 subgraphs were extracted from the benign samples and there was no matched results with our semantic signatures. As a

TABLE II

DETECTION RESULTS FOR REAL MALWARE SAMPLE

Malware Name	# of Testing Sample	# of Matched Sample	Detection Rate
Conficker	32	32	100%
Killav	80	80	100%
Koobface	29	25	86.20%
Nebuler	80	80	100%
OnlineGameHack18	80	80	100%
Palavo3	80	73	91.25%
Qhost	80	80	100%
Rustock	40	39	97.50%
TDSS3	80	80	100%
userinit	80	80	100%

result, *BinGraph* effectively discovered metamorphic malware without false positives.

V. CONCLUSION

In this paper, we present *BinGraph*, a novel approach to analyze and classify metamorphic malware. The subgraph analysis can provide not only metamorphic malware classification, but also behavior analysis of modules used by malware variants. Such information can be useful to AV venders and make the malware authors harder to develop metamorphic malware. For the future work, we will extract semantic signatures presenting common behavior across various kinds of malware families, and analyze specific behavioral features of metamorphic malware.

VI. ACKNOWLEDGMENTS

This research was supported by the KCC(Korea Communications Commission), Korea, under the R&D program supervised by the KCA(Korea Communications Agency)(KCA-2012-12-911-01-111). Additionally, this work was partially supported by Seoul City R&BD program WR080951.

REFERENCES

[1] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007.

[2] Symantec, "Global internet security threat report," Apr. 2012. http://www.symantec.com/threatreport/.

[3] McAfee, "Threats report: First quarter 2012," 2012. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2012.pdf.

[4] M. Driller, "Metamorphism in practice." 29A Magazine, 2002.

[5] L. Julus, "Metamorphism." 29A Magazine, 2000.

[6] Rajaat, "Polymorphism." 29A Magazine, 1999.

[7] K. Rieck, T. Holz, C. Willems, P. Dussel, and P. Laskov, "Learning and classification of malware behavior," in *Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 08)*, pp. 108–125, June 2008.

[8] D. Mohanty, "Anti-virus evasion techniques and countermeasures." Whitepaper, Aug. 2005. http://www.hackingspirits.com/ethhac/papers/whitrepapers.asp.

[9] G. Taha, "Counterattacking the packers." McAfee, 2007.

[10] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in Computer Virology*, 2008.

[11] P. Ferguson, "Observations on emerging threats," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2012.

[12] K. Thomas and D. Nicol, "The koobface botnet and the rise of social malware," in *IEEE Int. Conf. Malicious and Unwanted Software (Malware 10)*, pp. 63–70, Oct. 2010.

[13] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior based spyware detection," in *15th Usenix Security Symposium*, 2006.

[14] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *Usenix Annual Technical Conference*, 2007.

[15] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell, "A layered architecture for detecting malicious behaviors," in *Symposiumon Recent Advances in Intrusion Detection (RAID)*, 2008.

[16] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th USENIX Security Symposium*, Aug. 2009.

[17] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[18] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[19] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," in *IEEE Security & Privacy*, pp. 32–39, 2007.

[20] U. Bayer and E. Kirda., "Ttanalyze: A tool for analyzing malware," in *15th Ann. Conf. of European Inst. for Computer Antivirus Research (EICAR)*, pp. 180–192, 2010.

[21] F. Cohen, "Computer viruses: Theory and experiments," *Computers & Security*, pp. 22–35, Feb. 1987.

[22] D. Chess and S. White., "An undetectable computer virus," in *Virus Bulletin Conference*, Sept. 2000.

[23] A. Moser, C. Krugel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *IEEE Security & Privacy*, pp. 231–245, 2007.

[24] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *23th Annual Computer Security Applications Conferecne (ACSAC 2007)*, pp. 421–430, Dec. 2007.

[25] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," in *IEEE Security & Privacy*, pp. 46–54, 2007.

[26] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *IEEE Security & Privacy*, pp. 32–46, 2005.

[27] M. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Aug. 2008.

[28] V. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature generation and detection of malware families," in *the 13th Australasian Conference on Information Security and Privacy (ACISP 2008)*, pp. 336–349, July 2008.

[29] L. Bai, J. Pang, Y. Zhang, W. Fu, and J. Zhu, "Detecting malicious behavior using critical api calling graph matching," in *1st International Conference on Information Science and Engineering*, pp. 1716–1719, Dec. 2009.

[30] M. Eskandari and S. Hashemi, "Metamorphic malware detection using control flow graph mining," *International Journal of Computer Science and Network Security*, Dec. 2011.

[31] J. Lee, G. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in *ACM Int'l Symp. on Applied Computing(ACM SAC 2010)*, Mar. 2010.

[32] K. Jeong and H. Lee, "Code graph for malware detection," in *Int'l Conf. on Information Networking (ICOIN)*, Jan. 2008.

[33] WIKIPEDIA, "Basic block," 2012. http://en.wikipedia.org/wiki/Basic_block.

[34] K. Han, I. Kim, and E. Im, "Detection methods for malware variant using api call related graphs," in *In proceedings of the International Conference on IT Convergence and Security*, Dec. 2011.

[35] P. Black, "Greedy algorithm," in *Dictionary of Algorithms and Data Structures*, Feb. 2005.

[36] A. Fried, "Whose internet is it, anyway," in *Blackhat Conference*, Feb. 2010.