Deriving Common Malware Behavior through Graph Clustering *

Younghee Park, Douglas Reeves
Cyber Defense Laboratory
Department of Computer Science Department
N.C. State University, Raleigh, NC, USA
{ypark3, reeves}@ncsu.edu

ABSTRACT

Detection of malicious software (malware) continues to be a problem as hackers devise new ways to evade available methods. The proliferation of malware and malware variants requires methods that are both powerful, and fast to execute. This paper proposes a method to derive the common execution behavior of a family of malware instances. For each instance, a graph is constructed that represents kernel objects and their attributes, based on system call traces. The method combines these graphs to develop a supergraph for the family. This supergraph contains a subgraph, called the *HotPath*, which is observed during the execution of all the malware instances. The proposed method is scalable, identifies previously-unseen malware instances, shows high malware detection rates, and false positive rates close to 0%.

1. INTRODUCTION

Malware (malicious software) continues to be widespread despite the common use of anti-virus software, and the diversity of malware is increasing. Some anti-virus vendors reported approximately half a million instances of malware for the first half of 2007, and several thousand new malware variants per day [10, 24]. The formidable increase in the number of malware samples has been a turning point in malware research. To reduce the resource and management costs identifying such large numbers of malware instances, deriving representative malware behavior has become an important challenge [23, 12].

Previous work for behavior-based malware detection is based on static analysis (i.e. disassembly), or on dynamic analysis at runtime. Static analysis relies on disassembly to identify malicious behavior, and makes use of control flow analysis and data flow analysis [13, 29, 19]. However, these methods need full disassembly for analysis, and may have problems with code polymorphism (e.g. encryption, packing of binaries) and metamorphism (e.g. code obfuscation). Moreover, a single signature or syntactic structure is not applicable to new threats and new instances.

*This work has been funded by the National Science Foundation under grant CNS-0831081. The contents do not represent the official position of the U.S. government.

ASIACCS '11, March 22–24, 2011, Hong Kong, China.

To overcome the drawbacks of static analysis, dynamic behavior-based analysis has gained much interest [2, 1, 28, 16, 26, 12] since the runtime behavior of malware cannot be changed dramatically. Many of these works use advanced techniques such as symbolic execution and taint analysis, causing major performance overhead, as discussed in [28, 15, 17, 27]. Kolbitsch et al. [17] proposed a malware detection method to efficiently extract a system call behavioral graph for each instance, which includes dependencies between system calls and their arguments, specifically, with the usage of buffers and strings. However, extracting common behavior that detects any variants of malware, with a low false positive rate, still remains a demanding problem.

In this paper, we propose a new malware detection method. This method makes use of one representative common behavioral graph for all malware instances in a type of malware family, instead of one behavioral graph per instance. The common behavioral graph is created from individual behavioral graphs through graph clustering. Each behavioral graph represents the usage of kernel objects, the object attributes, and the dependencies between these kernel objects. Such properties are obtained through monitoring system calls and their arguments at runtime. The essential observation is that each malware instance in the same family shows similar runtime behaviors, and that behavior is distinct from those of benign applications. The resulting common graph represents common behavior in malware instances of a family. The common graph has a unique subgraph, called the HotPath, which is shared by all the malware instances in the family. By using the common behavioral graph and the HotPath, the proposed method detects new variants of the family with high accuracy.

The proposed method is different from others in the following ways. There is one common graph per malware family. This graph is based on usage of kernel objects, rather than on specific system calls. The graph contains a common path, called the *HotPath*, which is execution behavior shared by all malware instances in the family. Obfuscation of malware instruction execution does not alter the common behavior of the family.

A prototype of the proposed method has been implemented. It uses the Xen hypervisor to intercept system calls with their arguments on the Windows guest operating system. When evaluated on well-known data sets, higher detection rates are achieved than by previous research, with false positive rates close to or at 0%. The overhead of graph construction and malware identification is on the order of tens of milliseconds.

Our contributions are as follows. First, this method proposes a method to extract a kernel object behavioral graph (KOBG) for a malware instance, from system call traces collected during execution. Second, from a set of KOBGs, the method constructs a common behavioral graph that captures representative execution

behavior of a malware family, by using graph clustering. The unique subgraph (*HotPath*) shared by all of the binaries in the family is derived automatically. Third, the generated common behavioral graph is scalable since the common graph is not changed significantly according to new malware variants added into the family. Finally, the proposed method of malware detection by matching with the common behavioral graph is robust against system call attacks.

The paper is organized as follows. Section 2 presents an overview, and details about the proposed method. Section 3 shows the results of evaluating the method with actual malware instances. Section 4 compares the method to previous work, and section 5 concludes the paper.

2. THE PROPOSED METHOD

The new method has two steps for malware detection. In the first step, a set of malware instances (executables) are processed to derive kernel object behavioral graphs (one per instance) that represent their behavior. In the second step, the graphs for malware instances that are in the same family are then clustered into a single graph (a Weighted Common Behavioral Graph) that represents the behavior of all members of that family. Finally, the clustered single graph detects new instances of malware with very low overhead by comparison to the weighted common behavioral graphs for different malware families.

2.1 Kernel Object Behavioral Graph Generation

The graph that represents the behavior of a malware instance is a *kernel object behavioral graph*. A kernel object is a memory block in the kernel. This memory block is a data structure whose members maintain information about the object. In the Windows operating system, several kernel objects are defined, including processes, threads, files, events, sockets, etc. [20]. Since the kernel object data structures are accessible only by the kernel, it is not possible for an application to locate these data structures in memory and directly alter their contents.

A kernel object behavioral graph is constructed from information that is collected from a running program in a virtualized environment. Based on the intercepted system call traces with the arguments, the method identifies relationships between kernel objects. A kernel object behavioral graph (KOBG) is a weighted directed graph described by $g = (V, E, \lambda, \mu)$. V is a set of vertexes (vertices, v), and each represents a type of kernel objects. The attributes of a vertex indicate the specific name of the kernel object. $E \subseteq V \times V$ is a set of edges e, and each edge indicates a dependency between two kernel objects. The dependency is presented by handle types and handle values. A dependency consists of a handle type and a handle value for the object. The object handle indicates an identifier to represent a system resource inside a kernel, which is allocated by an application. $\lambda : V \rightarrow N^+$ is a function assigning positive weight to the vertexes (initially, $\lambda(v) = 1\ \forall v \in V$). $\mu : E \rightarrow N^+$ is a function assigning positive weights to the edges (initially, $\mu(e) = 1\ \forall e \in E$).

When kernel objects are accessed via system calls, most such calls include the handle and the object attributes of kernel objects as arguments of the call. The object name is identified by the *ObjectAtrributes* parameter in the intercepted system call. Each kernel object has a creator function and a destructor function. For instance, *NtCreateFile* creates a new file or directory, or opens an existing file, directory, device, or volume, while *NtClose* terminates the kernel object with the handle created by *NtCreateFile*. In addition, *NtCreateProcess* makes a new process and its primary thread. The object name is extracted by reading *ObjectAttributes* parameter values. The kernel object is no longer accessible when *NtClose* or *NtTerminateProcess* is called with this handle.

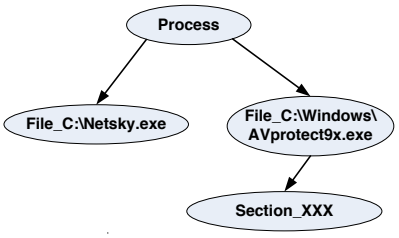
Once a kernel object is created by a system call, its handle is returned and used in other system calls as an argument. This means that the kernel object is accessed by other kernel objects associated with those system calls. For example, when *NtCreateThread* is called, it takes a process handle as input, and returns a thread handle. The dependency between process and thread objects is represented by an edge between the corresponding vertexes in the KOBG. The weight of all edges in the KOBG is 1. Figure 2 shows an example of a kernel object behavioral graph while comparing to a typical system call behavioral graph. Figure 1 shows an excerpt of system call traces of one instance of the Netsky worm, similar to [17]. As shown in Figure 2, this part of the system call traces generates two different graphs, a typical system call behavioral graph and kernel object behavioral graph. In the first trace of Figure 1, it opens the file (Netsky.exe). Next, a new file (AVProtect9.exe) is created to copy its code. Finally, the worm reads its own program code to copy itself into the new file.

```
1. NtCreateFile ( Out FileHandle A, ….. , ObjectAttributes -> C:\Netsky.exe)
2. ……..
3. NtCreateFile ( Out FileHandle B, …. , ObjectAttributes ->
C:\WINDOWS\AVprotect9x.exe)
4. ……..
5. NtCreatSection ( In FileHandle A, … Out SectionHandle C, ObjectAtrribtue->XXX)
NtMapViewOfSection ( In SectionHandle C, ….)
6. ……..
7. NtWriteFile ( In FileHandle B, ….)
8. ……..
```

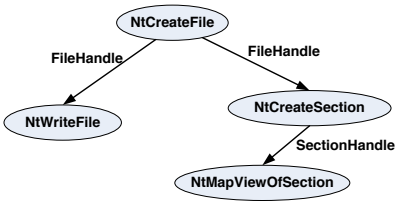
Figure 1: A Part of System Call Traces for Netsky.

First, for KOBG, each vertex indicates each kernel object and the object name in Figure 2(a). Each edge indicates a dependency between two kernel objects. The dependence is derived when one kernel object affects the use or creation of another kernel object. In detail, it is assumed that a kernel object, *Process*, implicitly appears since all the traces are generated by its own process. From the traces, since *NtCreateFile* function opens a file in the *C:* directory, one vertex for this file (*File* object) is created and the object name is *C:\Netsky.exe*. The vertex from the second trace, *File_C:\Windows\AVprotect9.exe*, is generated by the same process. The edge between these two kernel objects is implicitly connected to the *Process* object because a process opens and creates the file. In addition, *NtCreateSection* assigns a memory block. Another vertex for this section is created and the object name is a memory address in this trace. However, in other traces, the object name of the section can be DLL names as well. As in this figure, an edge is created between the *Section* object and the *File* object because this file makes this section. For comparison, Figure 2(b) represents an example of a typical system call behavioral graph, described in [17]. Note that the figure of a system call behavioral graph does not reflect on dataflow analysis.

In summary, a kernel object behavioral graph represents the usages of kernel objects labeling with their object names. Even if any kernel objects are generated by attackers, the relationship between two specific kernel objects is hard to be broken as shown in Section 3. In other words, since multiple kernel handle values can indicate the same kernel object at runtime, we use the kernel object name rather than original handle values. A handle indicates an identifier of a kernel object. However, the typical system call behavioral graph illustrates the usages of system calls based on the dependency of handle values. The dependency based on handles may be easily broken by attackers. Furthermore, rather than using



(a) An example of kernel object behavioral graph (Our approach)



(b) An example of system call behavioral graph

Figure 2: An illustration of kernel object behavioral graph from the exemplar code in Figure 1. Note that Figure 2(b) shows a typical system call behavioral graph based on the dependency of system calls.

original individual graph, the proposed work will use one common graph from a set of many graphs, as discussed in the next section.

2.2 Weighted Common Behavioral Graph Generation based on an Approximate Algorithm

For the execution of one malware instance, a KOBG is derived in the way just described. For malware identification purposes, the KOBGs of a family of malware instances are clustered into a single behavioral graph. Let $\mathbf{G} = \{g_1, ..., g_n\}$ be a set of n kernel object behavioral graphs constructed from the system call traces collected during the execution of n binaries, where these binaries are classified as belonging to the same malware family.

To cluster behavioral graphs, we make use of existing methods of clustering graphs, as in Definition 1. The *Weighted Minimum Common Supergraph* (WMinCS) is a clustering method formally described in [3]. Based on some well-known concepts in graph theory [9], including graph isomorphism, the maximum common subgraph(mcs), and the weighted minimum common subgraph ($wmcs$). The *Minimum Common Supergraph* ($MinCS$) of G and H is defined as Eq.(1). Note that $G - mcs(G, H)$ and $H - mcs(G, H)$ are the graphs obtained by removing the mcs of G and H from G and H, respectively.

$$MCS(G, H) = mcs(G, H) \cup (G - mcs(G, H)) \cup (H - mcs(G, H)) \quad (1)$$

The Minimum Common Supergraph for a set of graphs is similarly the union of mcs of those graphs and the edges and vertexes in those graphs that are not included in the mcs.

DEFINITION 1. *Let $\mathbf{G} = \{g_1, g_2, ..., g_n\}$ be a set of weighted directed graphs; a* Weighted Common Supergraph *of $\mathbf{G}$, is a weighted directed graph $g = (V, E, \alpha, \beta, \lambda, \omega)$ such there exist subgraph isomorphisms from g_i to g $\forall\, i \in \{1, ..., n\}$. We call g a* Weighted Minimum Common Supergraph *of the set of graphs $\mathbf{G}$, denoted $WMinCS(\mathbf{G})$, if there exist no other weighted common supergraph of $\mathbf{G}$ that has fewer nodes than g. Let $f_i : V_i \rightarrow V$ be an isomorphism between g_i and a subgraph of g. Let (u, v) be a pair of vertexes of V. The weight (w) of each edge $e = (u, v)$ is k/n if there exist exactly k distinct isomorphisms $f_j(e_{il}) = (f_j(u_i), f_j(v_l))$ such that $f_j(e_{il}) = e$. The weight of the vertex is simply set to 1.*

The computation of a WMinCS is a NP-complete problem. Moreover, the complexity of computing WMinCS of a set of graphs is exponential in the number of graphs. Consequently, the proposed method uses an algorithm that approximates the WMinCS of a set of graphs $\mathbf{G}$.

The approximate WMinCS algorithm is derived from the weighted maximum common subgraph ($wmcs$) of a pair of weighted graphs. The computation of $wmcs(g_1, g_2)$ is based on the computation of the mcs of two graphs. To generate $wmcs$ between two weighted behavioral graphs, the McGregor algorithm [8] is used among various known efficient algorithms in the literature[8]. The WMinCS between two weighted behavioral graphs is generated by Eq.(2) as follows.

$$WMinCS(g_i, g_j) = wmcs(g_i, g_j) \cup (g_i - wmcs(g_i, g_j)) \cup (g_j - wmcs(g_i, g_j)). \quad (2)$$

The approximate WMinCS equation between the two graphs is extended to a set of graphs, $\mathbf{G} = \{g_1, g_2, ..., g_n\}$. In other words, after ordering a set of behavioral graphs randomly, $WMinCS(g, g_i)$ $(2 \leq i \leq n)$ is iterated $n - 1$ times to compute an approximate $WMinCS(\mathbf{G})$ (Note that g is the first behavioral graph in the set after ordering G). The weights of $WMinCS(g, g_j)$ are repeatedly computed for the edges as defined in Eq.(2). Specifically, for the edges in WMinCS, the weight is equal to the sum of the weights of the isomorphic edges, divided by the number of behavioral graphs in the set (n). The weight of vertexes in the resulting WMinCS is simply set to 1, and is not used. Furthermore, the $wmcs$ of $\mathbf{G}$ after clustering all the graphs, which is shared by all of the behavioral graphs, is a particularly important subgraph which is called the *HotPath*. The weight of all edges in the HotPath is 1, since all behavioral graphs have this subgraph in common.

Finally, the approximately derived WMinCS for the set is reduced into one weighted common behavioral graph ($WCBG_\Theta(G)$) after pruning with a predefined threshold (Θ), as in the previous section. In other words, edges whose weight is less than Θ are removed, as well as vertexes isolated by their removal. The result is the Weighted Common Behavioral Graph (WCBG), a representative graph for each family of malware. The WCBG contains only edges or dependencies occurring in at least a specified fraction of the malware binaries. The WCBG summarizes all and only those properties shared by most of the malware instances, and the *HotPath* representing behavior demonstrated by all malware executions. The resulting WCBG is used for malware detection, as described in the next section. We now describe how WCBGs are used to detect new instances of malware in the second step of the method.

2.3 Malware Detection

We first generate a KOBG for a program suspected of being malicious. This KOBG of the new suspicious instance is compared with the WCBG for one malware family. An edge E_i in a WCBG is matched with an edge e_j in a behavior graph if the vertex and edge attributes of E_i and e_j are the same. The weights of all such matched edges are summed. This sum is then divided by the minimum of the number of edges in the WCBG, or in the kernel object behavior graph. The resulting measure of similarity is guaranteed

to be between 0 and 1, since all weights are also between 0 and 1.

A decision D as to whether a suspected program is malicious is made as shown in Eq.(3).

$$D = (HotPath \subseteq g_{new}) \wedge (\delta(WCBG, g_{new}) \geq \gamma) \quad (3)$$

In other words, to decide that the suspicious instance is malware, the KOBG must necessarily include the HotPath of the resulting WCBG, and the similarity between the KOBG and the WCBG must exceed a predefined threshold γ.

3. EVALUATION

This section presents the results of evaluating the proposed method. The method was implemented and used to detect a number of malware samples taken from "the wild", also used by other researchers to evaluate their work. The detection and false positive rates, and robustness of the proposed method are examined in detail.

3.1 Experiment Setup

Real-world malware samples were executed in a "sandbox" environment. Some malware instances (the training set) were used to construct WCBGs, while others (the test set) were tested to evaluate the detection and false positive rates.

As shown in Figure 3, two data sets were used in order to test the sensitivity of the method to differing inputs. The first data set, DATASET_1, consisted of 563 binaries. This set was obtained from the Anubis Project[1]; it was also used for the evaluation of the method described in [17]. In our work, the classification of binaries into malware families, and size of and assignment to training set and test set, are exactly as in [17]. DATASET_2 was collected from several well-known public websites[2], and consisted of 520 binaries. By using the Kaspersky anti-virus scanner, malware instances in the second data set were classified into malware families, as shown in Figure 3. All the binaries in the data set have Windows file format, and most of them were packed.

Family Name	Malware Type	DATASET_1		DATASET_2	
		Training Set	Test Set	Training Set	Test Set
Allaple	Exploit-based worm	50	50	N/A	N/A
Agent	Trojan	50	50	50	30
Mytob	Mass-mailing worm	50	50	50	30
Bagle	Mass-mailing worm	50	50	50	30
Mydoom	Mass-mailing worm	50	50	50	40
Netsky	Mass-mailing worm	50	13	50	40
Agobot	IRC Worm	N/A	N/A	50	50
Total Number	N/A	563		520	

Figure 3: Summary of malware families for experiment

One hundred popular Windows XP applications were used to evaluate the false positive rate of the proposed method. These included applications such as Notepad, Winzip, Winscp, Internet Explorer, Media Player, ICQ, MSN, Alzip, Emule, etc. The implementation of the proposed method was based on Ether [11], and the experimental system was a 3 GHz Intel dual core processor with 2GB of main memory.

3.2 Performance of Malware Detection

The detection and false positive rates of the proposed method were evaluated first. For the training set of each malware family, the weighted common behavior graph (WCBG) was constructed according to the method described in section 2. The WCBG for each training set was constructed with a threshold Θ equal to 0.5. The detection threshold for this purpose was set to $\gamma = 0.7$.

Family Name	DATASET_1		DATASET_2		False Positve
	Training Set	Test Set	Training Set	Test Set	
Allaple	1.00	0.90	N/A	N/A	0.00
Agent	0.60	0.38	0.82	0.83	0.00
Mytob	0.98	0.92	0.92	0.73	0.00
Bagle	1.00	0.80	0.76	0.67	0.00
Mydoom	1.00	0.90	1.00	0.88	0.00
Netsky	0.94	0.77	0.86	0.75	0.00
Agobot	N/A	N/A	0.88	0.76	0.08

Figure 4: Detection rates (D_t) and false positive rates (F_p) for the proposed method (Θ=0.5, γ = 0.7)

Detection Capability: Figure 4 shows detection and false positive rates of the proposed method. First, the ability of the WCBG to detect malware in the training set (from which the WCBG was constructed) was tested. The KOBG of each member of a training set was matched with the WCBG of its family. The detection rate for malware in the *training set* of DATASET_1 was more than 95% except for the Agent family. For this family, we could not collect system call traces completely, since most of the malware instances crashed the system; this was also reported in [17].

For the training set of DATASET_2, more than 90% of the binaries in the Mytob and the Mydoom family were correctly identified, with detection rates for other families ranging from 76-88%.

This same figure shows the effectiveness of the proposed method when applied to unknown malware instances (the test set) not included in the training set. In other words, these experiments were run to investigate how effective each WCBG was at detecting malware. In general, the detection rates for these test set members are somewhat lower than for members of the training set, as may be expected. Except for the Agent family of DATASET_1, the detection rates were over 80%. However, with the same data set in [17], this method for the test sets achieved higher detection rates than the previous work. For the test sets of DATASET_2, the proposed method showed similar performance to the results of [17]. Based on the information of kernel objects and their attributes, the method effectively detects malware variants.

The detection rates for each malware family vary from around 70% to 100%. It is possible that malware instances were often misclassified into a distinct family, as discussed in [17]. For instance, 11 of 30 malware instances classified in the Bagle family in DATASET_2 by the Kaspersky A/V product were classified into the Allaple or Trojan families by the Anubis project, and 3 samples in the set could not be classified. As discussed in [1], it is doubtful any anti-virus tool correctly classifies all instances of a malware family. Better malware classification techniques will improve the effectiveness of the proposed method.

To evaluate false positive rates for the proposed method, each binary in the set of 100 Windows application executables was tested against the WCBG of each malware family. Results are also shown in Figure 4. Only the Agobot family showed a non-zero false positive rate, while the other WCBGs all produced 0% false positive rates. The false positive rate for the Agobot family is caused by the very small number of the edges in the HotPath. That means that the binaries in the training set show a higher variety of behaviors.

The results demonstrate that the proposed method effectively and correctly identifies malware, since the common properties are very different from those of benign applications.

Comparison: DATASET_1 was also used in [17] for evaluation

[1] http://anubis.iselab.org

[2] http://vx.netlux.org, http://www.offensivecomputing.net, http://www.malwaredomainlist.com

purposes. Their work used system call traces with instruction logs to identify data flow dependencies, while the method proposed here aims to cluster a behavioral graph by using kernel objects. From this information, the method can effectively identify malware variants. The detection rates of the method proposed here were equal or higher to that of [17] for every family except the Agent family. With only system call traces, the proposed method achieved better performance with the new common kernel object graph shared in a family. It is simple, but it is robust and accurate enough to detect any variant. Details are omitted due to space limitations.

Robustness: Since the proposed method relies on system calls for identification purposes, attacks that target system call behavior may have an effect. A prime example would be the addition or injection of gratuitous system calls. These do not affect the malicious behavior of the malware, but simply attempt to obscure it.

An experiment was run to measure the influence of injected system calls on the ability of the proposed method to correctly detect malware. In this experiment, system call traces of varying lengths were copied from traces of the execution of benign (non-malware) applications. These copied traces were then pasted at random places into the system call traces of malware instances. The amount of copied system calls was varied from 0% to 100% of the length of the original malware system call traces. Following this, the significantly modified (injected) system call traces of malware instances in the training set were matched against the pre-generated WCBGs.

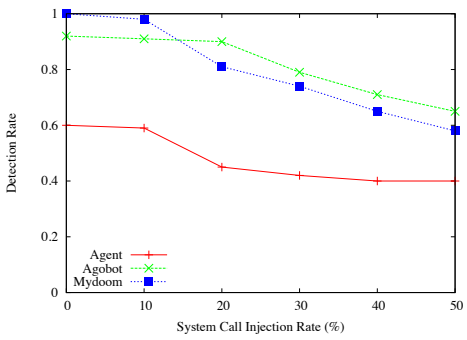


Figure 5: Detection rates against system call injection attack. (Θ=0.5, γ = 0.7)

Figure 5 shows the results. Only families actually affected by the system call attacks are shown in the figure. The Bagle, the Netsky, and the Mytob families, except for the Allaple family, were unaffected by the system call attacks, and are therefore not shown. For the affected families, the detection rates decreased according to the attack rates. However, the reduction in detection plateaus at 50%; there is no further change as the call injection rate increases to 100%. At a 40% injection rate, detection rates for these families decreased by roughly 30%.

Threshold Selection: The proposed method has two parameters, Θ (which affects the size of the common behavioral graph) and γ (which controls the sensitivity of malware detection). The system performance is not significantly impacted by Θ, since the HotPath remains the same regardless of Θ. The value of γ used in section 3 was selected based on experimentation. The value of γ is *knee of the curve* where the detection rates doesn't make much difference. By the experimental results in Section 3, Θ should be between 0.2 and 0.5, and γ should be from 0.5 to 0.9.

Other Features: An experiment was run to investigate the influence on the common behavioral graph sizes of the number of binaries used for their construction. This shows the scalability of the common behavioral graph depending on the number of binaries added. The results indicate there is virtually no change in either the size of the common behavioral graphs, or in the size of the HotPaths. In addition, the proposed method has overhead for system call tracing primarily depends on the malware analysis framework, which in our case is Ether. Compared with other dynamic behavior-based malware detection methods, the system overhead is insignificant only if system call traces are examined [17, 27]. The overhead of behavior graph construction is clearly very small (0.1 seconds or less). Details are omitted due to space limitations.

4. RELATED WORK

Behavior-based malware detection can be achieved by static analysis or dynamic analysis.

Malware analysis and detection using static disassembly has been proposed for years. The behaviors recoverable by static analysis are interpreted as "semantics" of a program. Existing methods use control flow and data flow analysis, semantics, abstract models and templates that describe the behavior of malicious programs [13, 25, 7, 14, 18, 29]. The usefulness of static analysis depends on the ability to correctly disassemble binaries, which has been shown to be problematic for packed code [11, 26].

Malware detection by monitoring the execution of a running program is a promising solution for overcoming the limitations of static analysis [5, 22]. For monitoring such behaviors, several analysis platforms have been developed recently [26, 11, 28]. Panorama [28] emulated malicious code to capture system-wide information flow based on tainted sources from the network interface or keyboard. Malspecs [6] extracted malicious behaviors, which are different from benign behaviors of benign programs, by using system call invocation. Based on taint information in instruction traces and memory logs, Inspector Gadget [16] utilized dynamic program slicing to extract a domain generation algorithm currently used in malware.

Although these approaches detect malware effectively through taint tracking, they require execution emulation and cause significant performance overhead, as discussed in [17]. Kolbitsch et. al [17] proposed an effective and efficient malware detection method based on a system call behavioral graph by using dynamic program slicing. This model captures data flow dependencies between system calls of interest. By using the crafted model, their method can be used at end hosts for detection, without resorting to taint analysis. Additionally, kernel objects at the level of the OS have also been used to analyze malicious behaviors for malware classification [2, 1]. In comparison, our method uses the relationship among kernel objects by using only system call traces. The relation was shown as a common graph, which is totally new method to detect malware with small overhead and high scalability.

5. CONCLUSION

This paper proposed a new method for detecting malicious software (malware) instances by using a kernel object behavioral graph (KOBG). The method uses kernel object behaviors to specify the behavior of malicious software, instead of relying on system calls. The KOBGs of a group of malware instances in the same family are combined into a Weighted Common Behavioral Graph (WCBG). This includes a special subgraph, the HotPath, that occurs in all instances of the family. The proposed method achieves high detection rates with very low false positive rates.

This work has the limitation shared by other methods using dy-

namic analysis, namely that they observe only partial behavior of an executable. The use of various techniques to explore most or all possible execution paths [4, 21] would improve the accuracy of this method, at the expense of greater overhead.

6. REFERENCES

[1] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, and F. J. andJose Nazario. Automated classification and analysis of internet malware. In *Proceedings of 10th International Symposium in Recent Advances in Intrusion Detection(RAID)*, volume 4637 of *Lecture Notes in Computer Science*, pages 178–197, Gold Goast, Australia, September 2007. Springer.

[2] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS)*, 2009.

[3] H. Bunke, P. Foggia, C. Guidobaldi, and M. Vento. Graph clustering using the weighted minimum common supergraph. In *Graph Based Representations in Pattern Recognition*, volume 2726 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 2003.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security(CCS)*, pages 322–335, New York, NY, USA, 2006. ACM.

[5] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 34–44, Boston, MA, USA, July 2004. ACM Press.

[6] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of ACM SIGSOFT symposium on The foundations of software engineering (FSE)*, pages 5–14, New York, NY, USA, 2007. ACM.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IN IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 32–46, 2005.

[8] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms Applications*, 11(1):99–143, 2007.

[9] D. J. Cook and L. B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[10] S. Corp. Symantec global internet security threat report, April 2008. http://www.symantec.com/.

[11] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of 31st IEEE Symposium on Security and Privacy (S&P)*, May 2010.

[13] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security(CCS'09)*, pages 611–620, Chicago, Illinois, USA, 2009. ACM.

[14] J. Kinder, S. Katzenbeisser, C. Schallhart, H. Veith, and T. U. München. Detecting malicious code by model checking. In *Proceedings of International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA)*, pages 174–187. Springer Berlin, 2005.

[15] E. KIRDA, C. KRUEGEL, G. BANKS, G. VIGNA, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th Usenix Security Symposium*, 2006.

[16] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10)*, May 2010.

[17] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *18th Usenix Security Symposium*, Montreal, Canada, August 2009.

[18] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[19] C. Krugel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, pages 207–226, 2005.

[20] M. Library. Kernel object. http://msdn.microsoft.com/en-us/library/ms724485(VS.85).aspx.

[21] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P)*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[22] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. *Computer Security Applications Conference, Annual*, 0:421–430, 2007.

[23] Y. Park, Q. Zhang, D. Reeves, and V. Mulukutla. Antibot: Clustering common semantic patterns for bot detection. In *Proceedings of 34th Annual IEEE International Computer Software and Applications Conference(COMPSAC)*, July 2010.

[24] D. Perry. Here comes the flood or end of the pattern file. In *Virus Bulletin*, Ottawa, 2008.

[25] D. Wagner and R. Dean. Intrusion Detection via Static Analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy(S&P)*, pages 156–168, Oakland, CA, USA, May 2001.

[26] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[27] B. Xin and X. Zhang. Memory slicing. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*, pages 165–176, 2009.

[28] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security(CCS)*, pages 116–127, New York, NY, USA, 2007. ACM.

[29] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, pages 411–420, 2007.