DiffSig: Resource Differentiation Based Malware Behavioral Concise Signature Generation

Huabiao Lu, Baokang Zhao, Xiaofeng Wang, and Jinshu Su

School of Computer, National University of Defense Technology, Changsha, China
{ccmaxluna,zhaobaokang}@gmail.com, {xf_wang,sjs}@nudt.edu.cn

Abstract. Malware obfuscation obscures malware into a different form that's functionally identical to the original one, and makes syntactic signature ineffective. Furthermore, malware samples are huge and growing at an exponential pace. Behavioral signature is an effective way to defeat obfuscation. However, state-of-the-art behavioral signature, behavior graph, is although very effective but unfortunately too complicated and not scalable to handle exponential growing malware samples; in addition, it is too slow to be used as real-time detectors. This paper proposes an anti-obfuscation and scalable behavioral signature generation system, DiffSig, which voids information-flow tracking which is the chief culprit for the complex and inefficiency of graph behavior, thus, losing some data dependencies, but describes handle dependencies more accurate than graph behavior by restrict the profile type of resource that each handle dependency can reference to. Our experiment results show that DiffSig is scalable and efficient, and can detect new malware samples effectively.

Keywords: Behavioral Signature, Anti-obfuscation, Scalable, Resource Differentiation, Iterative Sequence Alignment, Handle Dependency.

1 Introduction

Malware, short for malicious software, is software designed to disrupt computer operation, gather sensitive information, or gain unauthorized access to a computer system [1]. Malware includes computer viruses, worms, botnet, trojan horses, spyware, and so on. As we are know, Malware are the most serious threats in Internet for a long period. In fact, malware is the cause of most Internet problems such as spam e-mails and DoS (Denial of Service) [2].

The number of new malware samples is huge and growing at an exponential pace. Symantec observes 403 million new malware samples in 2011 [3], average 1.1 million malware samples per day. Efficiently analyzing such a scale malware samples and automated generating signatures for them is an interesting and challenging task.

Furthermore, obfuscated malware has become popular because of pure benefits brought by obfuscation: low cost and readily availability of obfuscation tools accompanied with good result of evading syntactic signature (e.g. byte-signature)

K. Mustofa et al. (Eds.): ICT-EurAsia 2013, LNCS 7804, pp. 271–284, 2013.

based anti-virus detection as well as prevention of reverse engineer from understanding malwares' true nature. Obfuscated malwares use code obfuscation in software engineering to make the samples of a malware family look differently but have the same functionality.

The obfuscation techniques commonly used in malware obfuscation are Dead-Code Insertion, Register Reassignment, Instruction Substitution and Instruction Reordering [4]. Dead-code insertion is a simple technique that adds some ineffective instructions to a program. Register reassignment switches registers from generation to generation while keeping the program code and its behavior same. Instruction substitution evolves an original code by replacing some instructions with other equivalent ones. Instruction Reordering obfuscates an original code by changing the order of its independent instructions in a random way.

A system call (syscall) is how a program requests a service from an operating system's kernel, include hardware related services (e.g. accessing the hard disk, network card), creating and executing new processes, and so on [5]. The behavior of a program can be thought as its effect on the state of the system on which it executes. Thus, treating behavior in terms of the syscalls invoked by a program is reasonable, and this allows us to succinctly and precisely capture the intent of the malware [6]. syscalls are non-bypassable interfaces, so malware intending to unsafely alter the system will reveal its behavior through the syscalls that it invokes [7]. The current perfect way to describe malware behavior is the dependencies between syscalls [8].

The dependencies between syscalls can be presented by behavior graph, in which nodes are syscalls and an edge is introduced from a node x to node y when the syscall associated with y uses as argument some output that is produced by system call x. For example, if a program has a function merging two files (A,B) into a new file (C), the behavior graph for the function is shown in the Fig. 1. Ntreadfile(A, BufferA) uses the handle of file A as source, while handle of file A is produced by Ntopenfile(A), thus, there is an edge from the node Ntopenfile(A) to node Ntreadfile(A, BufferA). And Ntwritefile(C, BufferA +BufferB) writes a buffer which is computed from buffers produced by Ntreadfile(A, BufferA) and Ntreadfile(B, BufferB), and handle of file C produced by Ntcreatefile(C), therefore, there are a edge from nodes Ntreadfile(A, BufferA), Ntreadfile(B, BufferB) and Ntcreatefile(C) to node Ntwritefile(C, BufferA +BufferB) respectively. Dependent syscalls cannot be reordered without changing the semantics [8]. Thus behavior graph can defeat various obfuscated technologies naturally. Although effective, it is unfortunately too complex to be obtained and used, and it often requires cumbersome virtual machine technology [2].

A syscall sequence is the ordered sequence of syscalls that a process performs during its execution. Syscall sequence keep the orders assigned by behavior graph, that is saying, given an edge from a syscall x to syscall y in behavior graph, syscall x must be executed before syscall y. Besides the order of syscalls decided by behavior graph, we can gain handle dependencies between syscalls. In computer, a handle is an abstract reference to a resource, such as a file, a registry key, a process, a (network) socket, or a block of memory [9]. The syscall y is

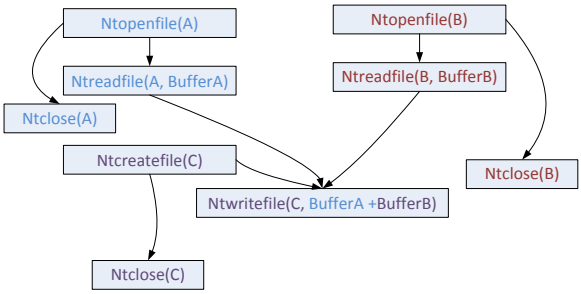

Fig. 1. Merging two files (A,B) into a new file (C)

dependent on syscall x on a handle only and only if the handle that is produced by syscall x and used by syscall y as input. we can figure out the dependency of syscalls on handles by simply checking the equality of handles in arguments of syscalls. In the Fig. 1, the dependency of Ntreadfile(A, BufferA)on Ntopenfile(A) is handle dependency. Therefore, the syscall sequence keeps the order and the dependency on handles presented by behavior graph, just losing other data dependencies that their data is modified between syscalls by mathematical or logical operations. The dependency Ntwritefile(C, BufferA +BufferB)on Ntreadfile(A, BufferA) and Ntreadfile(B, BufferB) are such data dependencies. All in all, syscall sequence is an approximate edition of behavior graph while evading strenuous information-flow tracking which is the chief culprit for the complex and inefficiency of graph behavior.

Dead-Code Insertion, Register Reassignment of malware obfuscation have no effect on syscall sequence. Instruction Substitution will evolves an original syscall sequece by replacing some syscalls with other equivalent syscalls. However, as limited syscalls in ordinary operating system, e.g., about 300 syscalls in Microsoft Windows OS, giving a syscalls block ,there is usually rare equivalent one to replace it. Therefore, syscall substitution has limited effect, we consider it in the future works. Instruction Reordering will lead to reorder the orders between independent syscalls. we present an Iterative Sequence Alignment (ISA) to perfectly defeat the syscalls reordering caused by malware obfuscation.

As executed in different environment or random tactics used, various samples of the same malware may bind the same handle to resource with different attributes, e.g. samples create files in different directories with the same purpose and writing to files with same contents, while looking the file path attributes of the handles, the handles are different and it is hard to relate them together. Thus, those differences hamper us to obtain handle dependency relation between different samples, and hamper us to gain a generalized behavior signature for those samples. We propose a resource differentiation scheme to seek common attributes of resource with same affect while hiding the differences of executing environment and the differences introduced by random tactics. The resource differentiation scheme aims to abstract the resources and to find handles with same usage from different samples. Thus it is called DiffHandle. DiffHandle is someway like DiffServ in IP QOS architecture of Internet.

After using DiffHandle to replace a handle with differentiated type of the resource that the handle reference to, and utilizing ISA to gain sub-signature (common non-consecutive ordered syscalls) set from syscall sequences of the same family, we obtain handle dependencies between syscalls in sub-signature set by backtracking those syscalls into original syscall sequences, at the same time, we gain the order restrictions between those syscalls. Finally, we generate a DiffHandle-signature consisting of syscalls that have handle dependencies –the handle can only relate to the designated type of resource– with others, and that have order restrictions between syscalls.

Comparing to behavior graph, DiffHandle-signature loses the data dependencies that their data is modified between syscalls by mathematical or logical operations, but, adds resource type (the type is defined by our resource differentiation scheme) restriction to handle dependencies. However, the most outstanding advantage of our method is avoiding complicated information-flow tacking, instead of it, gaining syscall sequence simply by hooking the SSDT (System Services Descriptor Table) table [11,16], even stirring advantage is that DiffHandle-signature can naturally be used as real-time detectors.

This paper proposes a resource differentiation based, anti-obfuscation, effective and efficient malware behavioral signature generation system, DiffSig, which is efficient, voiding complicated information-flow tacking which needs to use dynamic analysis infrastructure (virtual machine with shadow memory), generates a simplified but accurate malware behavioral signature for each malware family. In detail, the main contributions of this paper are as follows:

1. we propose a resource differentiation scheme (DiffHandle) to generalize syscall sequence by classifying resources while hiding the differences introduced by different executing environment or random tactics used by malwre.
2. we present an Iterative Sequence Alignment (ISA) to anti-obfuscation and gain sub-signature (common non-consecutive ordered syscalls) set from generalized syscall sequences.
3. we raise a concise but accurate behavioral signature presentation –DiffHandle-signature. Comparing to perfect but complicated behavior graph, DiffHandle-signature ignores some data dependencies that their data is modified between syscalls by mathematical or logical operations, but, adds resource type restriction to handle dependencies and avoids complicated information-flow tacking. It naturally suits to be used as real-time detector.

The rest of this paper is organized as follows. In section 2, we describe the system overview of DiffSig. And then we present the main three steps of DiffSig : DiffHandle, ISA and Backtracking refinement in sections 3, 4, 5 respectively. We validate our system using real malware samples in section 6. Finally, we conclude it and present future works in section 7.

2 System Overview

DiffSig consists of a kernel syscall monitor and three main steps to generate behavioral signature, as Fig. 2 shows.

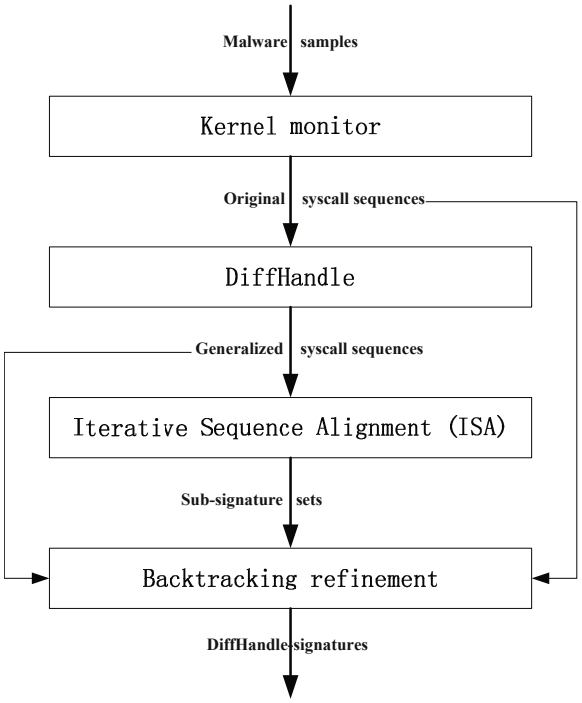

Fig. 2. Architecture of DiffSig

The kernel monitor collects syscall information for a designated process. In order to intercept and log system call information, the kernel monitor hooks the SSDT (System Services Descriptor Table) table [11,16] in Windows OS. It logs the timestamp, syscall name, arguments and return value of syscalls. Accordingly to timestamp of syscalls, all the syscalls of the process form a syscall sequence. This kind of sequence in which each syscall with raw arguments and return value is called original syscall sequence.

Resource Differentiation scheme (DiffHandle) classifies resource with same affect together, hiding the differences introduced by different executing environment or random tactics used by malware. For example, DiffHandle classifies file into: system files which generally locate under directory "C:\WINDOWS\system32"; private files which are operated only by a designated application, e.g. files under "C:\Program Files\iTunes" are private files of iTunes; and other types. The step of DiffHandle replaces a handle with differentiated type of the resource that the handle reference to, deletes the non-handle arguments of syscalls, and thus produces generalized syscall sequences in which each syscall is only with the types of resources it operates on.

Iterative Sequence Alignment (ISA) aims to extract all the common syscalls among various samples of the same family. We use sequence alignment to gain common syscalls. But, traditional sequence alignment can only gain partial

common syscalls because of syscall reordering, thus, we propose a new alignment method —Iterative Sequence Alignment (ISA). ISA applies multiple ways of traditional sequence alignment. The i^{th} way runs traditional sequence alignment on residual syscall sequences which consist of non-matched syscalls of $(i-1)^{th}$ way. ISA ends iterating when matched syscalls is less than predefined threshold. Some common non-consecutive ordered syscalls gained by each way are called as sub-signature. Multiple ways produce a sub-signature set.

Backtracking refinement finds out the handle dependencies that happens between syscalls in sub-signature set by putting those syscalls back to original syscall sequences of a same family. Backtracking refinement also obtains the order restriction among syscalls, especially, among syscalls from different sub-signatures. After the above three steps, we generate a concise but accurate DiffHandle-signature for each family.

DiffHandle-signature is formalized as a 3-tuple: $\{S, D, O\}$. $S = \{s_1, s_2, \cdots, s_n\}$ presents the common n syscalls extracted by DiffSig; $D = \{d_1, d_2, \cdots, d_m\}$ where $d_k = \{s_i, s_j, Type_{ij}\}(1 \leq k \leq m;\ s_i, s_j \in S)$ presents that there is a handle dependency which can only reference to resource type $Type_{ij}$ between syscall s_i and s_j; and the order restriction $O = \{o_1, o_2, \cdots, o_l\}$ where $o_k =< s_i, s_j > (1 \leq k \leq l;\ s_i, s_j \in S)$ presents that syscall s_i must be invoked before the syscall s_j.

3 Resource Differentiation Scheme (DiffHandle)

Resource Differentiation scheme (DiffHandle) classifies resource with same affect together, hiding the differences introduced by different executing environment or random tactics of malware. It is someway like DiffServ in IP QOS architecture in Internet. This step replaces a handle with differentiated type of the resource that the handle reference to, and delete the non-handle arguments of syscalls, produces generalized syscall sequences. We describe the differentiation scheme of three permanent resources: file, register key and network as follows.

3.1 File Differentiation Scheme

we classify all the files in Windows OS into four categories preliminarily according to their usage: System file, Private file, User file and Temporary file. Table 1 describes the four categories concisely. System files normally locate under directory "C:\WINDOWS\system32\", and are Windows system related services and data. Private files are normally only be operated by its owner application, e.g. files under the installation directory of a application is the private files of the application, thus, files under directory "C:\Program Files\" are private files to different applications, files under "C:\Documents and Settings\xxxx\Application Data" stores data of applications, and also are private files to corresponding application, another type of private file is the files under directory set by user for application , e.g. files under download directory of BitComet is private of BitComet. User files is files owns to a specified user or all users, such files

are created and modified during corresponding user active, files under directory "C:\Documents and Settings\Administrator\" are files of Administrator, and ones under "C:\Documents and Settings\All Users\" are shared among all users. Temporary files are intermediate files generated during applications execution, e.g. files under "C:\Documents and Settings\Administrator\Local Settings\Temp" are temporary files when user Administrator is active. Files not belong to above categories are treated as "other" file category.

Table 1. File Differentiation Scheme

TYPE	ID	Location	Description
System file	1	C:\WINDOWS\system32\	Windows System related execution and data
Private file	2	C:\Program Files\or others	Be operated only by its owner application normally
User file	3	C:\Documents and Settings\	Files owns to a specified user, such as Administrator
Temporary file	4	C:\Documents and Settings\ xxxx\Local Settings\Temp\	Intermediate files
Others	64	*	*

3.2 Register Key Differentiation Scheme

The most important type of register key is Autoruns. Autoruns are configured to run during system bootup or user login and they can make malwares run without any conscious or direct user interaction. Typical locations used by malwares are: "HKLM\System\Currentcontrolset\Services\%\Imagepath", "HKLM\Software\Microsoft\Windows\Currentversion\Run%", "HKLM\Software\Microsoft\Active Setup\Installed Components%", "HKLM\Software\Microsoft\Windows\Currentversion\Runonce%",and so on [15]. According to normal usage, locations are classified into seven categories preliminarily: Autoruns, HKLM\SYSTEM, HKLM\Software, HKLM\SECURITY, HKLM\SAM, HKEY_USER, HKEY_CLASSES_ROOT (HKCR). Table 2 gives a concise description to each of the seven categories. Register keys not belong to above categories are treated as "other" register key category.

3.3 Network Differentiation Scheme

We cluster network traffic into different categories according to higher-layer protocol (e.g. application-layer protocol). Network traffic are classified as Table 3. Application-layer types are identified by utilizing dynamic application-layer protocol identification technologies [14]. New category will be added as needed. The one not belong to above categories is treated as "other" network traffic.

All the resources can add new category as needed. We use longest prefix matching methods to identify which category the resource belongs to for File and Network traffic.

Table 2. Register key Differentiation Scheme

TYPE	ID	Location	Description
Autoruns	65	e.g. HKLM\Software\Microsoft\ Windows\CurrentVersion\Run	Shows what programs are configured to run during system bootup or user login
HKLM\SYSTEM	66	HKLM\SYSTEM \ HKEY_CURRENT_CONFIG\	information about the Windows system setup
HKLM\Software	67	HKLM\Software\	software and Windows settings, mostly modified by application and system installers
HKLM\SECURITY	68	HKLM\SECURITY\	linked to the Security database of the domain into which the current user is logged on
HKLM\SAM	69	HKLM\SAM\	reference all Security Accounts Manager (SAM) databases for all domains
HKEY_USER	70	HKEY_USERS\ HKEY_CURRENT_USER\	user profile actively loaded on the machine
HKCR	71	HKEY_CLASSES_ROOT	information about registered applications, e.g. file associations and OLE Object Class IDs
others	128	*	*

4 Iterative Sequence Alignment (ISA)

In this section, the most simple and basic sequence alignment, pairwise sequence, is introduced first; then we describe our Iterative Sequence Alignment (ISA).

4.1 Pairwise Sequence Alignment Algorithm

A pairwise sequence alignment is a matrix where one sequence is placed above the other to find and align common elements. Gaps ('-') are inserted to help in aligning matching characters. A mismatch occurs if elements in the same column are not identical. Fig. 3 shows results for pairwise sequence alignment between syscall sequences "O(A), R(A), CL(A),O(B), R(B), CL(B),CR(C), W(C), CL(C)" and "O(B), R(B), CL(B),O(A), R(A), CL(A),CR(C), W(C), CL(C)", where O(X) presents Ntopenfile(X), R(X) presents Ntreadfile(X, BufferX), CL(X) presents Ntclose(X), CR(X) presents Ntcreatefile(X), W(X) presents Ntwritefile(X, BufferX); assuming File A is a user file, File B is a temporary file, and File C also a user file, then the generalized syscall sequences is: "O(3), R(3), CL(3),O(4), R(4), CL(4),CR(3), W(3), CL(3)" and "O(4), R(4), CL(4),O(3), R(3), CL(3), CR(3), W(3), CL(3)". Since various samples of malware may bundling File A, B, C with different filename or file path, syscall generalization removes those differences and keep the rough profile of resource. In the rest of the paper, syscall sequences of the example are expressed in the same manner. The two syscall sequences in Fig. 3 are possible syscall sequences according to behavior graph described in Fig. 1. The common signature produced by the pairwise sequence alignment is "O(4)R(4)CL(4)*CR(3)W(3)CL(3)", which loses the operations on another user file.

Table 3. Network Differentiation Scheme

TYPE	ID	Description
DNS requests	129	*
HTTP-traffic	130	*
IRC-traffic	131	*
SMTP-traffic	132	*
P2P-traffic	133	*include some popular P2P types
FTP-traffic	134	include FTP-control, FTP-data and TFTP
SSL-traffic	135	associated to HTTPS connections
NetBIOS	136	*
TCP-traffic	137	TCP traffic other than those above-mentioned
UDP-traffic	138	UDP traffic other than those above-mentioned
ICMP-traffic	139	*
Listen	140	Listen on a port
others	192	*

4.2 Iterative Sequence Alignment (ISA)

In the above section, we introduce traditional one way sequence alignment which will lose some reordered syscalls. So, we propose a new sequence alignment mode, iterative sequence alignment, to defeat syscall reordering introduced by malware obfuscation. Fig. 4 shows results for iterative sequence alignment between syscall sequences "O(3), R(3), CL(3),O(4), R(4), CL(4),CR(3), W(3), CL(3)" and "O(4), R(4), CL(4),O(3), R(3), CL(3),CR(3), W(3), CL(3)". There are two ways alignment for the two sequences. The first way alignment is the same as traditional one way pairwise sequence alignment, and the second way alignment first obtain the mismatched elements in the previous way, then apply traditional sequence alignment on those mismatched elements . Thus, the sub-signature set is {O(4)R(4)CL(4)*CR(3)W(3)CL(3), O(3)R(3)CL(3)*}, it covers all the operations common in the two sequences. We define the regular iterative sequence alignment formally in the following.

O(3)	R(3)	CL(3)	O(4)	R(4)	CL(4)	-	-	-	CR(3)	W(3)	CL(3)
-	-	-	O(4)	R(4)	CL(4)	O(3)	R(3)	CL(3)	CR(3)	W(3)	CL(3)

Fig. 3. Example of Pairwise Sequence Alignment between Syscall Sequences

Definition 1. Iterative Sequence Alignment Problem
INPUT: A generalized sequence set $W = \{g_1, g_2, \cdots, g_{nw}\}$, a matched syscalls threshold to end the iteration: θ_{score}, and a noise threshold θ_{noise}.
OUTPUT: a sub-signature set Sig_w common among $(\lceil(1-\theta_{noise})(nw)\rceil)$ sequences of W. Sig_w satisfies the Formula 1 and $Score(SubSig_i)$ satisfies the matched syscalls threshold. $Score(SubSig_i)$ is the number of matched syscalls in i^{th} way sequence alignment.

$$\begin{aligned} &\textbf{Maximize} \quad Score(Sig_w) = \textstyle\sum_j (Score(SubSig_j)) \\ &\textbf{subject to} \quad Score(SubSig_j) \geq \theta_{score} \end{aligned} \tag{1}$$

The first way alignment:

O(3)	R(3)	CL(3)	O(4)	R(4)	CL(4)	-	-	-	CR(3)	W(3)	CL(3)
-	-	-	O(4)	R(4)	CL(4)	O(3)	R(3)	CL(3)	CR(3)	W(3)	CL(3)

The second way alignment:

	O(3)	R(3)	CL(3)	*
*	O(3)	R(3)	CL(3)	*

Fig. 4. Example of iterative Sequence Alignment between Syscall Sequences

5 Signature Refinement by Backtracking into Original Syscall Sequences

Result of iterative sequence alignment on the example is {O(4)R(4)CL(4)*CR(3) W(3)CL(3), O(3)R(3)CL(3)*}. However, we do not know whether the handle of CR(3) and the one of R(3) are same, syscall CR(3) and R(3) are handle dependable, and syscall R(3) must execute before W(3) from the result of ISA. This is what our signature refinement step do.

Definition 2. Signature Refinement Problem
INPUT: the selected $(\lceil(1-\theta_{noise})(nw)\rceil)$ generalized sequences $SG = \{sg_1, sg_2, \cdots, sg_{\lceil(1-\theta_{noise})nw\rceil}\}$, and corresponding $(\lceil(1-\theta_{noise})(nw)\rceil)$ original sequences $SO = \{so_1, so_2, \cdots, so_{\lceil(1-\theta_{noise})(nw)\rceil}\}$. Result of ISA: sub-signature set $SubSigSet = \{subsig_1, subsig_2, \cdots, subsig_{n\_isa}\}$ where each sub-signature $subsig_j = (syscall_{j1}, syscall_{j2}, \cdots, syscall_{jn_j})(1 \leq j \leq n\_isa)$ is some ordered syscalls that $syscall_{j(i-1)}$ must be invoked before $syscall_{ji}$.
OUTPUT: a DiffHandle-signature $FinalSig = \{FS, FD, FO\}$ that satisfies as Formula 2. DiffHandle-signature $FinalSig$ matches a original sequence sg_i means that there exists a map function which maps each handle type in FD to a handle

in sg_i and maps each syscall $s_x \in FS$ to a syscall s_y in sg_i, satisfying that for each element $\{s_i, s_j, Type_{ij}\} \in FD$, the mapper of s_i and the mapper of s_j are handle dependency in sequence sg_i and the type of the dependency is $Type_{ij}$, and for each element $< si, sj > \in FO$, the mapper of s_i stands before the mapper of s_j in sequence sg_i.

$$\begin{aligned} &\textbf{Maximize} \quad \|FD\| \\ &\textbf{subject to} \quad \forall syscall_k \in FS, \exists subsig_j \; that \; syscall_k \in subsig_j \\ &\textbf{and} \quad FinalSig \; matches \; all \; sg_i \; (1 \leq i \leq \lceil (1 - \theta_{noise})(nw) \rceil) \end{aligned} \tag{2}$$

6 Evaluation

In this section, we compare our DiffSig with previous famous system, Hamsa [12]. And verifies the effectiveness of DiffSig to detect new malware samples. we set following default parameters unless otherwise specified: matched threshold to end the iteration θ_{score} is 10, and a noise threshold θ_{noise} is 30%.

6.1 Data Set

We obtained a set of malware executables from mwanalysis.org [10] in the period from January 16, 2011 to March 21, 2011. According to the scan results of kaspersky anti-virus, we cluster the malware executables into different families. As a result, we gain 8 families and 331 malware executables described in Table 4 . The column "Family Name" is the scan results of kaspersky anti-virus pruning the variant name. Excepting detection result by heuristic methods, normally, the scan results consists of family name and variant name, while variant name is the string locating behind the last '.' in the scan result. E.g. in scan result of malware executable (MD5: 763ceb3a9127a0b9fac1bcf99a901d19): "Backdoor.Win32.Bifrose.usc", "Backdoor.Win32.Bifrose" is the family name, and "usc" is the variant name. The column "NUM of variants" presents the number of different variant names in a family. The column "NUM of executables" describes the number of different MD5 values in a family.

Since the kernel module hooking SSDT table not implemented yet, we utilize WUSStrace[13] to gain syscall sequences with detailed information. Besides running malware executables with WUSStrace to gain malicious syscall sequences, we also running normal application with WUSStrace to gain benign syscall sequences. Those normal programs contains firefox, iTunes, BitComet, Windows Office Word, realplay, FoxitReader, Skype, PartitionTableDoctor3.5, NokiaPCSuite7, PersonalBankPortal, Win32python2.7, and so on, totally 22 applications.

6.2 Performance of DiffSig

we compare our DiffSig with Hamsa [12] on above section mentioned malicious syscall sequences of malware executables and benign syscall sequences of normal

Table 4. The Families and Characteristics of Malware Executables

Family Name	NUM of variants	NUM of executables
Trojan-Downloader.Win32.CodecPack	13	48
Trojan-Dropper.Win32.VB	12	28
Trojan.Win32.VBKrypt	50	79
Worm.Win32.VBNA	14	75
Trojan.Win32.Refroso	21	27
Trojan.Win32.Cosmu	9	26
Trojan.Win32.Scar	7	21
Backdoor.Win32.Bifrose	19	27
Total	145	331

Table 5. The Performance of Our DiffSig Comparing to Hamsa

Family Name	Number of signatures		Detection rate		False Positive	
	DiffSig	Hamsa	DiffSig	Hamsa	DiffSig	Hamsa
Trojan-Downloader. Win32.CodecPack	1	3	95.8%	79.2%	0.045	0.135
Trojan-Dropper.Win32.VB	1	1	100%	78.6%	0	0
Trojan.Win32.VBKrypt	2	6	93.7%	88.6%	0	0
Worm.Win32.VBNA	2	5	93.3%	86.7%	0.045	0.18
Trojan.Win32.Refroso	1	4	92.6%	74.1%	0	0.090
Trojan.Win32.Cosmu	1	2	88.4%	76.9%	0	0
Trojan.Win32.Scar	1	—	90.4%	—	0	—
Backdoor.Win32.Bifrose	1	—	85.1%	—	0.045	—
average	1.25	3.5	92.4%	80.1%	0.0169	0.068

applications. Hamsa [12] extracts syscall token (consecutive syscalls) set,to form a signature as a set of common tokens, it aims to cover the most sequences under predefined false positive. Malicious syscall sequences are cut into two parts: one part as training set to generate behavioral signature and the other part as testing set to test the effectiveness of signature generated. Each part has half number of MD5 values and half number of variants.

Table 5 shows the performance of our DiffSig and Hamsa. The column "Number of signatures" describes the number of signatures generated by DiffSig and Hamsa respectively; The column "Detection rate" shows the detection ability of signatures to detect malware executables in testing set; The column "False Positive" presents the rate of treating normal application as malicious by

applying generated signatures to benign syscall sequences. From table 5, we can see that our DiffSig can generate signature for any family while Hamsa can not generate signature for Trojan.Win32.Scar and Backdoor.Win32.Bifrose because the two families have no invariable token in their syscall sequences. And we can see that our DiffSig produces fewer signatures since our Diffhandle-signature is more generalized, that the detection rate of our DiffSig is much higher and the false positive rate is much lower. In summary, our DiffSig generates more generalized and accurate signature, and our Diffhandle-signature is more suitable to anti-obfuscation than token set signature of Hamsa. Comparing the computing performance and detection performance with current behavior graph methods is our future work.

7 Conclusions

To surmount the complication of behavior graph but still to keep its effectiveness. We propose an anti-obfuscation and scalable behavioral signature generation system, DiffSig, which voids information-flow tracking which is the chief culprit for the complex and inefficiency of graph behavior, but describes handle dependencies more accurate than graph behavior by restrict the profile type of resource that each handle dependency can reference to. DiffSig is effective and efficient, generating concise but accurate signatures. Comparing with previous famous signature generation system, Hamsa, our DiffSig produces fewer signatures, detects more malware samples, and has lower false positive. Comparing the computing performance and detection performance with current behavior graph methods is our future works.

Acknowledgments. This work is supported by Program No. IRT 1012, the NSF of China Program No. 61202488, the Research Fund for the Doctoral Program of Higher Education of China No. 20124307120032, the NSF of China Program No. 61103194, and the NSF of China Program No. 61003303. We appreciate anonymous reviewers for their valuable suggestions and comments.

References

1. Wikipedia, `http://en.wikipedia.org/wiki/Malware`
2. Clemens, K., Paolo, M.C., Christopher, K., Engin, K., Xiaoyong, Z., Xiaofeng, W.: Effective and efficient malware detection at the end host. In: USENIX Security 2009, USENIX Press (2009)
3. Wikipedia, `http://www.symantec.com/threatreport/`
4. You, I., Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications (2010)
5. Wikipedia, `http://en.wikipedia.org/wiki/System_call`
6. Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X.: Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy (2010)

7. Srivastava, A., Lanzi, A., Giffin, J.: System Call API Obfuscation (Extended Abstract). In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 421–422. Springer, Heidelberg (2008)
8. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering (2007)
9. Wikipedia, http://en.wikipedia.org/wiki/Handle_(computing)
10. mwanalysis, http://mwanalysis.org/
11. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows kernel. Addison Wesley Professional (2005)
12. Li, Z., Sanghi, M., Chen, Y., et al.: Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In: IEEE Symposium on Security and Privacy (2006)
13. http://code.google.com/p/wusstrace/
14. Dreger, H., Feldmann, A., Mai, M., Paxson, V., Sommer, R.: Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In: 15th USENIX Security Symposium (2005)
15. Bayer, U., Habibi, I., Balzarotti, D.: A View on Current Malware Behaviors. In: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET 2009 (2009)
16. Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: AccessMiner: Using System-Centric Models for Malware Protection. In: CCS 2010. ACM Press (2010)