Growing Grapes in Your Computer to Defend Against Malware

Zhiyong Shan and Xin Wang

***Abstract*— Behavior-based detection is promising to resolve the pressing security problem of malware. However, the great challenge lies in how to detect malware in a both accurate and light-weight manner. In this paper, we propose a novel behavior-based detection method, named growing grapes, aiming to enable accurate online detection. It consists of a clustering engine and detection engine. The clustering engine groups the objects, e.g., processes and files, of a suspicious program together into a cluster, just like growing grapes. The detection engine recognizes the cluster as malicious if the behaviors of the cluster match a predefined behavior template formed by a set of discrete behaviors. The approach is accurate since it identifies a malware based on multiple behaviors and the source of the processes requesting the behaviors. The approach is also light-weight as it uses OS-level information flows instead of data flows that generally impose significant performance impact on the system. To further improve the performance, a novel method of organizing the behavior template and template database is proposed, which not only makes the template matching process very quick, but also makes the storage space small and fixed. Furthermore, the detection accuracy and performance are optimized to the best degree using a combinatorial optimization algorithm, which properly selects and combines multiple behaviors to form a template for malware detection. Finally, the approach novelly identifies malicious OS objects in a cluster fashion rather than one by one as done in traditional methods, which help users to thoroughly eliminate the changes of a malware without malware family knowledge. Compared with commercial antimalware tools, extensive experiments show that our approach can detect new malware samples with higher detection rate and lower false positive rate while imposing low overhead on the system.**

***Index Terms*— Malware detection, behavior, OS-level information flow.**

I. Introduction

THOUSANDS of new malware samples emerge on the Internet everyday. Analyzing a malware instance to create a detection signature requires substantially greater effort than that required by generating a new malware. This is especially the case when easy-to-use malware toolkits automatically create hundreds of unique variants to run on the Internet [1][25]. As a consequence, using a traditional signature-based detector to combat malware is becoming more difficult.

Manuscript received September 1, 2012; revised January 7, 2013; accepted November 5, 2013. Date of publication November 13, 2013; date of current version January 13, 2014. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. C.-C. Jay Kuo.

Z. Shan is with the Department of Computer Science, Renmin University of China and Purdue University, Beijing 100872, China (e-mail: zhiyongshan@gmail.com).

X. Wang is with the Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794 USA (e-mail: xwang@ece.sunysb.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2013.2291066

Behavior-based detection techniques can provide promising alternative solutions to the growing malware problem. Unlike signature-based techniques that examine the syntactic pattern of a program's binary, behavior-based techniques focus on the actual actions that the program performs in the system to access system services or resources. Hence, behavior-based detectors are difficult to be bypassed by obfuscations or polymorphisms that are used to evade signature-based detectors [2]. Moreover, as featured malware behaviors are often shared by a family of malware instances instead of pertaining to only an individual instance, behavior-based detectors are able to detect previously unseen malware instances and avoid the need of a large database of signatures to identify each known piece of malware instance [32].

However, the grand challenge in behavior-based malware detection is how to perform detection accurately (i.e., low false positive rate) with a low performance overhead. Commercial anti-virus tools often have a module for monitoring malicious behaviors, which is light-weight but not accurate [4][28]. The module only leverages a single system call and the parameters to determine a malicious program. For example, intercepting NtSetValueKey() and analyzing the arguments to determine whether a program is trying to modify a security sensitive registry key, and then popping up an alarm window when this is true. As a consequence, such a module imposes small performance overhead on the system but at the same time produces frequent false alarms that annoy users. Many users even simply disable the behavior monitoring module.

On the other hand, state-of-the-art behavior-based malware detectors [1][22][25][30][29] significantly improve the detection accuracy at the cost of heavy overhead on the system. They extract dependencies among system calls to construct dependency graphs, and match the activities of a program with predefined dependency graphs to determine if it is a malware. Extracting dependencies requires tracing data flow which significantly slows down the system and needs virtual machine technology to support. Moreover, matching dependency graph requires a complex algorithm that further slows down the system especially when the number of predefined dependency graphs is large in a real application scenario.

Therefore, existing detection technologies can not work effectively online, since they are cumbersome or inaccurate. By carefully analyzing existing technologies, we find that they commonly determine whether a program belongs to a specific malware family based on implementation-specific artifacts such as byte sequences and dependencies among system calls. When the artifacts are simple, the detectors are light-weight but not accurate. On the other hand, creating more accurate detectors with more complex artifacts would incur a heavy

overhead. Moreover, existing technologies often target to identify the exact family of a malware rather than simply discriminate a malware from benign software. Identifying a malware family is useful when cleaning up the impacts of a malware, but the need of a more complex specification to recognize the malware family affects the performance of the system.

In this paper, we devise a novel malware detector, named Growing Grapes, which can achieve accurate malware detection without incurring high overhead. Our proposed detector consists of a clustering engine and a detection engine. The clustering engine correlates suspicious objects into a number of clusters by tracking OS-level information flows and attaching a cluster label to each object. Each of the obtained clusters contains either all benign objects or all malicious objects. The detection engine determines a malicious cluster by matching a predefined behavior template. A **behavior template** consists of a group of independent **atomic behaviors**, each of which serves for different malware intent, for example, hiding itself from users or disabling anti-virus tools. Each atomic behavior consists of a system call and their arguments (i.e., host system details). All templates are stored in a behavior template database.

In order to have an accurate online detector, we propose two techniques. First, to achieve accurate detection, we use a simulated annealing algorithm to optimally select a set of behaviors to form a behavior template to identify a single malware. With the optimal combination of behaviors in a behavior template and the combination of templates in the template database, our malware detector can identify the maximum possible number of malware samples while incurring the minimum false positive rate. To further reduce the false positive rate, we implicitly take into account the source of the processes launching the behaviors when determining a malicious cluster.

Second, we take two means to achieve online detection: 1) We devise a novel method to correlate system calls of a malware's all processes using light-weight OS-level information flows rather than traditional data flows; 2) We design a novel structure for the template database. The database occupies a small and fixed-size of memory (about 21K) even when the number of templates contained increases up to millions. With the support of the database, a template searching and matching algorithm becomes very simple, and it only needs to simultaneously test 45 bits within one operation.

As the detection process is based on the clustered objects, Growing Grapes novelly identifies malicious OS objects in a cluster fashion, rather than one by one as done in traditional malware detection and analysis methods. With this feature, even an ordinary author can know how to effectively remove the malware without the knowledge of malware family. This is especially important for cleaning up sophisticated malware that consists of multiple processes or executables which monitor and restore each other, because only removing one or a part of them can not really disable the malware.

Experiments show that our approach can effectively detect 71.1% of unknown malware samples without false positives, while only imposing a small overhead on the system. Compared with commercial antivirus software, it achieves higher malware detection rate and lower false positive rate.

The contributions of this paper are four-fold:

1) We propose a novel behavior-based malware detection approach for online application. It leverages OS-level information flow, optimized behavior templates and a condensed template database to achieve accurate online detection.
2) To make the detection process efficient, we design a novel concise template database that can store a large number of templates without increasing the memory occupation and time of querying a template in the database.
3) We propose a novel clustering approach to collect the objects of a suspicious program together. Thus we can conveniently monitor a malware's complete behaviors across different processes and help users to eliminate all changes of a malware without malware family knowledge.
4) We have implemented the approach in Windows kernel and the testing results have verified its effectiveness.

In the rest of the paper, we first describe the Growing Grapes approach for detecting malware in Section II, and then introduce its implementation in Windows kernel in Section III. The prototype is evaluated in Section IV. Last, we present the related work and conclude our work in Section V and VI respectively.

II. Growing Grapes Approach

A. Overview

Our Growing Grapes approach consists of clustering engine and detection engine. The clustering engine groups the objects of a program together into a cluster. The detection engine decides whether the cluster is malicious by monitoring all behaviors of the cluster. If a cluster's behaviors match a predefined behavior template in the template database, then it is identified as malicious. The behavior template database has a novel structure to minimize the memory and runtime overheads. Moreover, the templates in the database are optimized to reduce false positives and negatives.

To illustrate how a specific piece of malware is detected in the system, we use a malware example "Worm.Win32.Lovesan.a". Growing Grapes correlates the processes and executable files of the malware into a cluster by tracing OS-level information flow, which is shown in Figure 1(a). Meanwhile, Growing Grapes monitors and records all atomic behaviors of the processes, which are shown in Figure 1(b). When a behavior appears, Growing Grapes searches the behavior template database using the current behavior and the history behaviors of the cluster, which is shown in Figure 1(c). Once matching a template, Growing Grapes alarms a malware.

B. Clustering Engine

The clustering engine clusters together the suspicious objects of a program. **Suspicious objects** are the ones that derive from the Internet or removable drives, and are thus suspected to be malicious. Suspicious objects only include processes and executable files because a process is possibly

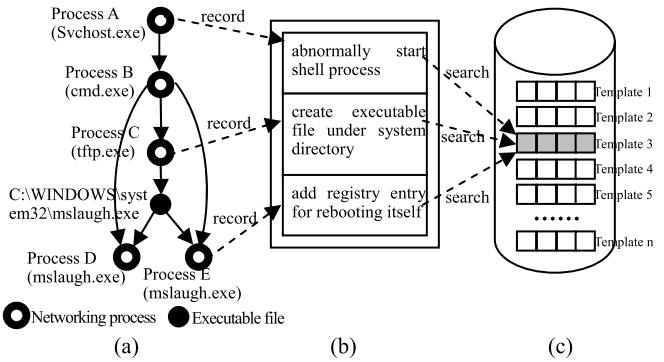

Fig. 1. A malware example. (a) Cluster. (b) Atomic behaviors. (c) Template database.

the agent of an intruder and an executable file determines the execution flow of a process which represents an intruder. Based on the cluster, we can completely monitor all atomic behaviors of a program and perform accurate detection by using multiple atomic behaviors to identify a single malware. Moreover, the cluster can help users to clean up the malware without knowing the exact family of the malware.

The challenge is how to correlate suspicious objects together into clusters in a light manner. Since objects of a malware often have various types and are scattered all over the system, it is difficult to associate them together. We observe that objects of a malware can be correlated together by tracing OS-level information flows, and at the same time the malicious objects can be clearly separated from the other objects through a proper way of attaching cluster labels to them. Accordingly, we devise a novel approach to correlate suspicious objects into clusters, which includes root rules, spreading rules and clustering rules. These rules are explained in details in the following subsections.

1) Root Rules: As all malwares come from either the network or removable drives, we design root rules to mark the objects from the network or removable drives as suspicious. These objects are start-points to trace suspicious objects.

- **Root Rule A:** *Marking processes which conduct remote communications as suspicious;*
- **Root Rule B:** *Marking executables (i.e., executable file) located at removable drives as suspicious.*

An executable in this paper represents an executable file with a specific extension, such as .EXE, .COM, .DLL, .SYS, .VBS, .JS, BAT, etc, or a special type of data file that can contain macro codes, say a semi-executable, such as .DOC, .PPT, .XLS, .DOT, etc. Growing Grapes does not allow a suspicious process to change the extension of a file in order to prevent its potential evasion of tracing. With these two rules, all malwares that attempt to enter the system can be tracked as there are only two ways for them to break into system, either through network communications or through a removable drive.

2) Spreading Rules: To track OS-level information flow, BackTracker [10] is a successful approach. However, the major challenge is how to make sure that it won't get the entire system marked as suspicious while at the same time preventing malwares to escape from tracing. This needs to trade off between reducing the number of marked objects and reducing the risk of malware evasion. Our approach is to trace preferentially the information flows with a high risk of propagating malwares while pruning the information flows with a low risk. Based on this principle, we have following rules to mark related objects as suspicious.

- **Spreading Rule A:** *Marking executable files created or modified by a suspicious process as suspicious;*
- **Spreading Rule B:** *Marking processes spawned by a suspicious process as suspicious;*
- **Spreading Rule C:** *Marking processes loading a suspicious executable file or reading a suspicious semi-executable or script file as suspicious;*
- **Spreading Rule D:** *Marking processes receiving data from a suspicious process through a dangerous IPC as suspicious.*

As an executable represents an inactive malware while a process represents an active malware, the information flows presented in these four rules have a high possibility of propagating malwares. Thus, to track the information flows with a high risk of propagating malwares, the spreading rules focus on tracing executables and processes. In the Spreading Rule C, Semi-executable and script file possibly contain malwares (e.g., macro virus in MS Word), and thus the processes reading them need to be marked. Although the macro virus protection in Office software can reduce the chances of macro virus infection, relying on it is very dangerous as crafted macro codes are able to subvert it and cause destructive damages. This has been observed in virus Melissa and W97M.Dranus.

To prune the information flows which have a low risk of propagating malwares, the spreading rules do not trace most reading and writing operations on ordinary files, directories and registry entries, which are frequently invoked but difficult to propagate malwares. However, subtle malwares might evade tracing by changing registry entries or configuration files which subsequently affect the processes reading them, so as to run malicious executables, escalate privileges, impose damages on system, etc. No matter what evasion schemes the malwares utilize, they need to run their own executables to perform the tasks, which are downloaded from the network, copied from removable drives, or obtained from changing local executables. Since all executable related operations are thoroughly traced by the Spreading Rule A and C, the malwares will be captured whenever trying to load their executables. The two rules are applicable to all existing malwares because they rely on their own executables to perform malicious tasks on a host, according to our analysis on Symantec Threat Explorer [7]. In case that a malware relies only on benign programs to perform attacks, the Root Rule A still can capture it when it requires a remote communication to accept commands to exploit the benign program to perform the malicious tasks. In addition, for a few special registry entries and configuration files that can be used by a malware to fool a benign program to execute arbitrary commands, Growing Grapes forbids a suspicious process to modify them. Therefore, although the operations on registry entries or configuration files are not traced, malwares still can not evade being detected by Growing Grapes.

To reduce the number of marked processes, the spreading rules only trace dangerous IPCs (Inter-Process

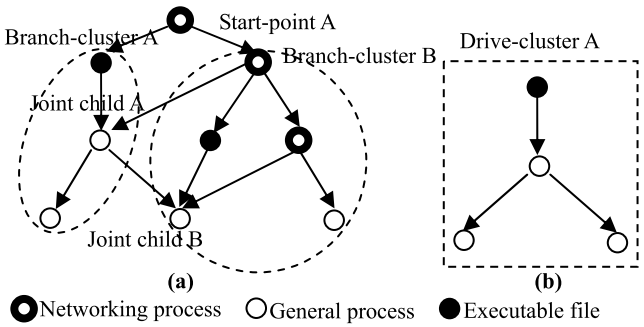

Fig. 2. Dependency graphs and the obtained clusters. (a) Branch Clusters. (b) Drive Clusters.

Communication). According to our investigation on Microsoft Security Bulletins [6], a primary source for analyzing attack vectors of Windows OS [5], the overwhelming majority of vulnerable IPCs can only be used to launch denial-of-service attack, disclose sensitive information, or escalate the privileges of the processes that send IPC data, rather than take control of the receiver process. Accordingly, they can not be used to propagate malwares. Moreover, IPCs that can propagate malwares often rely on network (e.g., Remote Procedure Call) and thus are traced by the Root Rule A. Consequently, we employ a Dangerous-IPC-List to trace dangerous IPCs since there are very few dangerous IPCs in a Windows OS.

3) Clustering Rules: Based on the spreading rules, the suspicious objects are actually connected to each other by information flows and form an existent but invisible dependency graph, which had been disclosed by the literature [10]. The graph is a directed graph and has a root node. Its nodes represent OS objects, e.g., a file, a process. Its edges represent information flow related operations, e.g., creating a process, modifying a file. Figure 2 (a) and (b) show two dependency graphs which are derived from a networking process and an executable file respectively.

The clustering rules are responsible for dividing the dependency graph into sub graphs, i.e., clusters. Note that, we do not intend to really generate dependency graphs to help cluster objects since this would not be applicable to an online approach. Instead, the clustering rules are implemented together with the spreading rules as follows: when an object is determined as suspicious by clustering or spreading rules, a proper cluster label, i.e., a number and a time stamp, will be attached to it at the same time in order to denote that it is a suspicious object and belongs to the cluster identified by the label. In other words, the clustering rules are enforced along with the spreading rules in real-time, rather than generating a dependency graph and then analyzing it.

When a root object is a network facing process, its dependency graph is too coarse-grained to be used to recognize malicious objects in a cluster fashion since it might contain both benign and malicious objects. In other words, we can not determine that all objects in a graph are malicious even if most of the objects in the graph are malicious. Thus, we must partition the graph into a number of sub graphs, say clusters, so that each cluster contains either only benign or only malicious objects.

According to the recent research [31][27] and our analysis on a huge number of malware descriptions in the Symantec Threat Explorer [7], malwares break into a host through three basic attack channels. The first is that, malwares exploit bugs in network-facing daemon programs or client programs and compromise them, then immediately spawn a shell or back-door process [31]. After this, the attacker tries to download and install attacking tools, as well as performs any other adversary actions. Accordingly, we have the following rule:

- **Clustering Rule A:** *Attaching a cluster label to a process and its descendants if the process is directly spawned by a network-facing process.*

We call this type of cluster a **branch cluster**, e.g., the Branch-cluster B in Figure 2 (a). A branch cluster corresponds to a sub graph of a dependency graph which roots from a network-facing process.

The other attack channel is that, malwares increasingly use social engineering to lure users into downloading and launching them [27]. After started, malwares copy themselves and make themselves resident in a host. Consequently, we have the following rule:

- **Clustering Rule B:** *Attaching a cluster label to a downloaded executable and its descendants.*

We also call this type of cluster a **branch cluster**, e.g., the Branch-cluster A in Figure 2 (a). The last channel is removable drives. Accordingly, we have the following rule:

- **Clustering Rule C:** *Attaching a cluster label to an executable file located on a removable drive and all its descendent objects.*

We call this kind of cluster a **drive cluster**, e.g., the Drive-cluster A in Figure 2(b).

Another issue for labeling objects is about a joint child who has multiple parent nodes in a dependency graph, e.g., the joint children A and B in Figure 2 (a). That is, when the parent nodes belong to distinct clusters, we have to determine the cluster label of the joint child. Basically, we make decision according to the priority sequence like "process' executable $\rightarrow$ parent process $\rightarrow$ other objects". Obviously, the joint child should inherit the cluster label from its parent process or executable file (i.e., a process' image file) if either of them exists instead of other objects. Moreover, as loading an executable is posterior to creating a process and necessarily overwrites the newly created process' code segment, the new process' activity is based on the loaded executable. Hence, the joint child should inherit the label from the loaded executable rather than the parent process if both exist. If more than one parent node has the same priority in the sequence above, the child inherits their labels in the reverse time order. Consequently, the joint children A and B are classified into Branch-cluster A and B respectively, as shown in Figure 2(a).

On the other hand, when splitting a dependency graph into different branch clusters, a sophisticated malware might intentionally separate an ASEP (Auto-Start Extensibility Point [8]) pair into two different clusters. Then, the two clusters work together to perform malicious actions and potentially evade Growing Grapes' detection. An ASEP is used to enable auto-starting of programs without an explicit user invocation, and

thus becomes a common target of infection by malwares. An ASEP pair represents an ASEP and the corresponding executable file. To mitigate this issue, we periodically scan the clusters to see whether there are split ASEP pairs, and combine the related clusters together if found.

4) Verification: With the above methods, an obtained cluster will consist of either all benign or all malicious objects. It is not possible that a cluster contains both benign and malicious objects, because if a cluster contains a malicious object, all objects in the cluster should also be malicious, which can be proved as follows:

Given a cluster $c = (V, E)$, where V is a set of vertices and E is a set of directed edges connecting the vertices. The vertices represent OS objects. We use v_r to represent the root node of the cluster c, which is the ancestor vertex of all vertices in the cluster. An edge (v_{n-1}, v_n) represents an OS-level information flow related operation that propagates malwares from the parent (source) object v_{n-1} to the child (destination) object $v_n \cdot v.f = benign|malicious$ represents that the corresponding object is benign or malicious.

(1) If the root object of the cluster c is malicious, then all objects in the cluster should be malicious. As the root object is the ancestor of all vertices in the cluster, for an arbitrary vertex v_n in the cluster, there at least exists a propagation path $\{(v_r, v_1), (v_1, v_2), (v_2, v_3), \ldots\ldots (v_{n-1}, v_n)\}$ that propagates malware from the root to the vertex v_n. According to the spreading rules presented previously, a malicious object can make an object to be malicious by executing a propagation operation: $(v_{m-1}, v_m) \epsilon E \wedge v_{m-1}.f = malicious$ So, if the root object is malicious, then $v_r.f = malicious \wedge (v_r, v_1) \epsilon E \rightarrow v_1.f = malicious, v_1.f = malicious \wedge (v_1, v_2) \epsilon E \rightarrow v_2.f = malicious$, ……, $v_{n-1}.f = malicious \wedge (v_{n-1}, v_n) \epsilon E \rightarrow v_n.f = malicious$. So, any vertex in the cluster should be malicious.

(2) If any object $v_n \neq v_r$ in the cluster is malicious, the root vertex v_r should be malicious and there should be a propagation path to propagate malware from the root to the vertex v_n, because the root is the only source to introduce malware into the cluster. Then, according to the result of (1), all objects in the cluster are thus malicious since the root object is malicious.

Therefore, if any of the objects in the cluster is malicious, all the objects should be malicious. In other words, a cluster contains either only benign or only malicious objects. Our experiments in Section IV further demonstrate the effectiveness of the labeling approach.

C. Detection Engine

The detection engine performs detection tasks by monitoring the activities of each cluster. To obtain an accurate detection, the engine decides a malicious cluster using multiple atomic behaviors rather than a single atomic behavior. The multiple atomic behaviors used to determine a malware serve as a predefined behavior template in a behavior template database that belongs to the detection engine.

At a high level, using multiple atomic behaviors to identify a malware is in accordance with the recent work [1] that uses multiple significant behaviors. An atomic behavior is often the core of a significant behavior, because a significant behavior is represented by a dependency graph that includes a mission-critical system call as the core step. Moreover, the result from another recent work [24] also supports our idea of using multiple behaviors. The result shows that a set of discriminative operations, e.g., delete_file, create_mutex, etc, can be used to recognize a malware family. This actually proves that multiple operations can be used to effectively detect a malware.

The challenge to building the detection engine is three folds. The first is how to extract proper atomic behaviors that reflect the intent of the malware authors from system-specific details. The second is how to construct behavior templates that can effectively detect known and unknown malware with a small number of false positives. The last is how to design an online mechanism to efficiently match the behavior templates. We present our strategies for addressing these three challenges in the following three subsections.

1) Defining Atomic Behaviors: We define atomic behaviors based on the analysis of the OS resources that are most likely being attacked. Monitoring the operations that manipulate such resources can efficiently capture the real intent of the attackers. Consequently, the obtained atomic behaviors depend only on the system details instead of malware details. Such behaviors are most possibly taken by malware authors across different malware families and thus can be applied to detect unknown malwares.

An atomic behavior consists of an operation, the manipulated object and the necessary arguments, which is critical to fulfill a malicious intent, e.g., modifying registry key value for surviving reboot. When the behavior is too specific, e.g., presenting the exact name of the registry key and value, the behavior may fail to recognize the minor variants of previously observed malware. Hence, a generalization is necessary.

We have two basic steps to generalize atomic behaviors: (1) extracting security-sensitive OS object types and operation types based on careful analysis on system details; (2) making meaningful combinations of the OS object types and operation types to form candidate atomic behaviors, which are shown in the bottom of the next page.

B_{file}, $B_{registry}$, $B_{process}$, B_{IPC} and B_{system} represent five sets of atomic behaviors. The operation types of various objects are a generalization of one or several system calls and necessary arguments. The *FileType* is recognized from the extension names of the files. The *ParentDirectoryType* is recognized by the environment variables or paths, which represents the parent directory of the file. *RegistryType* is identified by the paths of the registry keys. *ProcessType* is detected by the names and paths of the image files. As a result of the generalization, we obtained 63 candidate atomic behaviors. Some examples include "create executable files under system directory", "modify registry to disable firewall" and "kill antivirus processes".

Note that a candidate atomic behavior might not have the detection capability as that based on behaviors extracted from dependency graphs [1][30], and some of them might even produce high false positive rate. However, when using these atomic behaviors to construct behavior templates, our

algorithm based on combinatorial optimization will automatically exclude the atomic behaviors with a high false positive rate from the final set of templates.

Actually, our experiments in Section IV demonstrate that after the optimization process the template database only contains 57 of the 63 atomic behavior candidates while excluding the behaviors that have high false positive rates. We observe that an atomic behavior eventually incorporated into the template database generates false positive rate of 56% at most, and the average number of behaviors in each template is about 7.2. Thus, when using a template with 7.2 independent atomic behaviors to determine a malware, the expected false positive rate should be lower than 1.5%, though each of the atomic behavior has significant false positive rate. Furthermore, implicitly considering the source of the processes that require the atomic behaviors can further reduce the false positive rate. This is because the source of a process in a cluster is either the network or a removable drive and thus the process has higher possibility to be malicious than that of not in a cluster.

2) Constructing Atomic Behavior Template Database: The template database is critical to the effectiveness and efficiency of the detection engine. We use a set of malware and a set of benign software to train the template database. The issue is how to build a template database that can recognize the maximum possible number of malicious programs while minimizing the number of benign programs being falsely identified as malware. Meanwhile, considering the system overhead, a template and a template database should include the minimum number of atomic behaviors respectively. We address this problem using combinational optimization algorithm to select an optimal template database that best satisfies the requirements above. As our algorithm is similar to [1], we do not elaborate it here.

3) Detection Algorithm: As an online detector, the detection algorithm should be quick and light-weight, which is critical to the applicability of the detector. For Growing Grapes, the detection algorithm searches in the behavior template database to determine whether there is a template that matches the set of behaviors exhibited by the given cluster. A natural implementation of the algorithm might use a number to represent a behavior and a set of numbers to constitute a template, and store a set of templates into a template database. In a real application scenario, the template database might be huge and thus cost a significant amount of time to search within the whole database. As the detection algorithm will be called very frequently by related system calls and API functions, such implementation of the algorithm will significantly affect the system performance.

To accelerate the template searching and matching procedure, we design a novel detection algorithm that uses a number to represent a behavior template rather than a behavior. Each bit of the number represents an atomic behavior belonging to the template. Thus an integer with 64 bits can express a template that consists of 64 atomic behaviors at most. Accordingly, the template matching procedure only needs a single comparison between two integers instead of a serial of such comparisons.

If the template database is very large, it is time consuming to search through it. Hence, we do not store all of the templates one by one as traditional methods do. Instead, we propose a novel structure for the template database, which uses a fixed size of 21K memory to contain up to 2^{64} templates and tests a few bits within the 21K space to fulfill a query for a given template.

Our template database is organized based on a *template graph*, which is shown in Figure 3(a). The graph consists of 256 nodes and necessary edges between nodes. The 256 nodes are organized into 16 rows and 16 columns. For a given template with 64 bits, each row stores the hex value of four consecutive bits of the template, and each node in a row represents a candidate hex value. We use V(R) to represent a node, where V is the hex value of the node and R is the row number. An edge can only connect two nodes that locate at two neighboring rows respectively. Thus, a template is represented by 16 nodes from 16 rows respectively and 15 edges that connect them. For example, the template database in Figure 3(a) contains two templates, i.e., hx0102400000000021 and hx2121000000000001. The former template corresponds to the path 0(0)→1(1)→0(2)→2(3)→4(4)→0(5)→0(6)→0(7)→0(8)→0(9)→0(A)→0(B)→0(C)→0(D)→2(E)→1(F). The node 0(0) indicates the hex value of the first four consecutive bits is 0, the node 1(1) indicates the hex value of the second four consecutive bits is 1, and so on. However, as the two templates cross with each other at some nodes, e.g., the node 1(1), from the figure, one may also find some wrong templates that actually do not exist, e.g., hx0121000000000001 and hx2102400000000001.

To address this issue, we build a *crossing list*, for every node to record the templates crossing the node. Each element of the crossing list corresponds to a template that crosses the node. An element is a pair that consists of the values of the preceding

$$B_{file} = FileOperationType \times FileType \times ParentDirectoryType = \{Create, Delete, Modify, Read, Write, Load\} \times \{Executable, Configuration\} \times \{System, Windir, \mathrm{Pr}ogramFiles, Temp, User\,\mathrm{Pr}ofile, IE, DriveRoot, Startup, RemovableDrive\}$$

$$B_{registry} = RegistryOperationType \times RegistryType = \{Create, Delete, Modify\} \times \{Startup, Explorer, IE, Driver, Service, Firewall, AntiVirus, Update, Restore\}$$

$$B_{process} = \mathrm{Pr}ocessOperationType \times \mathrm{Pr}ocessType = \{Start, Kill, Inject, Hide\} \times \{System, Explorer, Cmd, IE, Log, Antivirus\} B_{IPC} = IPCOperationType \times IPCType = \{Create\} \times \{Mutex, Event\}$$

$$B_{system} = SystemOperationType = \{Hook, Restart, LogKeyStrokes, ChangeDate\}$$

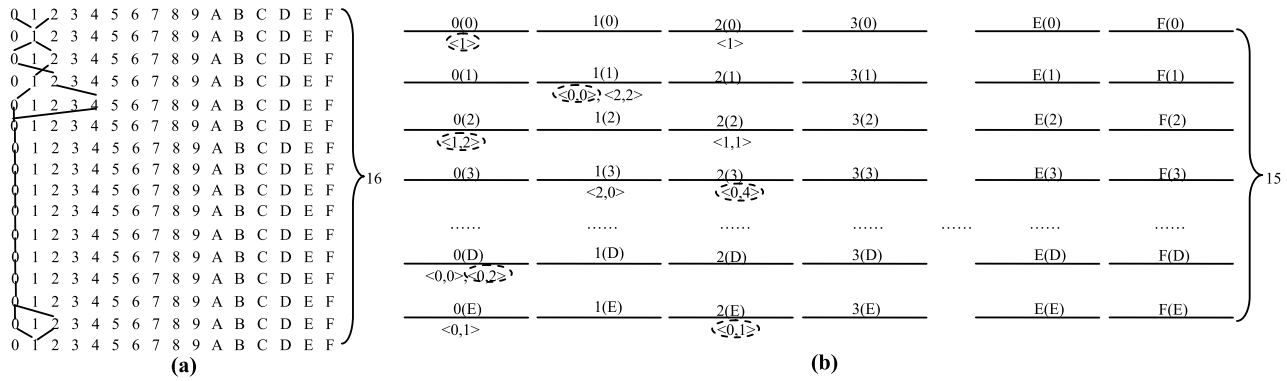

Fig. 3. Template graph for a behavior template database. (a) Template Graph. (b) Template Graph with Crossing Lists.

and succeeding nodes of the corresponding template. For example, in Figure 3(a) the node 1(1) should have a list with two elements: <0, 0> and <2, 2>. The element <0, 0> belongs to the template hx0102400000000021, where the two 0 indicate the values of the preceding node 0(0) and the succeeding node 0(2) respectively. The element <2, 2> belongs to the template hx2121000000000001, where the two 2 indicate the values of the preceding node 2(0) and succeeding node 2(2) respectively. As the crossing list of node 1(1) does not have two elements <0, 2> and <2, 0>, the wrong templates (hx0121000000000001 and hx2102400000000001) are thus excluded.

For a node at row 0, each element of its crossing list is only the value of a succeeding node since it does not have a preceding node. For a node at the last row F, it does not need a crossing list since it does not have the templates crossing problem. Thus the template graph can be transformed as in Figure 3(b). Since the last row F does not have crossing lists, to find a template, we only need to check the existence of 15 pairs in the crossing lists of corresponding 15 nodes from row 0 to E one by one. For example, Figure 3(b) shows the 15 pairs of the template hx0102400000000021 which are circled by dashed lines. The first pair is <1> of node 0(0), the second pair is <0, 0> of node 1(1), and so on.

To reduce the space for storing the graph and the time for matching a template in the graph, we store a node and its crossing list as a bit map which is 256 consecutive bits. The 256 bits represents all possible combination of the preceding and succeeding nodes since each row has 16 nodes. Each bit indicates whether the corresponding pair exists in the list. For example, the first bit indicates that if the pair <0, 0> exists and the last bit indicates if the pair <F, F> exists. A node at row 0 only needs 16 bits since an element in its crossing list only contains the succeeding node. A node at row F does not need any bits since it has not a crossing list. Accordingly, the template graph (i.e., the template database) only occupies 7200 bytes in total.

When a process in a cluster requests an atomic behavior, we combine the current behavior and the former atomic behaviors of the cluster together to form a behavior vector that has the same format as a behavior template. Then we use the behavior vector to query the template database. The query only needs one operation to simultaneously test 15 bits, which indicate the existence of 15 pairs in the corresponding crossing lists of 15 nodes from row 0 to E one by one. When all of the 15 bits are set, we can determine the existence of a corresponding template. Hence, the detection algorithm can be formally presented as shown at the bottom of the page.

Bitmap(i,j) represents the bitmap of node j(i). The operation [ ] after a bitmap tests whether the corresponding bit in the bitmap is set. The address of the bit in the bitmap can be computed by the expressions within the [ ]. Val(j) in the behavior vector indicates the value of the corresponding four consecutive bits. The operation to query whether there is a behavior vector hx0102400000000021 is as shown at the bottom of the next page.

The template database could lead to some wrong templates when two templates cross with each other at more than one consecutive nodes. Consider two templates hx0120000000000000 and hxF12FFFFFFFFFFFF. In the template graph, the wrong template hxF120000000000000 would also seem valid, because F12 and 120 appear in the crossing lists of nodes 1(1) and 2(2) respectively. The reason for this issue is that, at least 8 consecutive behaviors that constitute two consecutive nodes in at least two templates have the same value. To address this issue, when constructing the database, the order of the 64 behaviors should be carefully arranged so that the 8 consecutive behaviors are scattered into different non-neighbored nodes.

$$TemplateDatabase = \{Bitmap(i, j)\}, \quad i = 0, 1, ...E, j = 0, 1, ...F$$

$$BehaviorVector = \{Val(j)\}, \quad j = 0, 1, ...F$$

$$Query(BehaviorVector) = Bitmap(0, BehaviorVector.Val(0))[BehaviorVector.Val(1)] \cap \Big(\bigcap_{i=1}^{E} Bitmap(i, BehaviorVector.Val(i))[16 \times BehaviorVector.Val(i-1) + BehaviorVector.Val(i+1)] \Big)$$

Specifically, we use triple template databases T, T' and T". T is the original database while T' and T" are derived from T. Based on T, we exchange the positions of even behaviors to obtain T' and positions of odd behaviors to obtain T". A pair of exchanging rules can be as follows:

$$ExchangeEvenBehaviors: \ i \leftrightarrow (i+k)\%64, \quad i = 0, 2, 4, ...62, k > 8$$

$$ExchangeOddBehaviors: \ j \leftrightarrow (j+k)\%64, \quad j = 1, 3, 5, ...63, k > 8$$

A valid template should appear in all databases, because all the behavior values in the three databases remain unchanged while the behavior orders are changed. If appearing in only one or two databases, we can determine that the template is wrong in the databases. Hence, the detection algorithm is improved as follows:

$$Query(BehaviorVector) = Query(BehaviorVector, T) \cap Query(ExchangeEvenBehaviors(BehaviorVector), T') \cap Query(ExchangeOddBehaviors(BehaviorVector), T")$$

Query(BehaviorVector,T) tests the behavior vector in database T. *ExchangeEvenBehaviors(BehaviorVector) and ExchangeOddBehaviors(BehaviorVector)* reorder the behaviors in the vector according to the exchanging rules.

Based on the analysis previously, the three databases occupy only 21K bytes and the detection algorithm merely checks 45 bits in the three databases. Moreover, the speed and space taken by the algorithm are fixed even when the number of templates dramatically increases up to millions.

III. Implementation

To evaluate the effectiveness of the Growing Grapes malware detection approach, we have developed a prototype implementation for Windows XP, and carried out a series of experiments. Although XP is not as new as Vista, it is enough for verifying the Growing Grapes model since both versions of OS have very similar system calls and Win32 API functions based on which Growing Grapes works. Moreover, XP is still a popular OS platform on the Internet. As a result, it is not surprising that recent similar research projects [30][2][25][22] also perform on XP.

For the clustering engine that correlates suspicious objects into clusters, we intercept Windows system calls at the kernel level and Win32 API functions at the user level to attach a proper cluster label to each suspected object according to the root, spreading and clustering rules. To prevent intended bypassing, we always intercept a function at the kernel level rather than the application level if possible. For the permanent objects, the labels of files are stored in a specially created stream of each file. The labels of registry keys are recorded in a file under a specially protected directory. However, for the volatile objects, e.g., processes, their labels are temporarily stored in memory. Each cluster has a data structure to record all of its exhibited atomic behaviors, created or modified registry entries or files that can be used to launch the program after system booting, as well as whether the cluster is malicious.

For the detection engine, we monitor atomic behaviors and query the template database. All atomic behaviors are extracted by intercepting a single mission-critical system call/API function and analyzing the parameters. For example, monitoring NtSetValueKey() for "Change security settings". Some malware behaviors consist of more than one system call or Win32 function, for instance, the behavior "Inject into other processes" consists of NtOpenProcess(), NtAllocateVirtualMemory(), NtWriteVirtualMemory(), NtCreateThread(), etc. We only intercept the mission-critical function, i.e., NtCreateThread(). Every time an atomic behavior is intercepted, the detection engine generates a number that indicates what behaviors the cluster has exhibited by now and queries the template database in memory. If there is a match, the engine sends an alarm.

IV. Evaluations

A. Malware Detection

To demonstrate that our Growing Grapes system is effective in detecting malwares, we first collected 436 real-world malware samples mainly from a publicly available website [19]. The samples cover 127 malware families which are much more than used in the previous research [30][1], because we emphasize on detecting unknown malware from different families rather than merely different variants of the same family. This is closer to the real usage scenario that an end host works in the Internet for years and experiences various families of malware. Moreover, we also prepared 164 benign samples mostly from two trustworthy websites, i.e., technet.microsoft.com and www.download.com.

We set up a local network consisting of two servers and two hosts as a testing environment. Server A mainly stores malware samples, and runs IIS web server, ftp server and EZ-IRC server. Server B mainly stores benign samples, and runs IIS web server as a website. The host machines installed with Windows XP run the client programs that are often the attacking vectors for malwares, including mIRC, MSN Messenger, MS Outlook, eMule, IE, ftp client, etc. On one host,

$$\begin{aligned} Query(\text{hx0102400000000021}) = {} & Bitmap(0,0)[0+1] \cap Bitmap(1,1)[16\times 0+0] \\ & \cap Bitmap(2,0)[16\times 1+2] \cap Bitmap(3,2)[16\times 0+4] \cap Bitmap(4,4)[16\times 2+0] \\ & \cap Bitmap(5,0)[16\times 4+0] \cap Bitmap(6,0)[16\times 0+0] \cap Bitmap(7,0)[16\times 0+0] \\ & \cap Bitmap(8,0)[16\times 0+0] \cap Bitmap(9,0)[16\times 0+0] \cap Bitmap(A,0)[16\times 0+0] \\ & \cap Bitmap(B,0)[16\times 0+0] \cap Bitmap(C,0)[16\times 0+0] \cap Bitmap(D,0)[16\times 0+2] \\ & \cap Bitmap(E,2)[16\times 0+1] \end{aligned}$$

Firefox is installed to download samples from the server B. To emulate the real-world usage scenarios, we login into the hosts and perform tasks including browsing the malicious website and ftp server in the local network and downloading samples, sending and receiving adverse instant messages and emails, accessing P2P shared folders or removable drives that contain samples, etc. Thus, the samples are introduced into a host through various channels. With this testing environment, we intend to more thoroughly evaluate the capability of Growing Grapes to identify malware on an end host.

Before the evaluation, we validated the necessity of the clustering technique since it is critical to Growing Grapes. We modify the prototype so that multiple processes of a malware are not correlated together into a cluster while behavior templates are still applied to identify malware. The testing result shows that the modified prototype only identifies 105 of the 436 malware samples. Hence clustering is necessary for effective detection.

To evaluate the detection effectiveness on multiple datasets, we divide the samples into five disjoint sets, training the behavior template database on every set independently, and evaluating on the rest sets not used in each cycle of training. Moreover, we execute each cycle for nine different threshold values to explore the tradeoff between true positive rate, false positive rate, average number of behaviors in each template and number of templates in a template database. A threshold value is used by the behavior template database optimization algorithm to construct templates.

The evaluation results are shown in Figure 4. Considering true positive rate, our detector can successfully recognize 71.1% of unknown malware without false positives. Previous research [30] based on dependency graph reported the rate of 64% and commercial antivirus software reported the rate of 55% [16]. Hence, this is a significant improvement considering that we use much more malware families to perform evaluation than that of previous studies. Project HOLMES [1] obtained a better detection rate, but the result was not generated by a detector.

For the false positive rate, our detector produces at most 22.5% which is much less than that of HOLMES, i.e., 57.14%. The reason is that, when determining a malware, we utilize not only behaviors but also the sources of the processes launching the behaviors, which significantly improves the detection capability of each of the behaviors. Moreover, when optimizing the behavior template database, our objective function still tends to choose a low false positive rate when the threshold value exceeds the actual true positive rate, rather than simply return a large value as HOLMES does.

However, paper [30] reports no false positives as a result of detection based on dependency graph, while our detector raises false positives. The main reason is that our datasets for testing include a wide range of malware families, especially the families that exhibit a very few number of atomic behaviors. To detect such malwares, the template database needs to incorporate the templates that consist of only one to two atomic behaviors. Although these templates can detect the malwares with rare behaviors but at the same time tend to wrongly classify benign software as malicious.

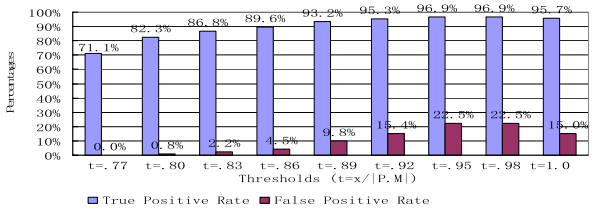

Fig. 4. True and false positive rates of Growing Grapes under different threshold number of true positives.

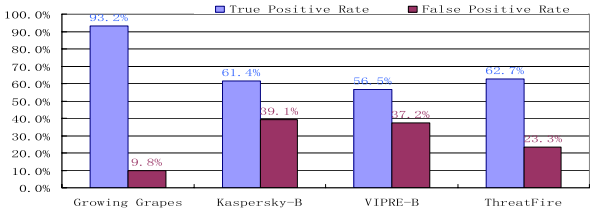

Fig. 5. Growing Grapes produces higher true positive rate and lower false positive rate than that of commercial anti-virus software.

B. Comparison With Commercial Tools

To further evaluate Growing Grapes, we performed another experiment using three popular commercial behavior-based anti-malware tools: Kaspersky-B, VIPRE-B and ThreatFire, where "B" indicates the behavior blocking module of the related anti-malware software. We test all of the malicious and benign samples in our evaluation set. Figure 5 shows the true positive rates and false positive rates of every tool. From the figure, our Growing Grapes is shown to outperform other schemes, with the highest true false positives and the lowest false positives.

As the internal technologies of the commercial tools are proprietary, we speculate that the better performance of our approach is due to our optimization on the behavior template database, which chooses a set of templates that cover the most malicious samples and the least benign samples. Moreover, the false positive rate is further reduced by implicitly taking into account the source of the process requiring the current behavior.

C. Overhead

In the following experiments, we evaluate the additional overhead imposed by Growing Grapes from three perspectives. The test-bed used in this evaluation consists of two machines. Machine A contains a Pentium-4 2.8GHz CPU with 1GB memory and runs applications including WinZip32, xCopy, BCC32 and WebBench, etc. Machine B contains an Intel Core 2 Duo 2GHz CPU with 2GB memory, and runs IIS web server and Telnet server. We installed Windows XP on both machines.

(1) Overhead of intercepted system calls and Win32 API functions. We first disable Growing Grapes, run a group of benign programs and malwares, and count the average CPU cycles spent in each system call and API function through rtdsc instruction. Then we enable Growing Grapes, run the malwares, the benign programs with cluster labels and without cluster labels to perform the test again. In all tests, the average CPU cycles of every system call or API function is calculated from 100 invokes.

TABLE I

OVERHEAD OF GROWING GRAPES (CPU CYCLES). THE COLUMNS GG-m, GG-bl AND GG-b SHOW THE CPU CYCLES TAKEN BY THE MALWARE PROGRAMS, THE BENIGN PROGRAMS WITH AND WITHOUT CLUSTER LABELS RUNNING ON GROWING GRAPES, RESPECTIVELY

Functions	Native	GG-m	GG-bl	GG-b	Functions	Native	GG-m	GG-bl	GG-b
NtCreateFile	334492	351886(5.2%)	351277(5.0%)	338515(1.2%)	CreateService	6568120	6695944(1.9%)	6692937(1.9%)	6568412(<0.1%)
NtOpenFile	167620	174667(4.2%)	174482(4.1%)	169728(1.3%)	OpenService	5490443	5624175(2.4%)	5621223(2.4%)	5490669(<0.1%)
NtWriteFile	245179	256332(4.5%)	255722(4.3%)	249820(1.9%)	NtSetValueKey	210491	227593(8.1%)	227219(7.9%)	210553(<0.1%)
NtCreateNamedPipeFile	204711	216817(5.9%)	216674(5.8%)	204790(<0.1%)	NtCreateKey	281722	299397(6.3%)	298917(6.1%)	281894(<0.1%)
NtCreatePort	37241	40645(9.1%)	40545(8.9%)	37278(<0.1%)	NtCreateProcessE	206458	217432(5.3%)	217184(5.2%)	208850(1.2%)

The testing results are shown in Table 1. With Growing Grapes enabled, the malware programs have 1.9%∼9.1% more performance penalty than native, while the benign programs have only 0∼8.9% overhead. In particular, the overhead incurred on the benign programs without a cluster label is lower than 1.9%. If a process does not have a cluster label, Growing Grapes will not perform any operations on its system calls. Since only the processes derived from the network or removable drives have a cluster label, most of the processes are almost not affected by Growing Grapes. The real malicious programs have the highest performance penalty as they always exhibit atomic behaviors that require analyzing the behaviors and matching behavior template. Therefore, Growing Grapes has an interesting performance penalty model which imposes the most penalties on real malicious programs, the next most penalties on the suspected programs and the least penalties on the benign programs. Generally, the performance impact from the system call and Win32 API function interception is small.

(2) Overhead of independent applications. We measured the execution time of a set of benign programs. These programs with certain parameters have different operational characteristics including file-bound, registry-bound, network-bound, process-bound and memory bound. The execution time of a program is the average duration from the start of a program to its termination. The results are shown in Figure 6. Growing Grapes imposes 1.28% overhead on average over the benign programs without cluster labels and 7.88% overhead on average over the benign programs with cluster labels compared to that of native. Some programs with certain parameters and cluster labels impose up to 10.37% overhead, which include "Reg import", "Rar e", "winmsd", "xcopy" and "rmdir". According to our analysis, these programs frequently perform behaviors similar to malicious behaviors, for example, modifying registry, creating executable files, deleting executable files and reading system information. In order to differentiate these behaviors from real malicious behaviors, Growing Grapes needs to spend longer time in monitoring and analyzing.

Furthermore, we measured the throughput of IIS web server on native Windows XP and the Growing Grapes prototype, respectively. The performance is evaluated using WebBench [35], a licensed PC Magazine benchmark program. Each reported measurement is an average of results from ten runs. In every testing session, each web server had one to twenty clients concurrently sending requests to it. The results are depicted in Figure 7. The performance loss of the web server when running on Growing Grapes is 7.5% on average compared with that of running on the native OS.

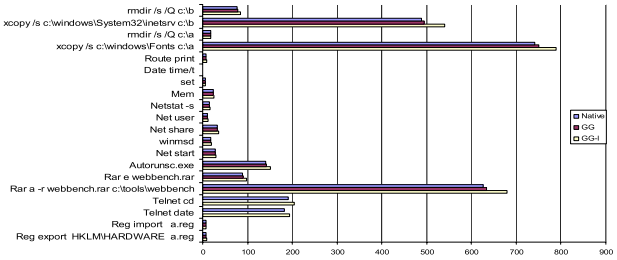

Fig. 6. The execution time of independent applications running on native Windows XP and Growing Grapes. GG and GG-l represent the applications running on Growing Grapes without and with labels.

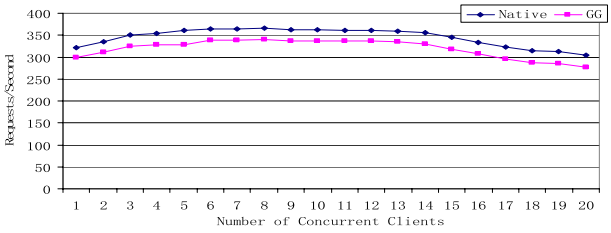

Fig. 7. IIS throughput on Growing Grapes and Native Windows XP. Growing Grapes causes 7.5% performance loss on average.

(3) Overhead over time. For this test, we used two tools to access IIS: WebBench and Uploader. Uploader is developed by us to automatically upload executable files onto IIS web server. We have continuously run WebBench, Uploader and IIS for 24 hours in the environment as presented above. Figure 8 shows the memory footprint of Growing Grapes mechanism when using WebBench or Uploader to access IIS, which are recorded as GG-WebBench and GG-Uploader respectively. The memory occupation of GG-WebBench and GG-Uploader increase 0% and 2270% respectively during the 24 hours testing. After careful analysis, we found that the memory increase is caused by accumulated cluster data structures. Each cluster in GG-WebBench is removed after its processes exit, but most clusters in GG-Uploader can not be deleted even after all its processes exit. These clusters contain executable files, thus Growing Grapes keep them in memory to find potential ASEP pairs according to Section 2.2.3.

Figure 9 shows the throughput of IIS when accessed by WebBench or Uploader. The throughput of IIS on GG-WebBench and GG-Uploader decrease 9.7% and 57% respectively during the 24 hours testing. Growing Grapes periodically scans existing clusters to find split ASEP pairs and combine the related clusters together. When running on GG-Uploader, many clusters are accumulated in memory and thus need significant time to scan them. This experiment

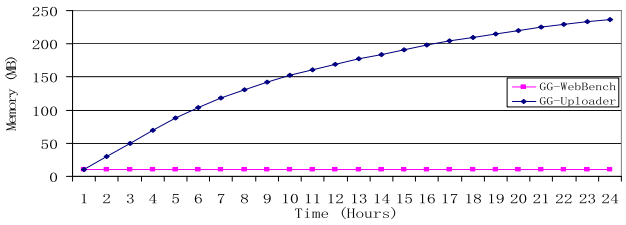

Fig. 8. The memory footprint of Growing Grapes. GG-WebBench and GG-Uploader represent that the IIS web server is accessed by WebBench and Uploader respectively.

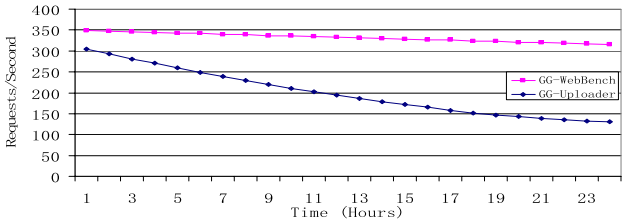

Fig. 9. The IIS throughput on Growing Grapes. GG-WebBench and GG-Uploader represent that the IIS web server is accessed by WebBench and Uploader respectively.

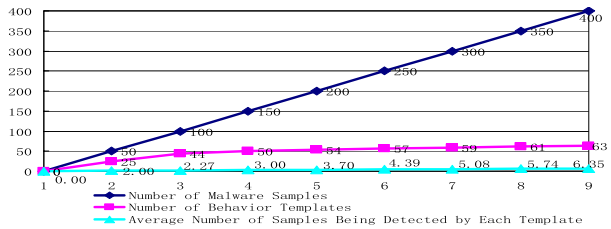

Fig. 10. The total number of behavior templates and the average number of malware samples being detected by each template increase slowly while the number of malware samples increases linearly.

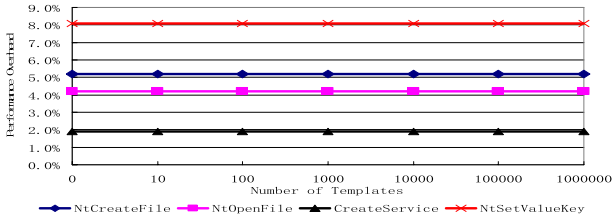

Fig. 11. The performance overhead of Growing Grapes on several important system calls and API calls keeps unchanged while the number of templates increases from 0 to 1,000,000.

indicates that, with Growing Grapes running, the performance of certain applications will degrade significantly after a long period of time, which is a limitation of Growing Grapes. We will consider the improvement method in the future.

D. Scalability

To evaluate the scalability of Growing Grapes, we first calculate the number of templates required by the template database to detect a certain number of malware samples with true positive rate at about 90%, using our algorithm for building template database. The results are shown in Figure 10. The number of templates does not scale linearly with the number of samples in the training dataset. The increase speed of templates significantly reduces as the number of samples increases linearly. One major reason is that the average number of behaviors being detected by each template increases when the number of samples increases, which in turn counteracts the increase speed of templates.

We then test the performance overhead when the number of templates increases. The results are illustrated in Figure 11. The performance overhead is represented by several system calls and API calls that are invoked by malicious processes. The performance overhead of Growing Grapes almost does not change when the number of templates increases from 0 to 1,000,000. As presented in Section 2.3, with our intelligent design of the template database, the time for searching a template in the template database is fixed and not affected by the number of templates in the database.

V. Related Work

Growing Grapes is a type of behavior-based detector which monitors system calls or API calls, but it differs from all existing behavior-based studies. Previous work takes into account the relations among system calls, for example, "system call sequence" [3][14][34], "API sequence" [11], "system call N-Gram" [12][13], "system call Finite-State-Automata" [23][9], "system call stack" [26][21], "system call frequency" [15] and "system call dependency graph" [1][22][29][30][25][33]. However, Growing Grapes ignores relations among system calls and only utilizes individual mission-critical system calls and their arguments, which can succinctly capture the intent of the malware author.

HOLMES [1] also employs multiple behaviors to recognize a malware and develops an optimization algorithm to combine candidate behaviors into discriminative specifications. However, HOLMES is not an online system. Moreover, it uses dependency graphs to represent behaviors and thus has the problems faced by other detectors using dependency graphs. A recent effort [30] detects malware based on dependency graphs but does not trace data flow, and thus addresses the low performance issue. However, it can only detect previously seen data dependencies. So, its capability to identify previously unseen malware is limited. Furthermore, as it relies on a graph matching algorithm to determine a malware, the performance impact is not clear when deployed in a real application scenario that requires enormous number of behavior graphs. The authors in [20] propose a behavior-based approach to detect malware. The basic difference lies in that it aims to clean up malware impacts within a virtual machine when committing the content of the virtual machine into the host environment. We propose a much more comprehensive scheme for detecting malware on an end host, and also a novel method of generating and storing behavior templates, which not only ensures a good detection performance but also makes the detection very simple and scalable.

Research efforts in [10][17][18] demonstrate the approaches of tracing OS-level information flow with the support of virtual machines or backend hosts. Growing Grapes use these approaches to correlate malware objects. In order to support online detection on a host, Growing Grapes significantly improves the tracing performance by pruning the information flow that has a low risk of propagating malware.

VI. Summary

In this paper, we propose Growing Grapes, a novel scheme towards building a behavior-based accurate online detector that

requires low false positive and high performance simultaneously. Growing Grapes has two engines. One is the clustering engine responsible for collecting the objects of a suspicious program into a cluster by tracing light-weight OS-level information flow rather than traditional data flow. The other is the detection engine that determines a malware using multiple simple behaviors, i.e., a behavior template, rather than a single complex behavior. The detector thus novelly identifies the malicious changes of a malware in a cluster fashion rather than one by one, which helps the ordinary users to thoroughly clean up the malware. The template database is optimized to reduce the false positive rate while preserving high true positive rate. With a novel design of the template and database structure, it can complete a query in one operation and occupies merely 21K memory space which allows storing up to 2^{64} templates. Extensive experiments show that Growing Grapes can detect 71.1% of unknown malware samples without false positives while the performance overhead is less than 9.1% and 1.9% on malware programs and most benign programs respectively.

REFERENCES

[1] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proc. IEEE Symp. Sec. Privacy*, Berkeley, CA, USA, Apr. 2010, pp. 45–60.

[2] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using system-centric models for malware protection," in *Proc. 17th ACM CCS*, Chicago, IL, USA, Oct. 2010, pp. 399–412.

[3] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. IEEE Symp. Sec. Privacy*, Oakland, CA, USA, May 1996, pp. 120–128.

[4] P. Szor, *The Art of Computer Virus Research and Defense*. Reading, MA, USA: Addison-Wesley, 2005.

[5] M. Howard. (2003). *Fending Off Future Attacks by Reducing Attack Surface* [Online]. Available: http://msdn.microsoft.com/en-us/library/ms972812.aspx

[6] (2013, Dec. 10). *Microsoft Security Bulletins* [Online]. Available: http://www.microsoft.com/technet/security/current.aspx

[7] Symantec, Inc., Mountain View, CA, USA. (2012, Aug.). *Threats List* [Online]. Available: http://www.symantec.com/business/security_response/threatexplorer/threats.jsp

[8] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, *et al.*, "Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management," in *Proc. 18th LISA Syst. Admin. Conf.*, vol. 4. Nov. 2004, pp. 33–46.

[9] C. C. Michael and A. Ghosh, "Simple, state based approaches to program-based anomaly detection," *ACM Trans. Inf. Syst. Sec.*, vol. 5, no. 3, pp. 203–237, Aug. 2002.

[10] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proc. ACM Symp. Oper. Syst. Principles*, 2003, pp. 223–236.

[11] J. Xu, A. H. Sung, P. Chavez, and S. Mukkamala, "Polymorphic malicious executable scanner by API sequence analysis," in *Proc. 4th Int. Conf. HIS*, Dec. 2004, pp. 378–383.

[12] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Sec.*, vol. 6, no. 3, pp. 151–180, Jan. 1998.

[13] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Proc. 7th USENIX Sec. Symp.*, vol. 7. 1998, pp. 1–6.

[14] S. Mukkamala, A. Sung, D. Xu, and P. Chavez, "Static analyzer for vicious executables (SAVE)," in *Proc. 20th ACSAC*, 2004, pp. 326–334.

[15] D. Kang, D. Fuller, and V. Honavar, "Learning classifiers for isuse and anomaly detection using a bag of system calls representation," in *Proc. 6th IEEE Syst. Man Cybern. IAW*, Jan. 2005, pp. 1–8.

[16] E. Larkin, *Top Internet Security Suites: Paying for Protection*. New York, NY, USA: PC Mag., Jan. 2009.

[17] K. Farhadi, Z. Li, A. Goel, K. Po, and E. Lara, "The taser intrusion recovery system," in *Proc. 20th ACM SOSP*, Dec. 2005, pp. 163–176.

[18] N. Zhu and T. Chiueh. "Design, implementation, and evaluation of repairable file service," in *Proc. 21st ICDE*, 2003, pp. 1024–1035.

[19] (2013, Nov. 13). *Offensive Computing* [Online]. Available: http://www.offensivecomputing.net/

[20] Z. Shan, X. Wang, T. Chiueh, and X. Meng, "Safe side effects commitment for OS-level virtualization," in *Proc. 8th ACM Int. Conf. Auto. Comput.*, 2011, pp. 111–120.

[21] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection,"in *Proc. 11th ACM CCS*, 2004, pp. 318–329.

[22] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. 14th ACM Conf. CCS*, Oct. 2007, pp. 116–127.

[23] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati, "A fast automaton-based approach for detecting anomalous program behaviors," in *Proc. IEEE Symp. Sec. Privacy*, Jan. 2001, pp. 144–155.

[24] T. Holz, C. Willems, K. Rieck, P. Duessel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. 5th Conf. DIMVA*, Jun. 2008, pp. 108–125.

[25] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *Proc. 11th Int. Symp. Recent Adv. Intrusion Detection*, Cambridge, MA, USA, Sep. 2008, pp. 1–20.

[26] H. Feng, O. Kolesnikov, P. Folga, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. IEEE Symp. Sec. Privacy*, May 2003, pp. 62–75.

[27] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Proc. 6th Int. Conf. Detection Intrusions Malware, Vulnerabil. Assessment*, Como, Italy, Jul. 2009, pp. 88–106.

[28] O. Sukwong, H. Kim, and J. Hoe, "An empirical study of commercial antivirus software effectiveness," Ph.D. dissertation, Dept. Electr. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA, USA, Jun. 2010.

[29] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. 6th Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 5–14.

[30] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proc. USENIX Sec. Symp.*, 2009, pp. 351–366.

[31] N. Li, Z. Mao, and H. Chen, "Usable mandatory integrity protection for operating systems," in *Proc. IEEE Symp. Sec. Privacy*, 2007, pp. 1–15.

[32] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based spyware detection," in *Proc. USENIX Sec. Symp.*, San Diego, CA, USA, Feb. 2006, pp. 1–19.

[33] A. Bose, X. Hu, K. G. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *Proc. 6th Int. Conf. Mobile Syst., Appl., Services*, Breckenridge, CO, USA, Jun. 2008, pp. 225–238.

[34] I. Burguera, U. Zurutuza, and S. N. Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Sec. SPSMD*, 2011, pp. 15–26.

[35] PC Magazine, New York, NY, USA. (2013). *PC Magazine Benchmarks* [Online]. Available: http://www.pcmag.com/encyclopedia_term/0,2542,t=WebBench&i=48947,00.asp

Zhiyong Shan is an Associate Professor with the Department of Computer Science, Renmin University of China, and a Research Scholar with the Department of Computer Science, Purdue University. He received the Ph.D. degree in computer science from the Chinese Academy of Science. He received the President Award of the Chinese Academy of Science in 2004 and the Beijing Science and Technology Achievement Award in 2005. His research interests include operating system and computer security.

Xin Wang is an Associate Professor with the Department of Electrical and Computer Engineering and an Affiliated Professor with the Department of Computer Science, Stony Brook University. She received the Ph.D. degree in electrical and computer engineering from Columbia University. Her interests include wireless networks, mobile and distributed computing, and computer security. She received the NSF Career Award in 2001.