www.ijecs.in
*International Journal Of Engineering And Computer Science ISSN: 2319-7242*
*Volume 4 Issue 8 Aug 2015, Page No. 13921-13926*

# Study of Growing Grapes Technique for Malware Detection

*Mr. Bhushan Kinholkar*

PG Student, Department of Computer Science Engineering
SSBT's College of Engineering and Technology, Jalgaon, India.
bhushank0029@gmail.com

**Abstract:** ***The Behavior based detection is promising to solve the pressing security problem of malware. The great challenge lies in how to detect malware in a both accurate and light-weight manner. The 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. Malware based on multiple behaviors and the source of the processes requesting the behaviors. 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, 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.***

**Keyword:** Malware Detection, Growing Grapes, OS Level

## 1. Introduction

The 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. Behavior based detectors is difficult to be by passed by obfuscations or polymorphisms that are used to evade signature based detectors. 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. The grand challenge in behavior based malware detection is to perform detection accurately with a low performance overhead. Commercial antivirus tools often have a module for monitoring malicious behaviors, which is light-weight but not accurate. The module only leverages a single system call and the parameters to determine a malicious program. For example, intercepting and analyzing the arguments to determine whether a program is trying to modify a security sensitive registry key, and then given up an alarm window when this is true. As a outcome, 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 meaningful 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 meaningfully 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 cannot work effectively online, since they are bulky or inaccurate. By carefully analyzing existing technologies, 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 study of growing grapes the devise a novel malware detector, named Growing Grapes, which can achieve accurate malware detection without incurring high overhead. The 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 antivirus tools. Each atomic behavior consists of a system call and their arguments. All templates are stored in a behavior template database.

In order to have an accurate online detector, the define two techniques. First, to achieve accurate detection, using 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, the malware detector can identify the maximum possible number of malware samples while incurring the minimum false positive rate. The further reduce the false positive rate; implicitly take into account the source of the processes launching the behaviors when determining a malicious cluster. Second, take two means to achieve online detection: 1) The 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)The design a novel structure for the template database. The database occupies a small and fixed size of memory 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.

## 2. 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 This paper reports preliminary results aimed at establishing such a definition of self for Unix processes, one in which self is treated synonymously with normal behavior. That short sequences of system calls in running processes generate a stable signature for normal behavior. The signature has low variance over a wide range of normal operating conditions and is specific to each different kind of process, providing clear separation between different kinds of programs. Further, the signature has a high probability of being relaxed when abnormal activities, such as attacks or attack attempts, occur. These results are significant because most prior published work on intrusion detection has relied on either a much more complex definition of normal behavior or on prior knowledge about the specific form of intrusions. The suggest that a simpler approach, such as the one described in this paper; can be effective in providing partial protection from intrusions. One advantage of a simple definition for normal behavior is the potential for implementing an on line monitoring system that runs in real time [1].

In this paper, present a robust signature based malware detection technique, with emphasis on detecting obfuscated malware and mutated malware. The hypothesis is that all versions of the same malware share a common core signature that is a combination of several features of the code. After a particular malware has been first identified, it can be analyzed to extract the signature, which provides a basis for detecting variants and mutants of the same malware in the future. Encouraging experimental results on a large set of recent malware are presented [2].

In this paper capitalize on earlier approaches for dynamic analysis of application behavior as a means for detecting malware in the Android platform. The detector is embedded in an overall framework for collection of traces from an unlimited number of real users based on crowdsourcing. The framework has been demonstrated by analyzing the data collected in the central server using two types of data sets those from artificial malware created for test purposes, and those from real malware found in the wild. The method is shown to be an elective means of isolating the malware and alerting the users of a downloaded malware. This shows the potential for avoiding the spreading of a detected malware to a larger community [3].

There are many different levels on which IDS can monitor system behavior. It is critical to profile normal behavior at a level that is both robust to variations in normal and relaxations by intrusions. In the work reported here, chose to monitor behavior at the level of privileged processes. Privileged processes are running programs that perform services (such as sending or receiving mail), which require access to system resources that are inaccessible to the ordinary user. To enable these processes to perform their jobs, they are given privileges over and above those of an ordinary user (even though they can be invoked by ordinary users). In UNIX, processes usually run with the privileges of the user that invoked them. However, privileged processes can change their privileges to that of the super user by means of the set mechanism. One of the security problems with privileged processes in UNIX is that the granularity of permissions is too coarse Privileged processes need super user status to access system resources, but granting them such status gives them more permission than necessary to perform their specific tasks. Consequently, they have permission to access all system resources, not just those that are relevant to their operation. Privileged processes are trusted to access only relevant system resources, but in cases where there is some programming error in the code that the privileged process is running, or if the privileged process is incorrectly configured, an ordinary user may be able to gain super user privileges by exploiting the problem in the program. For the sake of brevity, usually refer to privileged processes or programs simply as and use the qualifier only to resolve ambiguities [4].

An automatic technique for building such specifications is desirable, both to reduce the AV vendors' response time to new threats and to guarantee precise behavioral specifications. If the behavioral specification used for detection is not specific enough, then there is a risk that benign applications will be flagged as malware. Similarly, if it is too specific then it may fail to detect minor variants of previously observed malware. In this paper are define address the challenge of automatically creating behavioral specifications that strike a suitable balance in this regard, thus removing the dependence on human expertise. The making the observation that the behavioral specifications used in malware detection are a form of discriminative specification. A discriminative specification describes the unique properties for a set of programs, in contrast to another set of programs. This paper gives an automatic technique that combines graph mining and concept analysis to synthesize discriminative specifications, and explores an application of the resulting specifications to malware detection. Given a set of behavior graphs that describe the semantics exhibited by malicious and benign applications, the graph mining operation extracts significant behaviors that can be used to distinguish the malware from benign applications. As these behaviors are not necessarily shared by all programs in the same set (as, for example, there are many ways in which a malicious program can attack a system), the use them as building blocks for constructing discriminative specifications that are general across variants, and thus robust to many obfuscations. Furthermore, because our graph mining and specification construction algorithms are indifferent to the details of the underlying graph representation, technique complements and benefits from recent advances in binary analysis and behavior graph construction [5].

A common application of virtual machines (VM) is to use and then throw away, basically treating a VM like a completely isolated and disposable entity. The disadvantage of this approach is that if there is no malicious activity, the user has to re do all of the work in her actual workspace since there is no easy way to commit (i.e., merge) only the benign updates within the VM back to the host environment. In this work, develop a VM commitment system called Secom to automatically eliminate malicious state changes when merging the contents of an OS-level VM to the host. Secom consists of three steps grouping state changes into clusters, distinguishing between benign and malicious clusters, and committing benign clusters. Secom has three novel features. First, instead of relying on a huge volume of log data, it leverages OS-level information flow and malware behavior information to recognize malicious changes. As a result, the approach imposes a smaller performance overhead. Second, different from existing intrusion detection and recovery systems that detect compromised OS objects one by one, Secom classifies objects into clusters and then identifies malicious objects on a cluster by cluster basis. Third, to reduce the false positive rate when identifying malicious clusters, it simultaneously considers two malware behaviors that are of different types and the origin of the processes that exhibit these behaviors, rather than considers a single behavior alone as done by existing malware detection methods. The having successfully define Secom on the Feather-weight Virtual Machine (FVM) system, a Windows based OS-level virtualization system. Experiments show that the prototype can effectively eliminate malicious state changes while committing a VM with small performance degradation. Moreover, compared with the commercial anti malware tools, the Secom prototype has a smaller number of false negatives and thus can more thoroughly clean up malware side effects. In addition, the number of false positives of the Secom prototype is also lower than that achieved by the on-line behavior-based approach of the commercial tools [6].

## 3. Growing Grapes Technique

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.

### 3.1 Cluster 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 the agent of an intruder and an executable file determines the execution flow of a process which represents an intruder. Based on the cluster, they 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 [7].

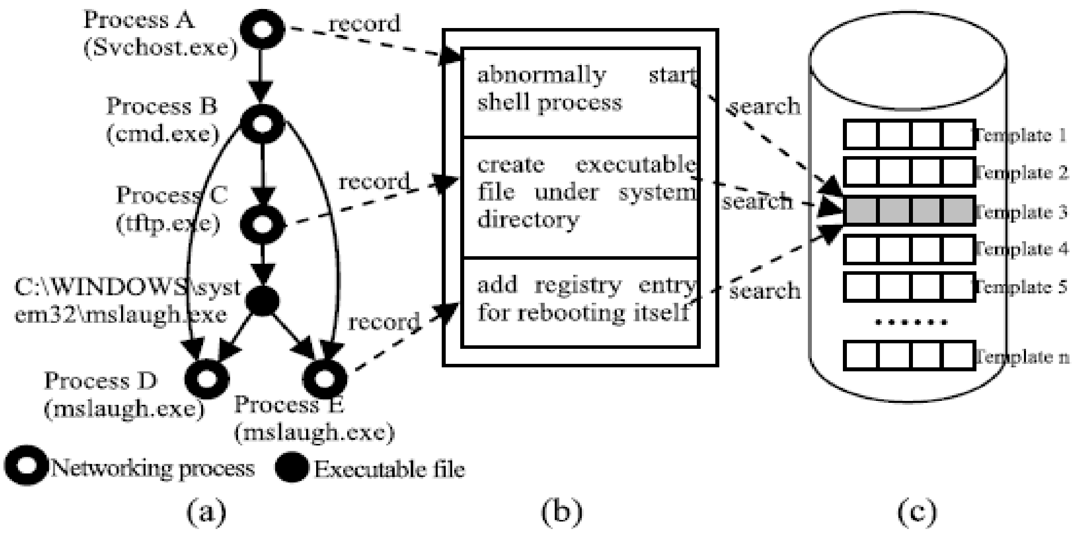

**Figure 1:** (a) Cluster, (b) Atomic Behavior, (c)Template

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. The 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, the 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.

### 3.2 Root Rule

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 executable (i.e., executable file) located at removable drives as suspicious.

An executable in this paper are define growing grapes technique 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.

### 3.3 Spreading Rule

To track OS-level information flow, Back Tracker 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 tradeoff between reducing the number of marked objects and reducing the risk of malware evasion. The 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, the following rules to mark related objects as suspicious.

- **Spreading Rule 1):** Marking executable files created or modified by a suspicious process as suspicious
- **Spreading Rule 2):** Marking processes spawned by a suspicious process as suspicious
- **Spreading Rule 3):** Marking processes loading a suspicious executable file or reading a suspicious semi executable or script file as suspicious
- **Spreading Rule 4):** 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 [7]. Thus, to track the information flows with a high risk of propagating malwares, the spreading rules focus on tracing executable 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.

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. 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 [7]. 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 permit a suspicious process to modify them. Therefore, although the operations on registry entries or configuration files are not traced, malwares still cannot avoid being detected by Growing Grapes. To reduce the number of marked processes, the spreading rules only trace dangerous IPCs Inter Process Communication. According to investigation on Microsoft Security Bulletins, a primary source for analyzing attack vectors of Windows OS, 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 cannot 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, the employ a Dangerous-IPC-List to trace dangerous IPCs since there are very few dangerous IPCs in a Windows OS.

### 3.4 Clustering Rule

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 [7]. 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, this technique 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.

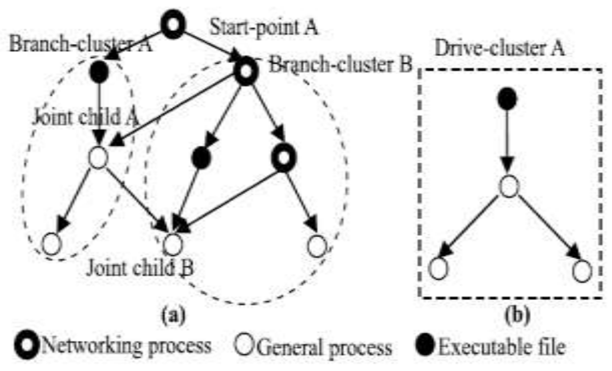

**Figure 2**: (a) Branch Cluster, (b) Drive Cluster.

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 to abrasive-grained to be used to recognize malicious objects in a cluster fashion since it might contain both benign and malicious objects [7]. In other words, this technique cannot determine that all objects in a graph are malicious even if most of the objects in the graph are malicious. Thus, 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 and analysis on a huge number of malware descriptions in the Symantec Threat Explorer, 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 backdoor process. After this, the attacker tries to download and install attacking tools, as well as performs any other adversary actions.

- **Clustering Rule 1:** Attaching a cluster label to a process and its descendants if the process is directly spawned by a network-facing process.

This type of cluster a branch cluster, e.g., the Branch cluster B in Figure 2. 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. After started, malwares copy themselves and make themselves resident in a host. Consequently, the following rule.

- **Clustering Rule 2:** Attaching a cluster label to a downloaded executable and its descendants.
- **Clustering Rule 3:** Attaching a cluster label to an executable file located on a removable drive and all its descendent objects.

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, determine the cluster label of the joint child. Basically, make decision according to the priority sequence like other objects. Obviously, the joint child should inherit the cluster label from its parent process or executable 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) pair into two different clusters. Then, the two clusters work together to perform malicious actions and potentially evade Growing Grapes detection [7]. 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, periodically scan the clusters to see whether there are split ASEP pairs, and combine the related clusters together if found.

## 3.5 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 [7]. 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 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 also supports our idea of using multiple behaviors. The result shows that a set of discriminative operations 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. The present strategies for addressing these three challenges in the following three subsections.

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.

The having 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 next paragraph.

B_file, B_registry, B_process, and B_system represent four sets of atomic behaviors. The operation types of various objects are a generalization of one or several system calls and necessary arguments. The File Type is recognized from the extension names of the files. The Parent Directory Type is recognized by the environment variables or paths, which represents the parent directory of the file. Registry Type is identified by the paths of the registry keys. Process Type is detected by the names and paths of the image files. As a result of the generalization, and obtained 63 candidate atomic behaviors. Some examples include "create executable files under system directory", "modify registry to disable firewall" and "kill antivirus processes".

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 is 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, do not store all of the templates one by one as traditional methods do. Instead, here is define a novel structure for the template database, which uses a fixed size of 21K memory to contain up to 264 templates and tests a few bits within the 21K space to fulfill a query for a given template.

## 4. Conclusion

In this paper, the overall study of 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.

## Reference

[1] 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.

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

[3] 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.

[4] 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

[5] M. Frederickson, 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.

[6] 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.

[7] Zhiyong Shan and Xin Wang," Growing Grapes in Your Computer to Defend against Malware ", Ieee transactions on information forensics and security, vol. 9, no. 2, February 2014.