Automatic Functionality Detection in Behavior-Based IDS

Tomas Nykodym, Victor Skormin, Andrey Dolgikh
Binghamton University
Binghamton, NY
tnykody1@binghamton.edu

Abstract—Detection of malicious functionalities presents an effective way to detect malware in behavior-based IDS. A technology including the utilization of Colored Petri Nets for the generalized description and consequent detection of specific malicious functionalities from system call data has been previously developed, verified and presented. A successful effort was made to neutralize possible attempts to obfuscate this approach. Nevertheless, the approach has two major drawbacks. First, target functionalities have to be initially specified by an expert, which is a time consuming, sometimes subjective and error prone process. Second, the identification of typical functionalities indicative of malicious programs is not generally straightforward and requires reverse engineering and careful study of many instances of malware. Our paper addresses these drawbacks, clearing the way for a full-scale practical application of this technology. We utilized graph mining and graph similarity assessment algorithms for processing system call data resulting in automatic extraction of functionalities from system call data. This enabled us to identify sets of functionalities suggesting software maliciousness and construct a general obfuscation-resilient malware detector. The paper presents the results of the implementation and testing of the described technologies on the computer network testbed.

Keywords—Behavior Based IDS; Signature generation; Colored Petri Nets

I. INTRODUCTION

Malware today presents a significant threat. In the past years, malicious codes became widespread and were commonly used to conduct information attacks [2], and as in the case of the Stuxnet worm [1], even industrial espionage and sabotage. Consequently, Intrusion Detection is a very active area of research that continues evolving as the malware techniques improve to overcome existing defenses. However, the most popular malware detection scheme remains binary signature-based detection. Although it has many practical advantages, this technology can be evaded by using automatic tools like code packers and metamorphic engines, and leads to a dead end due to exponentially growing database of binary signatures.

Behavioral detection offers a more promising approach to malware detection since behavior signatures are more obfuscation resilient than the binary ones. Indeed, changing behavior while preserving the desired (malicious) functions of a program is much harder than changing only the binary structure. More importantly, to achieve its goal, malware usually has to perform some system operations (e.g. registry manipulation). Since system operations can be easily observed and they are difficult to obfuscate or hide, malicious programs will expose themselves to behavioral detection. However, the behavioral detector has to be able to distinguish malicious operations from benign (executed by benign program) ones which is often difficult. Moreover, maliciousness of an executed functionality can often be determined only by its context or environment. Therefore the challenge of behavioral detection is in devising a good model of behavior which is descriptive enough to allow for discrimination of benign vs malicious programs and which can be tuned to the target environment.

In principle, there are two kinds of behavior detection mechanisms: misuse detection and anomaly detection. Misuse detection looks for specific behavioral patterns known to be malicious, while the anomaly based approach responds to unusual (unexpected) behavior. The advantage of anomaly based detection is in its ability to protect against previously unseen threats, however, it usually suffers from a high false positive rate. Misuse detection is usually more reliable in terms of better detection performance (fewer false positives and often no false negatives) but it has two major drawbacks. First, defining a set of malicious patterns (signatures) is a time consuming and error prone task that calls for periodic updating, similarly to how binary signatures are used today. Second, it cannot detect any malicious code that does not utilize the functionalities contained in the known malicious set and thus its capabilities to detect a zero day attack are limited. Constructing a large, comprehensive database of malicious functionalities would alleviate the second drawback placing formidable obstacles in the way of the attacker. Moreover, if the specifications of discovered malicious behaviors are general enough, they may well prevent certain types of attacks (e.g. code injection) to happen on the system. This will not only increase the cost of a successful attack, it can also limit the set of malicious actions and thus reduce the possible damage caused by an attack.

In this paper we present a software tool for analysis of runtime behavior of programs enabling automatic extraction of significant functionalities. These functionalities can then be converted into features and used as an input to some general classifier. Thus, well known classification techniques from machine learning can be applied to the problem of malware
detection. We base our approach on an assumption that functionality (especially as seen from the OS perspective) of malicious and benign programs differ significantly and therefore it should be possible to build a model distinguishing one from another.

We model program behavior by building a Kernel Object Access Graph capturing how kernel objects (objects managed by operating system, e.g. files, processes) are manipulated. Unlike most other approaches, we do not restrict behavior to process boundaries and thus our model can capture inter-process cooperation. Program execution usually contains lot of repetitive actions while the number of the repetitions usually does not affect semantic meaning. Such redundant information increases the amount of computation needed to process the graphs. More importantly, the number of repetitions can vary depending on the input data or even state of the operating system and thus make representations of the same activity look different. For these reasons, we compress the graphs obtained from the execution traces by removing repetitive subgraph sequences and replacing frequently occurring subgraphs by single nodes. By compressing the repetitive structures in the access model, we can efficiently represent even execution traces of long running programs and produce robust behavioral signatures. Behavior of a program (malicious or benign) can then be created by selecting significant subgraphs from the set of obtained malicious access graphs. Thus, the malicious signature is described by actions happening in the system rather than what particular program executes. We believe this approach is more obfuscation resilient and provides more information for detection (since it can contain execution context if needed). Our algorithm selects subgraphs present in traces from several malware instances (measure of generality) with maximal distance to subgraphs found in execution traces of benign programs to decrease the false positive rate. We collected runtime traces from several legitimate and malicious programs and evaluated our behavioral scheme in terms of performance and generality of the created signatures.

The contributions of our paper are as follows:

- We describe the use of kernel object access graph as a model for behavior description.
- We develop an efficient graph compression scheme abstracting from unimportant aspects of execution trace (e.g., number of repetitions) and significantly decreasing the size of the search space for later processing.
- We develop an algorithm extracting the behavioral signature from the execution trace.
- We evaluate our model on several legitimate and malicious programs and estimate its discriminative power.

II. Behavior Model

The program behavior can be defined in many ways depending on the level of abstraction. More detailed models can express behavior more precisely, however, they can be avoided easily. On the other hand, models which are too general may fail to discriminate between very different behaviors and thus allow malware to look like a benign program. For the purpose of malware analysis, observing behavior on the system call level is a natural choice since it captures information about how important objects in the operating system (e.g., files, processes) are manipulated. At the same time it abstracts from specific (user-mode) computation behind it and it is thus being more resilient to obfuscations. Several behavioral models built over system calls were studied in the past and their use for malware detection was evaluated. In [3], the authors use Colored Petri Nets to describe specific malicious functionalities. They describe functionality as a sequence of operations achieving significant results in the programs environment. It can be viewed as significant part of program behavior. In [4] the authors model behavior by constructing data dependency graphs of system calls, extracting several kinds of simple data dependencies and modeling the complex ones by using program slicing techniques. While this model is highly descriptive and precise, the use of data dependencies opens several possibilities for obfuscation since it relies on taint propagation, which is avoidable [7].

A. Tracking Kernel Object Access

We model the behavior by building a data dependency graph of system calls following only dependencies of kernel objects and handles. Kernel objects represent important objects in the operating system, such as files or processes. They are managed by the operating system kernel and so they can be accessed and manipulated only through system calls. Programs access kernel objects via handles, which are identifiers unique in the context of each process. Since all important data structures and resources in the operating system are represented by a kernel object, the malicious program cannot do much without manipulating at least some of them. Kernel object dependencies are difficult to hide and since we ignore any other data dependencies, this model provides a robust, obfuscation resilient tool for malware analysis.

![Figure 1: Example of access graph component](image)

B. Kernel Object Access Graph

Dependencies between kernel objects can be represented by a directed graph, where each vertex represents an access (by system call) of a particular kernel object and edges represent dependency on (or modification of) the object state. Every vertex is labeled by system call number and edges are labeled by the type of access. Since it is directed and acyclic, it defines partial ordering on the system calls observed in the trace. Kernel Objects are tracked by their handles if they are not named, or by their names if they are available.
The following operations are monitored:

- **Object Creation** - a system call that creates the object, if it is known
- **Obtaining a Handle to an Object** - a system call which obtained the handle, but did not create the object (e.g. opening a file)
- **Read access** - any non-modifying access of an object, e.g. read file or query directory
- **Write/modify access** - any modifying access

The kernel object dependency graph can be defined as follows:

\[ G = (V, E) \]

- \( V \) is the set of vertices. Every system call in a trace is represented by a vertex, labeled by its system call index and its sequence number.
- \( E \) is the set of edges. There is an edge \((u, v)\) if:
  - Vertex \( v \) references a kernel object created by a system call represented by vertex \( u \), or it uses a kernel object via a handle obtained from \( u \)
  - Vertex \( v \) references kernel object \( O \), which was modified by vertex \( u \)
  - Vertex \( u \) accessed kernel object \( O \) for reading and vertex \( v \) is the first system call after \( u \) which modifies \( O \)
  - Vertex \( u \) uses a handle closed by vertex \( v \)

An example of a dependency graph is shown in Figure 1. (All graphs are drawn without redundant edges. Edge \( e \) is pictured only if it lies on the longest path between \( u \) and \( v \) for some vertices \( u, v \) from \( V \)).

**C. Access Graph Compression**

Since programs can easily execute several thousand system calls in less than a minute and each system call is represented by a vertex, the size of graphs built from execution traces quickly becomes unmanageable. Moreover, graphs often contain redundant information which obfuscates the graph rather than carries useful information. Therefore, after constructing the graphs we compress them by removing redundant information and condense frequently used structures into single vertices.

1) **Removing Repetitions and Redundant Information**

There is usually a lot of repetition in program execution which carries little information. In fact, two graphs representing the same behavior can differ just by the number of repetitions of individual subgraphs and look very different. We replace repetitive structures that are connected via modifying the same set of kernel objects by a single occurrence. There are three basic types of repetition:

- **Sequence** is a repetitive structure where each member modifies the same kernel object.
- **Parallel Sequence** is represented by identical parallel subgraphs having the same predecessors and successors. It represents multiple execution of the same functionality accessing the same kernel object for reading.
- **Loop** is a sequence where each instance uses a kernel object created by a previous instance and outputs a kernel object consumed by the next

Repetitions can be composed of subgraphs which are isomorphic to each other. Sequence and loop repetitions are replaced by the first member of the component while absorbing the outgoing edges from the last component and introducing a backwards edge, creating a loop. The parallel sequence is simply replaced by only one occurrence.

2) **Replacing frequent subgraphs by single vertices**

Subgraphs often occur in many places and in traces from several programs. This is due to the reuse of the code and by executing the same functionality in multiple contexts. In particular, all the API functions which are composed from more than one system call will inherently show up as a frequent subgraph structure in many places. Since they always represent the same functionality there is no need to keep several copies of them and each instance can be replaced by a single node. This operation further reduces the data size and thus reduces the cost of subsequent processing. Since API functions are executed frequently and some of them translate into relatively large subgraphs, this size reduction can often be significant. Moreover, it also identifies structures which should always remain the same. Any deviation in standard API functions is by itself suspicious.

**III. Significant Functionalities**

Functionality represents a part of program execution that is logically connected and achieves some recognizable goal. It can be observed as a frequently occurring pattern in kernel object access graphs. The simplest possible functionality is just a single system call. Although such functionality is not very meaningful, the frequency of execution of some system calls can provide some information about the nature of the program and therefore even this simple functionality can be useful. More complex functionalities can have constraints on their arguments and contain several system calls and dependency relations between them. Significant functionalities are those which provide good descriptive power to program classification. In this work, we are primarily looking for malicious functionalities, i.e. functionalities which are often executed by malware while they are rare in legitimate software. The ideal malicious functionality is such which is never executed by legitimate programs. Example of such functionality would be infection of executable files as performed by classical viruses. However, most functionalities cannot be declared as 100% malicious. For example, code injection is usually a strong sign of malicious behavior, but it can be used by legitimate programs for debugging or profiling purposes. In theory, there may be functionalities that are seldom executed by malware while they are rare in legitimate software. The ideal malicious functionality is such which is never executed by legitimate programs. Example of such functionality would be user interaction. Although some of the malware (e.g. rogue antiviruses) can interact with the user, most malicious programs try to stay hidden. Thus, we define significant functionality as such a functionality for which the probability of occurrence in malicious or legitimate programs is above some threshold.
A. Functionality Specification

There are two major requirements for the functionality specification model. First, it has to be powerful enough to allow for a large set of possible patterns such that reasonably complex functionalities can be expressed. Second, it needs to be resistant to modifications since malicious code can employ some behavioral obfuscation techniques.

1) Colored Petri Nets

Previous work [3] showed that Colored Petri Nets provide sufficient expressive power while allowing for efficient detection (with some additional constraints). Informally, CPNs consists of a set of places and a set of transitions connecting the places. Each place has an assigned set of token types also called the color set. This specifies the types of the tokens that can be assigned to this place. Transitions connect the places and represent possible state changes in the modeled system. Each transition connects a set of input places to set of output places. It can have constraints defined as predicates over the sets of tokens present at input places. These constraints determine whether the transition is enabled in the given state. Transitions can also introduce new tokens in the output places. New tokens are defined as a function of the set of input tokens. More rigorous description of CPNs can be found in [9].

2) Functionality Representation

We generate the functionality specification in the form of Colored Petri Nets constructed so that they match some induced subgraph of the kernel object access graph. In addition, we allow constraints of system call arguments, so that highly specific functionality can be expressed. The obtained CPN is closely related to the access graph it was generated from. There is one place representing functionality recognition. Every other place represents a vertex in the original graph. It can contain tokens representing the underlying system call and its arguments. Thus, every token in our CPN represents an operation over a kernel object. There are two kinds of transitions. The first transition type represents a path in the underlying access graph. It connects two places if there is a path between them in the access graph and no other vertex belonging to such path is contained in the CPN. Such a transition represents a next step in the functionality matching. It introduces a new token in in the output place representing the underlying system call whenever this system call is executed and there is a matching token in the input place. The second type of transition is functionality recognition. It is connected to all the places and in case there is a matching token in every place, these tokens are removed and the functionality recognition token is placed in the special recognition place. Since tokens represent kernel object operations, they can be removed from the CPN only when the underlying objects cease to exist (there is no living reference) or there is a positive match to a known signature. Exceptions to this are tokens representing permanent objects like files and registry keys. These tokens are deleted only when the underlying file (registry key) is overwritten or in the case of a positive signature match. Since tokens are removed only in certain cases, the CPN’s memory footprint would eventually reach unacceptable levels. Especially in case of files, tokens could stay in the CPN for virtually unlimited time. The problem might be solved by adding transitions and removing tokens in

B. Functionality Extraction

Functionalties can be identified as frequent subgraphs of the kernel object access graph. However, subgraph mining is computationally expensive and the graphs, though considerably reduced, were still large in size. Moreover, as we observed from manual inspection of the obtained graphs, most of the frequent structures seem to be contained together. This is because functionality as we defined it is usually executed at one time without many additional system calls related to the same Kernel Object. For these reasons, we employed a heuristic technique which first converts graph components into a string and then performs a search for repeated factors[10]. Since the functionalities do not have to be exactly the same, we search for repeated factors with edit distance lesser or equal to k, where k is an experimentally set constant. Graphs can’t be expressed as strings without losing information so we can’t rely on a string search algorithm to find all functionalities. However, our heuristic worked well in our experiments and computing common string factors is a much more efficient operation than frequent subgraph mining. Although the feature of functionality locality obviously can’t be relied upon in case of malware detection, as malware can employ obfuscation techniques, it can be exploited as of now to observe malicious functionalities in malware which does not use behavioral obfuscations which would break this property (we have not come across a malware sample which would use such techniques). If in the future malware begins to apply behavioral obfuscation techniques, a proper subgraph mining algorithm might be required.

C. Functionality Recognition

Signatures can be recognized simply by CPN simulation with monitoring of the detection place. Generally, CPN

![Figure 2 CPN Representation of code injection functionality](image-url)
simulation can be a very complicated task with possible non-deterministic execution. Therefore, it is worth mentioning that simulation of CPNs generated by our approach is fairly straightforward and deterministic. In our case CPN simulation is performed by simulating execution of the transitions. The simulation becomes nondeterministic, if the net contains a reachable state (marking of places) where two (or more) conflicting transitions are enabled. It is easy to see that this is possible only when two detection transitions with overlapping sets of inputs are enabled, or when a transition symbolizing object deletion collides with the detection transition over the same object. The latter case is easily solved by giving priority to the object deletion. The first case can be solved by allowing all detection transitions to be executed “in parallel”. All enabled detection transitions are first executed and then, at the second step, their input tokens are removed. Consequently, the simulation algorithm is always deterministic. We can split simulation of the CPN into individual steps, each representing an intercepted system call. For each intercepted system call, we need to check all the input places for matching tokens (tokens representing operations over the same kernel object). In addition, we need to check all the detection transitions which can possibly end with the given system call.

IV. Malware Detection

Significant functionalities can be used as features and fed into some existing classifier. The classification of feature vectors has been studied extensively in machine learning and there are many algorithms which can be used for this purpose. However, this work focuses on extracting functionalities and investigating their discriminative power. We perform classification only as a proof of concept, therefore, we use only a simple scheme for malware detection. First, we pick only those functionalities which occur only in malware and are shared at least by several instances. We then compute minimal graph edit distance to a subgraph found in benign program traces (we again use a string matching heuristic to find possible candidates). Every malicious functionality is assigned a score based on how many malware samples contain this functionality multiplied by the minimal graph edit distance to the benign subgraph relative to its size. The program is then classified as malicious if the sum of the scores of executed malicious functionalities is above some experimentally set threshold.

V. Evaluation

We evaluated our scheme on execution traces obtained from several benign and malicious programs running on Windows XP. System call traces were recorded from our driver which intercepted system calls with their arguments by hooking into the SSDT table. Since we wanted to evaluate our approach in general conditions without any prior knowledge about importance of individual system calls for security, we intercepted all of the calls referenced by the service table, except a few for which we could not find the correct specification of input arguments. We used our driver to obtain execution traces from several malicious and benign applications.

- Benign:
  - Firefox 4, Internet Explorer 8, Windows Media Player, TotalCommander, Notepad++

- Malicious:
  - Trojan.Win32.Vilsel.ual, Virus.Win32.Parite.a,
  - Worm.Win32.VBNA.a, Worm.Win32.VBNA.jla,
  - Worm.Win32.VBNA.jmd, Worm.Win32.VBNA.jph,
  - Packed.Win32.Krap.b, Packed.Win32.Krap.gy,
  - AdWare.Win32.AdMoke.bj

Malicious programs were obtained from the Offensive Computing web site [10] and include malware samples of different types and from several families. Benign programs were selected to represent a typical user setup. We joined the obtained samples into three testing traces so that each trace consisted of several malware types and several benign programs. The results can be seen in Table 1.

<table>
<thead>
<tr>
<th>Trace #</th>
<th>Syscalls #</th>
<th>Unique Graph Components #</th>
<th>Functionalities #</th>
<th>Malicious Functionalities #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6927937</td>
<td>1047</td>
<td>341</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>3704217</td>
<td>862</td>
<td>307</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>20719</td>
<td>217</td>
<td>49</td>
<td>9</td>
</tr>
</tbody>
</table>

Since we monitored all of the system calls, the size of execution traces grew rapidly with time, quickly becoming unacceptable. Therefore we used traces obtained only for a limited amount of time, ranging from 1 minute to 20 minutes in the case of longest execution trace. Data characterizing our traces can be found in Table 1. It can be seen that our compression scheme worked very well and allowed us to process even longer traces. The problem we faced when processing large traces was a large memory footprint (over 10GB for large traces). This was caused by the fact that some of the constructed graphs had way over ten thousand nodes before the compression took place and because we applied compression on the whole graph at once so we initially need to keep the whole structure in the memory. The memory footprint reached unacceptable levels for long traces and limited the length of our experimental traces. This problem can be solved in the future by employing some incremental compression scheme where graphs can be reduced in real time.

We obtained the traces by simulating normal user work over the given time, while capturing traces from all running processes. Traces of malicious programs were naturally considerably shorter because most of the malware needs only limited time to perform its task. However, some of the more recent malware instances produced traces of significant size even in a short time (especially the sample of rogue antivirus). We tested all malware samples individually so that they would not interfere with each other. We joined the traces afterwards to enable the search for common functionalities.

A. Found Malicious Functionalities

Several malicious functionalities were discovered in our testing samples which allowed for classification of given malware. Two of the discovered functionalities were easy to
analyze. One was a classical example of code injection while the other was “unpack and execute”. It is not surprising that this particular functionality was found in almost all malware instances (with some minor differences) since malware today often employs code packers to avoid detection by binary signatures. Most of the other functionalities were hard to analyze and often consisted of accessing some particular operating system resource.

B. Detection Results

For the tested samples, we built a detector differentiating malicious from benign programs with only one false negative and no false positives by detecting malicious functionalities. However, size of our testing sample was limited and in general results can be expected to be worse. Machine learning techniques can be used to further enhance the detection performance. Rather than looking for strictly malicious functionalities, we can take the whole frequency vector as an input (feature vector) for some classifier algorithm.

VI. RELATED WORK

This paper is a continuation of previous work on malicious functionality specification using CPN technology [3]. While previously malicious functionality had to be defined by an expert user, our work enables automation of the whole process. It facilitates combining more malicious functionalities in the decision process thus decreasing the risk of false positives. Moreover, following kernel object dependencies rather than mere system calls allows us to avoid the need for employing a de-obfuscation engine and possible errors or incomplete specifications introduced by this engine.

Our work is closely related to automatic behavior signature generation and malware clustering. Both problems were approached by several research projects.

Kolbitsch et. al. [4] presented an approach to behavioral matching based on following dependencies of system call arguments. They automatically extracted the dependencies by running process in special execution environment which enabled them to capture even functional dependencies. In the behavior-matching phase, expected values of complex functional dependencies were computed by pieces of the original program obtained by applying program slicing techniques. The resulting behavioral profile was thus closely tied to the particular executable and detection rate decreased significantly even for slightly altered samples of similar functionality (same family). In contrast, we do not use any functional dependencies and do not restrict activity to the context of one process. Thus, the malicious functionality described by our system is less tied to the particular malware sample. Moreover, we do not create a profile of a particular process, instead selecting functionalities which suggest malicious behavior.

Authors in [5] developed a method for constructing near-optimal behavior signatures from the behavior graph introduced in [4]. They use models of both benign and malicious programs to detect malicious behaviors and use concept analysis to find optimal signatures. They claim to achieve an 86% detection rate on previously unseen malware with no false positives and ability to detect the same malicious behaviors even across the malware families. As in [4], they use behavior graphs following data dependencies which capture even complex semantic relationships. Thus, malware employing advanced behavior obfuscation such as using covert channels [7] to pass the information could escape detection.

Lanzi et al. [6] built a system-centric model of benign programs observing sequences of executed system calls. In particular, they investigate if benign and malicious programs can be discriminated by the n-grams of executed system calls. They came to the conclusion that system call sequences executed by benign programs are too diverse. However, they conclude that malicious programs can be identified by their access patterns to resources such as files or registry keys.

VII. CONCLUSION

We described a technique for modeling behavior of programs by tracking how they access and modify Kernel Objects. Graph compression techniques allowed for compact representation of program behavior and we were able to find patterns frequently occurring in malware samples representing malicious functionalities. Our results showed that these functionalities can be used to identify malware and introduced a method for automatic generation of significant functionalities in the form of Colored Petri Nets which can be used as an input to a classifier algorithm. We proved the feasibility of the concept by applying a simple detection scheme capable of detecting given malware samples while causing no false positives.

VIII. ACKNOWLEDGEMENT

This research is funded by the Air Force Office of Scientific Research (AFOSR). The authors are grateful to Dr. Robert Herklotz of AFOSR for supporting this effort.

REFERENCES
