MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

Eng. Gheorghe Florin HĂJMĂȘAN

PhD THESIS
– ABSTRACT –

CONTRIBUTIONS TO BEHAVIOUR BASED MALWARE DETECTION

Scientific coordinator,
Prof.Eng. Radu Gabriel DĂNESCU, PhD

Committee:

PRESIDENT: Prof.Eng. ***Ioan Salomie***, PhD - Technical University of Cluj-Napoca;

SUPERVISOR: Prof.Eng. ***Radu Gabriel Dănescu***, PhD - Technical University of Cluj-Napoca;

MEMBERS: Prof.Eng. ***Mitică Craus***, PhD - "Gheorghe Asachi" Technical University of Iași;
Conf.Eng. ***Adrian Dărăbanț***, PhD - Babeș-Bolyai University of Cluj-Napoca;
Conf.Eng. ***Adrian Coleșa***, PhD - Technical University of Cluj-Napoca.

– Cluj-Napoca –
2021

str. Memorandumului nr.28, 400114 Cluj-Napoca, România
tel. +40-264-401200, fax +40-264-592055, secretariat tel. +40+264-202209, fax +40-264-202280
http://www.utcluj.ro/

1. Introduction

The amount of malicious software, shortly known as *malware*, is continuously rising, alongside the diversity and complexity of malware. Statistics provided by AV-Test Institute [Insa] indicate that the total number of malware in 2020 was over 1.1 billion samples. A steep growth in the number of malicious files that began in 2012 represents the moment when the traditional, signature-based security solutions started to struggle and the need for more advanced behavior-based technologies has arisen.

The types of risks to which both individual users and organizations are exposed range from data loss to identity theft and financial loss. As malicious software has become the foundation of a highly profitable industry [BBC, McG18], malware authors are continuously trying to find new and more sophisticated methods to avoid detection [K$^+$20, MD18]. All these indicate there is a real need for innovation and the development of efficient and proactive security solutions. Behavior-based dynamic malware detection is the most proactive detection method.

This thesis aims to improve dynamic malware detection by proposing solutions to some of the most important challenges faced in this field. The thesis focuses on developing a proactive behavioral malware detection solution that dynamically monitors the behavior of processes, uses advanced behavioral heuristics to identify malicious actions and evaluates the behavior of processes. The proposed security technology should be capable of detecting new and previously unknown threats and should be suitable for real-time protection. In addition, the solution aims to be resilient to evasion, when a malware's payload is distributed among multiple processes and should also have a small overall performance overhead and a short response time to new threats.

One of the challenges in dynamic malware detection is the fact that a single action is usually insufficient to distinguish between malware and clean applications, and taking multiple actions into account is necessary to achieve a better accuracy. In addition, not all malicious actions can be considered equally significant. Some have more severe consequences, while others with minor immediate consequences, may indicate the malicious nature of process when performed together with other actions. An incorrect behavioral evaluation may result in too many false positives or poor detection.

Another challenge comes from the way dynamic detection solutions evaluate threats. Currently, most of dynamic malware detection techniques evaluate the behavior of a process and, using a set of rules, decide if that process is malicious or not. The rule set must accurately differentiate between malicious and non-malicious processes. Because a balance between detection rate and number of false positives must be assured, a dynamic detection system can not be too aggressive when evaluating a single process. Advanced malware may take advantage of this lack of aggression and evade being detected by separating malicious actions into multiple processes through process creation or code injection.

The third challenge in dynamic malware detection addressed by this thesis is the performance overhead. Monitoring the behavior of processes dynamically, at run-time, by installing filters (such as file system, registry, process and APIs filters) implies certain costs in terms of performance. More often than not, behavioral detection solutions have been a source of frustration for users, even if they provide a higher level of protection than traditional, signature-based ones. Considering this, it is very important to find ways of improving the performance of behavior based malware detection solutions, and at the same time to maintain the detection rate at the same level, which is a significant challenge.

The fourth improvement to dynamic malware detection proposed by this thesis addresses the response time to new, undetected threats. With some effort, experienced malware developers can be one step ahead of security researchers. They can test their malicious software against security solutions and adapt the malware until it evades detection. Considering this, advanced security solutions should be both *proactive*, so that they can generically detect new malware samples with known malicious behavior, as well as *reactive*, so that detection for new malicious behavior can be added as soon as possible. When a malware with an undetected behavior is released, users are vulnerable to attacks until their security product is updated with heuristics capable of detecting that malicious behavior. This time gap can be lengthy in some cases, because of all the steps in the development process of a security solution, which include careful development, rigorous testing and that certain certifications are obtained before an update can finally be deployed to users.

1.1 Thesis Outline

The thesis is organized into eight chapters. The first chapter presents the motivation for this research. The second chapter describes the concepts and definitions used throughout the thesis, specific to the malware detection domain. Chapter 3 presents existing research regarding malware detection.

Chapter 4 starts with an overview of the proposed solution's architecture and a description of the components, then highlights the advantages of the scoring engine over artificial intelligence algorithms. The chapter describes the key elements identified that are needed to obtain an advanced scoring engine and presents the evaluation process. The chapter explains how the detection model can be extended and maintained in response to the continuously evolving threats. The malware detection and false positives rates obtained by evaluating the solution are also presented, highlighting the contribution of the key elements of the scoring engine to the malware detection rate.

Chapter 5 presents a method to detect malicious groups of processes instead of single malicious processes, a method for constructing such groups, together with a way to evaluate their actions so that malware groups can be detected. A method to clean the infected system based on the actions performed by the processes in the detected group is also presented.

Chapter 6 describes two key methods proposed by this thesis to improve the performance of a behavior based detection solution: asynchronous heuristics and a dynamic reputation system. Asynchronous heuristics help reduce the overhead perceived by the user, while a dynamic reputation system helps reduce the system-wide resource consumption.

Chapter 7 presents the solution proposed by this thesis to reduce the time to market of a behavioral detection solution and improve it's reactivity. We propose using an interpreter virtual machine, called Behavioral Virtual Machine (BVM), that interprets behavioral heuristics stored as bytecode signatures. This solution allows researchers to rapidly add new detection algorithms (heuristics), having the advantage of deploying them to users as a signature update. In addition to improving the reactivity of the security solution, BVM also provides a way to quickly create prototypes for new ideas of heuristics.

Chapter 8 presents the conclusions of this thesis and highlights the main contributions to the field of dynamic malware detection.

2. Proposed Solutions

2.1 Dynamic Behavior Evaluation for Malware Detection

We propose a set of dynamic behavior evaluation concepts, valuable to any security researcher who wants to develop an advanced evaluation mechanism for a behavior-based security solution with high detection accuracy. Furthermore, these concepts were used to implement such a solution. The solution was evaluated, with results indicating a high detection rate and few false positives. It is also capable of identifying relations between various entities resulting in a panoramic perspective of the protected system. Moreover, the architecture is simple and easy to adapt or extend to detect the newest threats.

The high-level architecture of the proposed solution is illustrated in Figure 2.1. The solution can be controlled and configured using the Security Application Service - the user can change the settings, add exclusions and make decisions regarding the detection. For example, the user can decide between allowing a detected process to run or to block it. In addition, through this component the security solution may send telemetry to a server or make queries to a Cloud component for various services (e.g. hash based whitelisting).

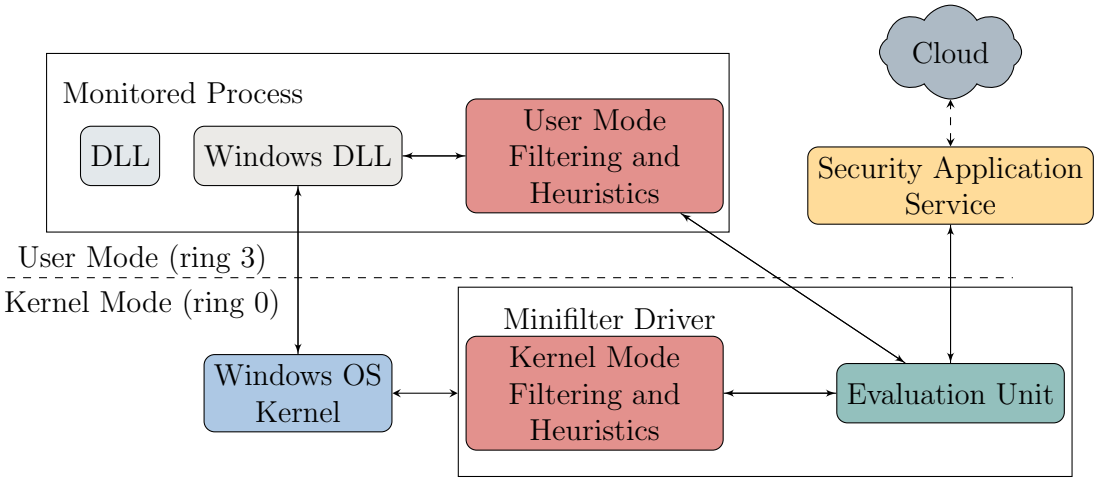

Figure 2.1: Behavioral Detection Solution Architecture

A *minifilter* driver was implemented to monitor the actions performed by processes from Kernel-Mode. It registers *callback routines*, that are called whenever an event occurs in the *file system*, *registry keys* or *processes*. The actions performed by processes are also filtered at User Mode level, using *API interception* (hooking) through a Dynamic Link Library (DLL) injection into the monitored process. Whenever the filter components intercept an action, the filtered information is dispatched to the heuristics. They will process the intercepted information and send alerts to the *Evaluation Unit*, if necessary.

All the alerts triggered by the heuristics are received by the *Evaluation Unit*, which processes the alerts and the associated scores. If the Evaluation Unit determines that a process should be detected, a notification is sent to the the *Security Application Service* component, in order to display a notification in the user interface.

The proactive security solution is based on *behavioral heuristics*. They are located in the same components as the filtering mechanisms, as shown in Figure 2.1. Each heuristic is an algorithm (function) that analyzes the actions intercepted through monitoring. More

precisely, they are called whenever a targeted event occurs, process the information related to that event and decide if a targeted action is being performed. If such an action is identified, the heuristic triggers an alert (i.e. the heuristic is triggered). Each alert has an associated score and is sent to the *evaluation unit*. Examples of actions that can be detected using heuristics include creating a copy of the original file, hiding a file, injecting code into another process, creating a startup registry key, deactivating some critical OS functionalities (e.g. updates) or terminating critical processes. The solution can be very easily extended with new heuristics, without affecting the existing ones.

2.1.1 The Evaluation Unit

The evaluation unit (scoring engine) represents a series of algorithms and operations applied on information resulting from heuristic alerts, in order to evaluate the behavior of processes. These operations were identified over time, as a result of practical experience in behavioral based malware detection (over 10 years).

We propose using an incremental scoring engine, that evaluates alerts issued by the heuristics. Each heuristic sets a different score, representing an integer value. The scoring engine computes a general score for the process and compares it to a detection threshold. If this score reaches that threshold, the process will be declared malicious. The *threshold*'s value is adjustable between three levels of aggressiveness, to accommodate the user's preferences: better detection or fewer False Positives (FPs).

False positives are a big concern when developing a security solution intended for millions of users. To mitigate against this issue, we propose an *exception mechanism* to allow a process to execute a certain action, that would otherwise be detected by one of the heuristics. More precisely, the points that would have been given by that heuristic are either set to 0, or are lowered with a certain percentage.

Malware that delegate some of their payload to one or multiple child processes can be harder to detect because some heuristics are triggered on the parent process and some on the child entities. This increases the risk that the malware attack will remain undetected. In some cases, one of the child processes is detected, but this may not be enough to clean the system, if the parent process remains undetected. Another similar method is when malicious code is injected into a process.

To detect malware that use similar evasion mechanisms, we introduce the concept of *parent process heuristic propagation*, meaning that all the malicious actions performed by the child processes are propagated to the parent process. This consists of propagating all the heuristics and the associated points to the parent process entity. The points can be propagated fully or only in part, representing a percentage of the total. This percentage or propagation weight depends on the heuristic and on the type of the involved processes. Similarly, we introduce *injected process propagation*, when heuristics are propagated from the injected entity to the injector process.

Based on the previous remarks regarding the incremental score, heuristic exceptions, parent and injected process propagation, the total score of a process is computed as shown in Equation (2.1). N_h denotes the number of triggered heuristics and $H_i(P)$ the score given by the i-th heuristic for the process P. $E(P, H_i)$ denotes the exception weight for the i-th triggered heuristic and may be considered as a function of the evaluated process P and the heuristic H_i, with $E(P, H_i) \in [0, 1]$. N_{ch} and N_{ih} represent the number of heuristics propagated from the child processes and injected processes. P'_j denotes the child process or the injection victim from which heuristic H'_j is propagated to the process P. The propagation weight is denoted as W, with $W \in [0, 1]$.

$$S(P) = \sum_{i=1}^{N_h} E(P, H_i) H_i(P) + \sum_{j=1}^{N_{ch}+N_{ih}} W(P, P'_j, H'_j) E(P'_j, H'_j) H'_j(P'_j) \ . \qquad (2.1)$$

To correctly compute the score of processes using propagation, the scoring mechanism must maintain a process collection, that acts like a database, storing information about the monitored processes and the relations between them. These relations will be used for score evaluation. We propose an algorithm for creating and maintaining such a collection.

2.1.1.1 Flags for Heuristics and Processes

A scoring engine integrated in a solution used at a large scale should be very adaptable. We propose using exception and propagation weights that will allow us to increase the detection rate and reduce the number of FPs. The weights may be seen as functions of process and heuristic *attributes*, called *flags*. Each process or heuristic can have none, one or multiple flags set at a certain moment.

Such flags are useful for a better calibration of the scoring engine. For example, a process can be marked with the flag *TypeIsBrowser*. Then, it can be excepted from heuristics that have a flag indicating they detect the access of URLs or the download of files. This way FPs on browsers, that commonly perform such actions, can be avoided. However, in addition to the *TypeIsBrowser* flag, the process may have a flag indicating it is an injection victim (*FlagIsInjected*) or that it is being exploited (*FlagIsExploited*). In this case, the previous exception may not be taken into consideration, allowing a better detection by not excepting a formerly clean process that was compromised through a code injection or an exploit of a vulnerability.

2.1.1.2 Process Re-evaluation Rules

Heuristics and flags can be combined in *re-evaluation rules*. If one of the rules matches, it will trigger another heuristic with an associated score. This method can significantly improve the detection rate, especially if the initial heuristics and flags may not indicate that the process is malicious per se, but do when regarded as a whole. Such rules can be easily written in signature files, as presented in the following example.

```
Flag2              // flag2 must be set
!Flag8             // flag8 must NOT be set
Heur24             // Heur24 must be triggered
!Heur71            // Heur71 must NOT be triggered

Trigger: Heur149   // will trigger Heur149
Points:  35        // with 35 points
```

Listing 2.1: Example of re-evaluation rule

2.1.2 Extending and Maintaining the Detection Model

Malicious software is continuously evolving in order to use the latest features provided by operating systems and programming languages, as well as to exploit the latest unpatched (zero-day) vulnerabilities. If the current detection model does not identify a new

malicious technique, the current model must be extended, usually by implementing new heuristics. The detection model also needs to be updated because of the clean applications. With new releases and updates being performed everyday, there is a chance that clean applications will exhibit new behaviors that can generate additional false positives. On the other side, once identified by heuristics, such new behavior can help the detection model to better differentiate clean applications from malicious software. All these reasons emphasize the need for a detection model that is easy to maintain and extend.

We presented how to identify actions that can be monitored and how to implement new heuristics to analyze those actions. We exemplified this process on a sample from the *backdoor trojan* malware family.

2.1.2.1 Telemetry Mechanism

The telemetry mechanism is the most important aspect in the process of extending and maintaining the detection model. New heuristics and their scores are fine tuned in lab before they are released, until the expected detection and false positive rates are reached. However, in lab testing barely covers a small fraction of the real world scenarios, so the telemetry mechanism is needed to be able to evaluate the detection in the real world.

The telemetry mechanism is enabled only with the user's consent. Every time a detection occurs, it generates an anonymized telemetry file and uploads it to a cloud service. A telemetry file can contain valuable information about the detected process and also about the system and context in which the detection was triggered. Automated systems or security researchers classify the telemetry into true positive (TP) or false positive (FP).

Each new heuristic is initially released to the real world in a testing, *telemetry-only* phase (*beta*). The telemetry of beta heuristics is processed and if the resulted statistics are as expected, the beta marker is removed. If the telemetry statistics contains too many false positives or too few detected malware, the beta heuristic is adjusted and updated for as many times as needed.

2.1.2.2 Measuring the Precision and Performance of Heuristics

Continuously evaluating and improving the quality of the heuristics and the detection model is a critical aspect for a security solution. This requires measuring the precision of heuristics. The precision is defined as the number of TP cases divided by the total number of cases (TP + FP). Using the telemetry files, the precision for each heuristic can be computed for a certain evaluation interval.

All the telemetry files are processed and classified into FP or TP cases. After this operation is done for each heuristic, a table containing the total number of cases, the number of TP and FP cases and the Precision of each heuristic is generated. This table can be sorted in decreasing order by Precision and obtain the heuristics with the best TP/Cases ratio, as well as underperforming heuristics that need adjustments.

When an application is monitored, the normal flow of the code execution is hijacked. For each filtered action, instead of running only the application's code on the CPU, the code of the heuristics is executed as well. This translates into a certain performance overhead caused by each heuristic. The time spent during each heuristic's execution can be determined by obtaining the time immediately before calling the heuristic's callback and the time at the return from the callback.

These measurements are conducted multiple times in various real world usage scenarios. For each heuristic some statistics are saved. Because these scenarios are complex

a heuristic can be called multiple times, so the number of entries in the heuristic's callback is counted. The minimum time, the maximum time and the average time spent in the callback are computed. In addition, the average overhead for a heuristic is computed as the product between the number of entries and the average time spent in the heuristic's callback. Using these statistics, the performance overhead of heuristics can be evaluated.

2.1.2.3 Adjusting the Heuristic's Points

The detection rate of a heuristic can be improved simply by adjusting it's number of points. This strategy is supported by the fact the scoring mechanism is really flexible and easy to adapt. The score of any heuristic can be easily changed or can be fine tuned for certain types of processes, without having to change the code, only by adjusting some weights. Identifying how a heuristic's points should be adjusted can usually be done by analyzing the telemetry statistics. For example, the initial assessment for a heuristic performed in the laboratory indicated that it should set 7 points for an alert. Based on the telemetry from real world users, the heuristic was classified as under-performing. The statistic shows how the number of FP and TP change if the heuristic's points are adjusted to a certain value. The results of such an analysis are illustrated in Figure 2.2.

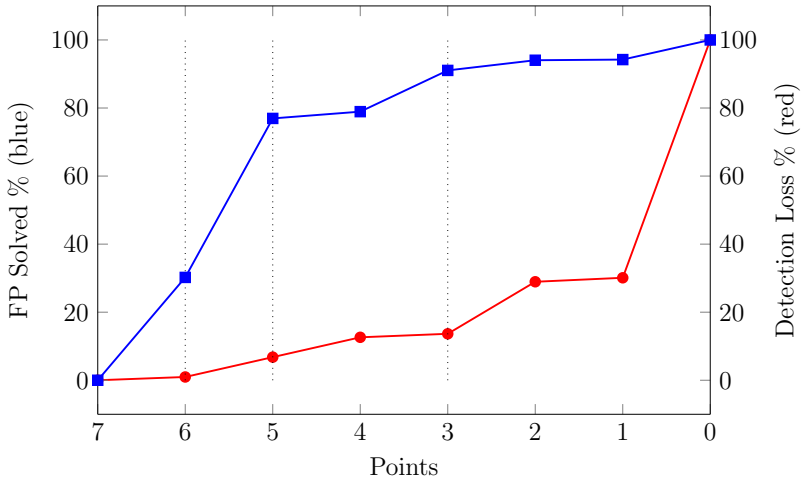

Figure 2.2: False Positive Reduction vs False Negative Increase of a Heuristic

The key idea when analyzing such charts is to search for points on the horizontal axis where the distance between the blue line (FP Solved %) and the red line (Detection Loss %) is the biggest, especially in relation with the neighboring points, and also the red line is reasonably low. Of course there are situations in which such points do not exist. This means that there is not a case of adjusting the points, instead the heuristic may need an adjustment of its logic, because it is not malware specific.

The precision of a heuristic can be easily improved using the points adjustment method. For the previous example the heuristic had a precision of 74.35% when it set 7 points. By reducing the amount of points to 3 there is a gain of over 22% in precision increase, to a total of 96.55%.

2.1.2.4 Adding Re-evaluation Rules and Flag Induced Exceptions

Using telemetry statistics, decision trees like the one presented in the Figure 2.3 can be created. The statistic was generated for telemetry containing *Heur0* heuristic, as illustrated in the root node. The telemetry files were classified into two classes: TP and FP. The next step was to search for the most discriminant feature (heuristic or flag) that is mostly present in one of the classes and almost absent in the other one. At first iteration this was the *Heur1* heuristic, which was found in 78.16% of the FP cases and in 16.51% of the TP cases. The search of discriminant features continues for each branch of the decision tree until one of the classes reaches 0% or there are no more features to make a significant difference between the classes.

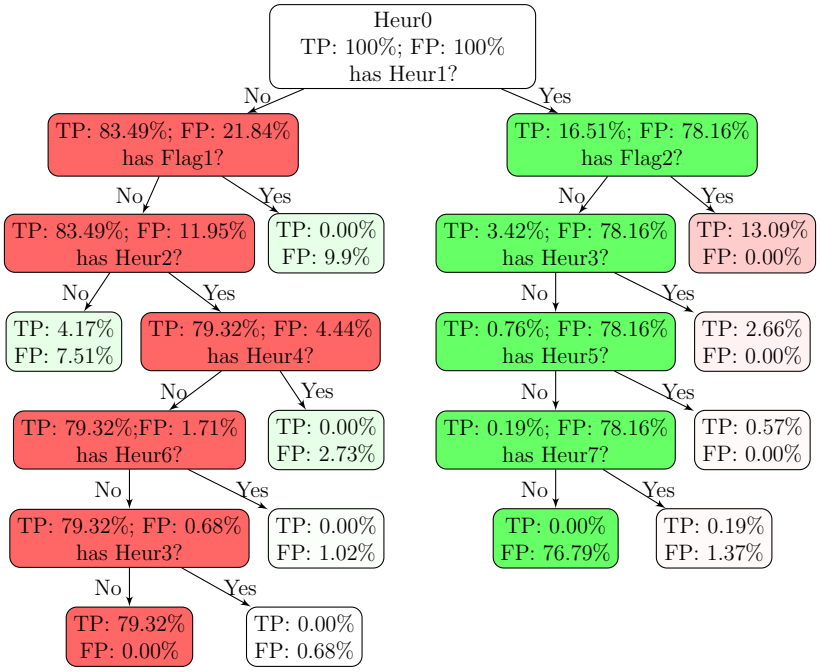

Figure 2.3: Decision Tree for Combining Heuristics and Flags

After the decision tree is generated, a security researcher can analyze it and search for nodes that have one of the percentages significantly high and the other one close to 0. New re-evaluation rules can be created to increase detection (based on the red nodes) and to lower false positives by adding exceptions for processes that have a certain combination of heuristics and flags (based on the green nodes).

2.1.3 Experimental Results

We performed detection and false positives tests to evaluate the security solution. The solution used for testing consists of 267 heuristics, 59 re-evaluation rules, 48 heuristic and 71 process flags. The results of the detection and false positives test are presented in Table 2.1. These results indicate that the security solution has a good detection rate and a very small number of FPs. However, using the exception mechanism the FP rate can be theoretically kept very close to zero.

We were also interested to evaluate the solution by comparing it to other detection mechanisms that make use of a scoring engine. Based on the description provided by

Table 2.1: Malware and false positives detection test results

	Total Files	Not detected	Detected	Detetion rate
Malicious	17069	2120	14949	87.57%
Clean	23792	23758	34	0.14%

Agarwal et al.[ASJ+16] and Treadwell et al. [TZ11] we compared several properties of these systems with our proposed solution. In contrast to the other solutions, our proposed scoring engine has higher user adaptability, is more advanced than the other two systems because it is based on behavioral characteristics, it targets diverse malicious behaviors, it is less susceptible to evasion mechanisms [MDL+12] and has a designated mechanism to mitigate against FPs. Also, the proposed solution can be extended with new heuristics without needing to alter the scores associated to other heuristics or the threshold.

2.2 Evasive Malware Detection using Groups of Processes

A strategy used by malware to avoid dynamic detection is to distribute their payload into multiple processes. Due to the large number of processes used by some malware samples and because the process trees for these samples are complex, with multiple levels and branches, correlating process scores using propagation can be difficult and less effective.

We propose a new strategy to track the actions performed by multi-process malware and evaluate their behavior. This strategy is based on creating groups of related processes, by dividing the processes into *creators* and *inheritors*. We present the way groups are influenced by process creation and code injection events and introduce group-based behavioral heuristics. We describe how the behavior of processes and groups is evaluated and how remediation can be performed on the detected entities.

A major contribution of our solution is that it automatically correlates the behavior of individual processes within a group, thus eliminating the need for a distinct correlation phase, as presented in [JHJ+16], which is both costly and complex. As a result, the heuristics are easier to develop, the evaluation is more straightforward and cleanup is better performed. Furthermore, using group heuristics and re-evaluation rules we can add detection for advanced threats by combining these features with the classification of adversarial tactics and techniques provided by the MITRE ATT&CK® knowledge base.

In addition to new concepts and an improved detection model, we also present improvements to the architecture of the security solution that make it easier to understand and maintain. A high level view of the improved architecture is presented in Figure 2.4.

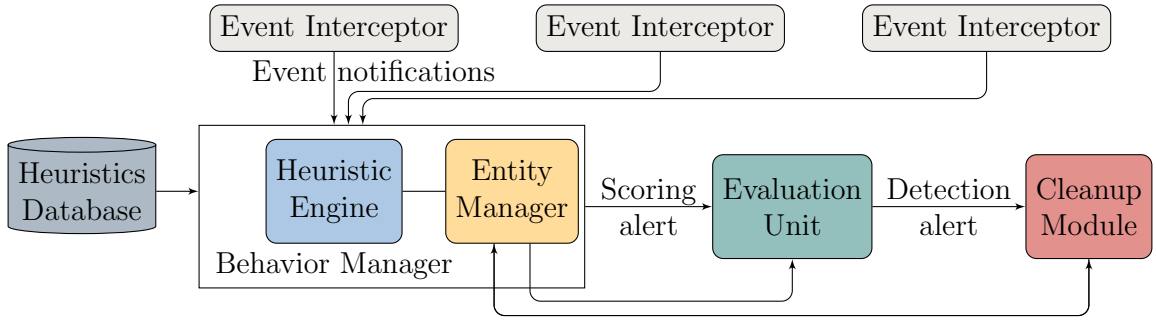

Figure 2.4: Behavioral Detection Solution

The first such improvement consists of making the interception method transparent to the heuristics. The actions performed by processes are monitored using *Event Interceptors* and encapsulated into events that are sent to the *Behavior Manager*, consisting of the *Heuristic Engine* and the *Entity Manager*. This way the heuristics only register for events generated by these interceptors. With the previous architecture, if a heuristic was based on API hooking, it needed to be implemented in User Mode, following UM development constraints. Similarly, if it was based on file system filtering, for example, it needed to be implemented in KM, following constraints specific to KM development. The new architecture allows for the heuristics to be implemented in a single place with a single set of constraints, which makes development and code reuse easier. In addition, the heuristics are not strongly coupled with the interception method.

An essential improvement is the *Entity Manager*. It uses information provided by the Event Interceptors, together with information from some heuristics (e.g. for detecting code injection) to manage the processes and groups on a system and their relations.

Another key improvement is the *Heuristics Database* in which important information used by heuristics is stored. Previously, this information was stored in the code of the heuristics, which made changing the heuristics more difficult. In addition, the Heuristics Database can also be used to store simple heuristics.

The *Cleanup Module* is also a new component. It was added to illustrate how the remediation of a system can be performed once a detection has occurred using the information provided by the *Evaluation Unit* and newly added *Entity Manager* component.

2.2.1 The Management of Groups

The *Entity Manager* maintains a collection of processes executing on the client system. This collection is dynamically updated to reflect the addition of new processes in response to process creation, and the removal of other processes in response to process termination. The Entity Manager divides processes into one or multiple *groups* and maintains a set of associations indicating the groups each of those process belongs to. An example illustrating multiple groups of processes is presented in Figure 2.5.

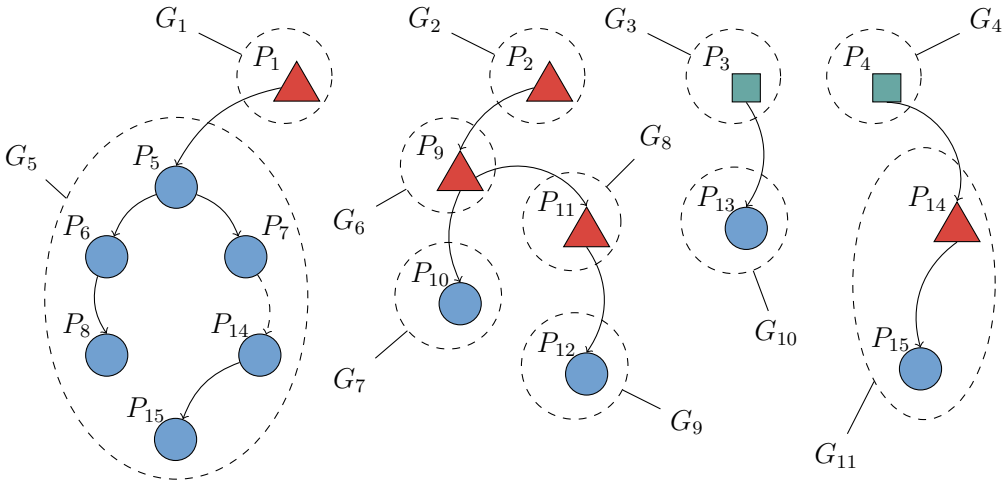

Figure 2.5: Groups of processes

The solid arrows in Figure 2.5 indicate process creation and the dashed arrows indicate code injection. The direction of each arrow indicates the direction of the relation-

ship between the respective entities: process P_6 is a child of process P_5, process P_7 has injected code into process P_{14}. Groups of related processes are represented as dashed lines, encircling those processes, and are denoted as G_i, $i \in \{1, 11\}$. For example, P_1 is the sole member of group G_1, while G_5 contains processes $P_5 \dots P_8$, P_{14} and P_{15}.

Processes are divided into three distinct categories: *group creators* - illustrated using triangles, *group inheritors* - circles and *unmonitored processes* - squares. By assigning a category - or a role - to each process, the groups of processes are much easier to identify and manage. Smaller groups, consisting of processes that are actually related, can be created, avoiding the creation of a single, large group per system. The correct identification of related processes is an essential aspect for the proper functioning of the solution, because groups are used to correlate the actions of related processes as well as perform a comprehensive cleanup of the protected system.

2.2.2 Heuristic's Evaluation

Figure 2.6A illustrates a heuristic that listens for events to identify six actions in a certain time order. If these actions are identified, the heuristic will trigger an alert. If a heuristic listens for actions performed only by a process it is called *process heuristic*. If it listens for actions performed by all the processes inside a group it is called *group heuristic*. An example of group heuristic is illustrated in Figure 2.6B. Whenever processes $P_1 \dots P_4$ perform actions $A_1 \dots A_6$ in a specific order, such a heuristic will trigger an alert for the group that contains, among others, processes $P_1 \dots P_4$. Process creation is illustrated as a zigzagged arrow.

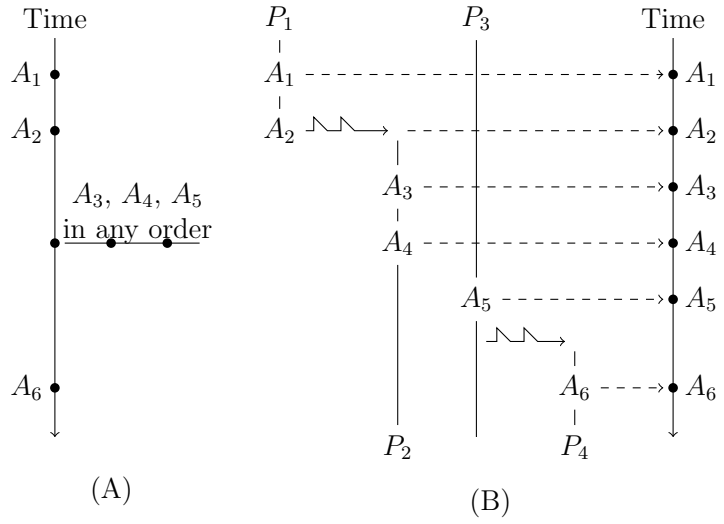

Figure 2.6: Heuristic's Logic Example

Even if the malicious actions are distributed among a group of processes and each member of the malicious group performs only a small amount of these actions, using group heuristics the security solution is able to correlate between the actions of related processes and detect such an attack.

Correlating the behavior of processes within a group using group heuristics also enables improving the Evaluation Unit, as it can maintain and update *aggregated scores* for both processes and groups. Because a group contains processes related through process

creation and injection, there is no more need to have heuristic's propagation between child and parent processes and between injection victim and injector processes. This is making the evaluation of a process more straightforward. The updated equation for computing the total score of a process is presented in (2.2).

$$S(P) = \sum_{i=1}^{N_h} E(P, H_i) H_i(P), \quad E(P, H_i) \in [0, 1] \ . \tag{2.2}$$

The total score of a process P, is denoted as $S(P)$ and is equal to the sum of all N_h scores of triggered heuristics $H_i(P)$. Each score being multiplied with the associated exception weight $E(P, H_i)$. The exception weight for the i-th triggered heuristic may be considered as a function of the evaluated process P and the heuristic H_i.

The total score of a group $S(G)$, computed as shown in (2.3), simply adds another dimension, by iterating all the N_p processes inside a group and aggregating the group heuristics scores. N_j denotes the number of group heuristics triggered on the P_j process.

$$S(G) = \sum_{j=1}^{N_p} \sum_{i=1}^{N_j} E(P_j, H_i) H_i(P_j), \quad E(P_j, H_i) \in [0, 1] \ . \tag{2.3}$$

2.2.3 Detecting Advanced Cyberattacks

Cyberattacks occur every day as malicious actors attempt to take advantage of vulnerable systems and profit at the expense of businesses and even public institutions. Very often, such an attack is orchestrated by an Advanced Persistent Threat (APT) actor, who are highly skilled and motivated, well funded and have ample resources at their disposal. Some actors are even sponsored by various nation states.

Because APT attacks are carefully planned and are often designed for a specific victim after significant time was spent researching the target organization, they are highly sophisticated and therefore challenging to detect. To solve this issue we propose an approach that combines the capabilities of our behavioral security solution with the MITRE ATT&CK [MIT] knowledge base of adversary tactics and techniques. This allows us to develop new specific threat models for advanced threat detection.

MITRE ATT&CK classifies all malware techniques in 14 tactics, that represent the reason that determines a malware to perform an action. Each tactic is broken down into multiple techniques and sub-techniques that indicate how a tactical goal is achieved by performing specific actions.

Using the Re-evaluation rules presented in Chapter 4 of the thesis, we can illustrate how a new detection model based on the MITRE ATT&CK knowledge base can be implemented. First of all, a re-evaluation rule is created for each existing heuristic to associate it with its corresponding tactic and technique. An example of such re-evaluation rule for a heuristic that identifies if a process hides a file is illustrated below. Similar rules can be created for other heuristics, such as achieving persistence (tactic TA0003).

```
HEUR_HIDE_FILE
SetTactic:       TA0005    // TA0005 Defense Evasion tactic
SetTechnique:    T1564     // T1564 Hide Artifacts technique
SetSubTechnique: 001       // 001 Hidden Files and Directories
```

Listing 2.2: Set MITRE information for HEUR_HIDE_FILE

Once all the heuristics are augmented with the associated tactic and technique, new re-evaluation rules that contribute to the total score of a process or group can be created. Such re-evaluation rules are more generic and allow referencing high level malicious tactics or techniques, regardless of the specific action that was used to perform them. An example of re-evaluation rule that combines two tactics is presented below.

```
TA0005  // TA0005 Defense Evasion is set for process or group
TA0003  // TA0003 Persistence is set for process or group
Trigger: HEUR_DEFENSE_EVASION_PERSISTENCE
Points: 100  // give 100 points
```

Listing 2.3: Rule to detect combinations of tactics

This type of behavior combinations are less common for clean application, so a large amount of points can be added to the total score to reach the detection threshold. The evaluation is also performed at group level, so it is not important which process performs which technique or tactic, as long as the actions are executed by processes that are part of the evaluated group. This allows the detection of advanced, multi-process attacks.

2.2.4 Remediation

In order to assure the best protection of a system, once a malicious entity is detected, whether it is a process or group of processes, all traces of that entity must be removed from the system and any changes performed by it must be undone. The *Cleanup Module* is responsible for taking such actions, based on information received from the Evaluation Unit and the Entity Manager. We propose an algorithm for system remediation after a detection alert was issued, that is implemented in the Cleanup Module. The thesis also enumerates several actions commonly performed by malware that are intercepted by the security solution and can be remediated using the information stored for each action.

2.2.5 Experimental Results

We implemented the presented concepts into a behavior-based solution and compared the detection rate of the group based approach and a non group based solution.

Table 2.2 shows that the detection was improved for both tested collections of malware samples with 10.61% and 5.16% . This shows that at least 5% of the malware in both collections are multi-component or multi-process, thus proving the need of changing the detection approach to a group based solution.

Table 2.2: Malware detection test

Samples	Detected (no groups)	Detected (no groups)[%]	Detected (with groups)	Detected (with groups)[%]
47933	37054	77.3%	42142	87.91%
16490	13084	79.34%	13935	84.5%

The results of the false positives test showed that the number of false positives does not change when augmenting the security solution with group awareness. This is due to the fact that the groups generated for legitimate applications usually contained a small number of processes with few triggered heuristics.

2.3 Performance Improvements in Behavior Based Malware Detection Solutions

For dynamic behavioral detection solutions, monitoring and analyzing the actions performed on a system in real time comes with an intuitive downfall: the performance overhead. This thesis proposes two solutions for this problem, that can be used separately or combined. The first approach takes advantage of the advances in hardware and uses asynchronous processing, thus reducing the impact on the monitored applications.

The second approach relies on a dynamic reputation system, based on which different monitoring levels for applications can be defined. The differential monitoring of processes according to their dynamic reputation leads to a diminished general performance overhead and also a lower false positive rate.

Both proposed approaches are practical, straightforward to implement and to use as well as easy to maintain (the dynamic reputation is self-adjustable according to the defined rules). In addition, the proposed approaches solve the performance issue globally, at framework level, without having to alter the detection model.

2.3.1 Asynchronous Heuristics

The first proposed solution for reducing the performance overhead is the use of asynchronous heuristics. As almost all of the newly released devices are multiprocessor or multi-core, this approach allows applications started by the user to run swiftly and with less impact on a core, while the behavioral heuristics run asynchronously and in parallel on another core. This allows a customer to have a great user experience and at the same time benefit from security. Situations in which certain applications or even the entire system seem to *freeze* will also be encountered less frequently or may no longer occur at all. The logic of the asynchronous heuristic framework is presented in Figure 2.7.

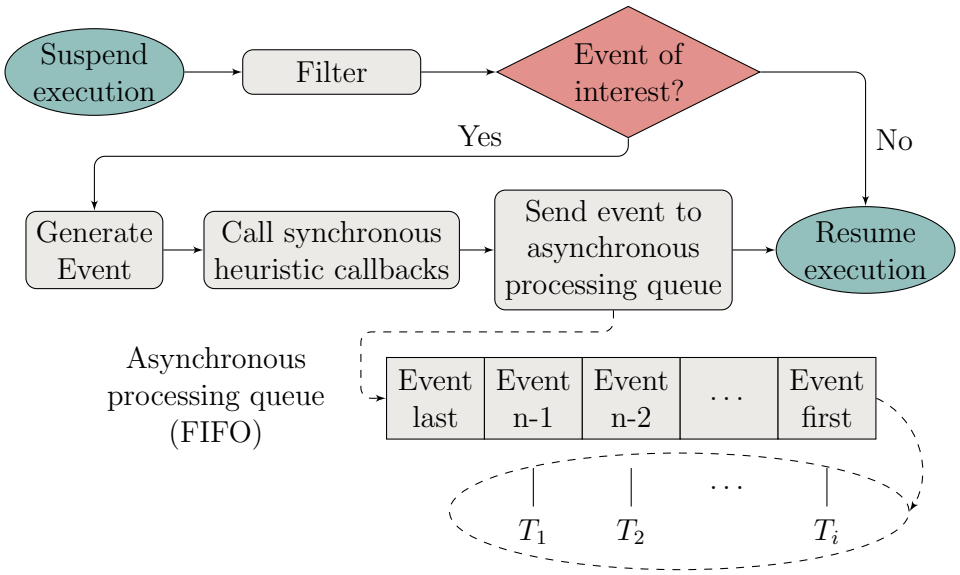

Figure 2.7: Asynchronous Processing

The first step indicates that the execution of a monitored application is suspended by a filter installed by the security solution. The filtered event is processed and if it is not of interest for the security solution, the execution of the suspended application resumes. If the event is of interest, the framework generates an event structure that is passed as a

parameter to the synchronous heuristic callbacks, if there are any. Then, the framework enqueues the event into an asynchronous FIFO processing queue and immediately resumes the suspended execution. Essentially, the filters act as producers and the worker threads (denoted T1 ... Ti) as the consumers. Once an event is dequeued by a worker thread, all the asynchronous heuristics that listen for that event will be called in the context of that thread. Then, the worker thread continues with the next event in the queue.

2.3.2 Record and Replay

Extensive testing is critical for a security solution to identify any bugs early and prevent any issues for users. Developing comprehensive testing scenarios (testcases) requires a considerable amount of time. In addition, to validate the detection rate, multiple large collections of malware need to be executed in controlled environments. However, malware is not always predictable or deterministic. Samples can alter their behavior, for example based on commands received from a control server. In order to solve these problems and improve the stability of the security solution through better testing, we propose two new features: *record* and *replay* events.

The Event Interceptors (or filters) intercept the actions performed by a monitored process and send the intercepted data to the *Event Generator*, as illustrated by Figure 2.8. The intercepted data is transformed into standardized events that are then distributed to various *listeners*. A listener is usually a heuristic that registers one or multiple function callbacks that are called whenever an event of interest occurs. The call can be set to be performed synchronous or asynchronous, depending of the listener's need. The *Record* feature is implemented by creating a listener that registers callbacks for all the events and saves them in a database file. In order to lower the performance overhead, the registered callbacks are called asynchronously.

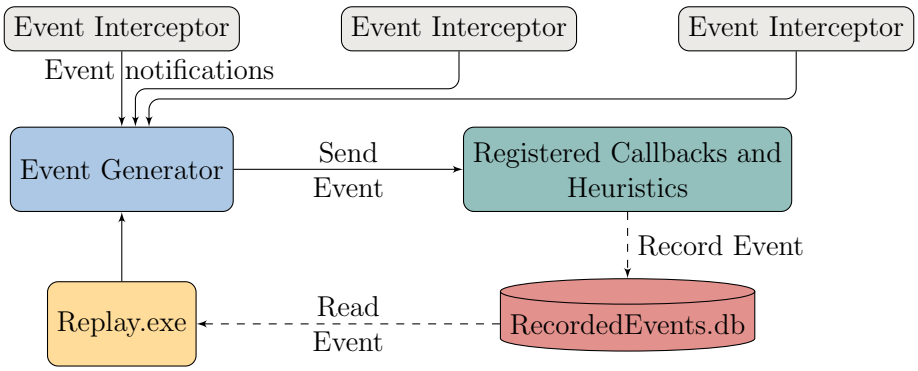

Figure 2.8: Record and Replay Diagram

To test the solution using the *Replay Events* feature, recorded events are read by the *Replay.exe* process and passed to the Event Generator, as illustrated by Figure 2.8. These synthetic events are then dispatched to the registered callbacks and heuristics as if they were generated from an action intercepted by a filter. For the heuristics and the rest of the security solution the replay process is transparent and the synthetic events are handled as if they were authentic.

Because the recorded scenarios are fixed and do not depend on the environment or a certain malware execution, we are able to verify that the heuristics, the evaluation and, in the end, the whole security solution has the same output for a certain replay. Using

Record and *Replay*, multiple functionality testcases and malware scenarios can be recorded once and replayed multiple times, especially to test new releases. This results in a good, comprehensive testing that ensures a great stability for the security solution.

In addition, new tools can be developed that automatically process collections of recorded events and propose behaviors that can be targeted for malware detection or FP reduction. The collection of recorded events can be used as training sets for ML based malware detection and to validate new detection models. A malware behavior database can be created and then queried to identify samples that manifest certain behavior, or to track the evolution in terms of techniques used by a malware campaign. Furthermore, malware samples can be classified into families based on their recorded behavior similarity.

2.3.3 Dynamic Reputation of Processes

Existing reputation systems are efficient in determining the reputation of widely distributed files and usually store the reputation scores in the Cloud. However, there are many networks isolated from the Internet or with very limited access to it, such as enterprise or corporate networks. Moreover, the computers in such networks commonly have custom applications installed. In these scenarios, a static reputation system that relies on the Cloud will be almost useless.

If a dynamic reputation is used instead, based on the fact that usually in these environments the same applications are used over and over again, if they do not perform any malicious action, a good reputation for those applications can be built in time (e.g. 1-2 days). From that moment whenever that application starts again, it will be monitored very lightly. This approach brings better performance and also fewer false positives. It also does not require another analysis of the file and neither a researcher or an automated framework to analyze it, because the reputation was build dynamically on the user's machine.

Additionally, the reputation of an application can be reported to a higher level reputation server, to be used by other systems without needing to wait for the reputation to be built. In a similar way, if an application performs a malware specific action, its reputation will be set to bad and reported to the server. A summary of the proposed reputation system is illustrated in Figure 2.9.

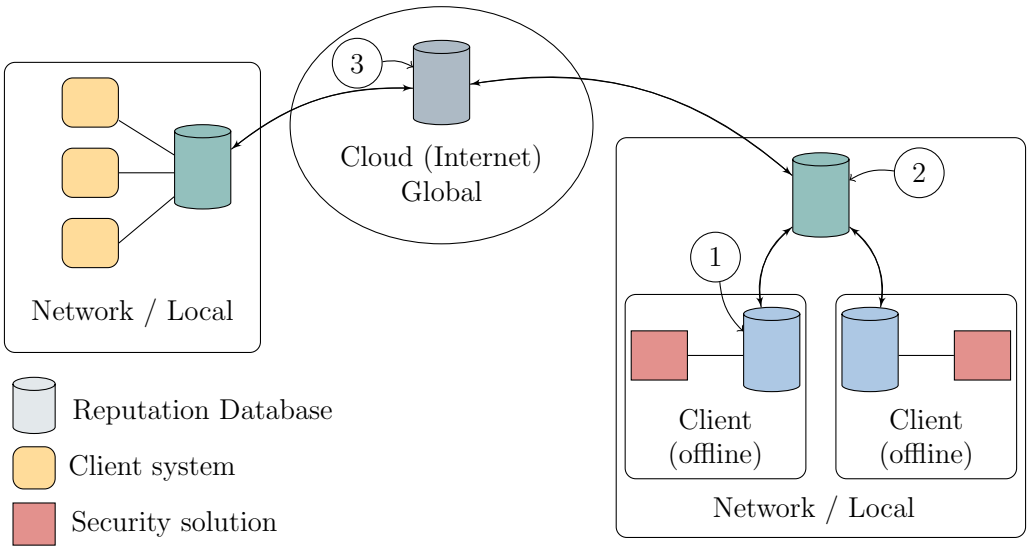

Figure 2.9: Dynamic Reputation Servers

In the most restricted way the dynamic reputation (database) server will be installed only on the client system (1) (which may be offline, no connection to Internet or to a network is required). Additionally, another server can be installed locally, at network level (2). This way all the client installed databases can query reputations already built in that network. These types of setup are very useful in enterprise environments. Furthermore, there can be a public reputation server, installed in the Cloud (3), which can be queried by any device connected to the Internet. The entire reputation system can function with any combination of these three layers: only client, only local/network, client + public/global a.s.o. These layers can also share reputations between each other.

The first challenge to compute the dynamic reputation of a process is to uniquely identify it. The methods to identify a file used for static reputation are simple, for example by computing a hash on the file (i.e. SHA, MD5). In the case of dynamic reputation a process must be identified with all its modules loaded into memory. To solve this problem we designed a so called fingerprint of a process. This fingerprint uniquely identifies a running application at a certain moment in time, by hashing all the modules loaded into memory, including pages of memory that were written but are not necessarily part of a module (e.g. are the consequence of a code injection). Each fingerprint has a reputation (untrusted score) associated to it.

When a new process is started, its fingerprint is computed and one or more of the available databases are queried for the associated untrusted score. If the fingerprint was not found in any queried database, the score will be set to a default score, according to some rules. Figure 2.10 illustrates how the reputation of a process can evolve in time.

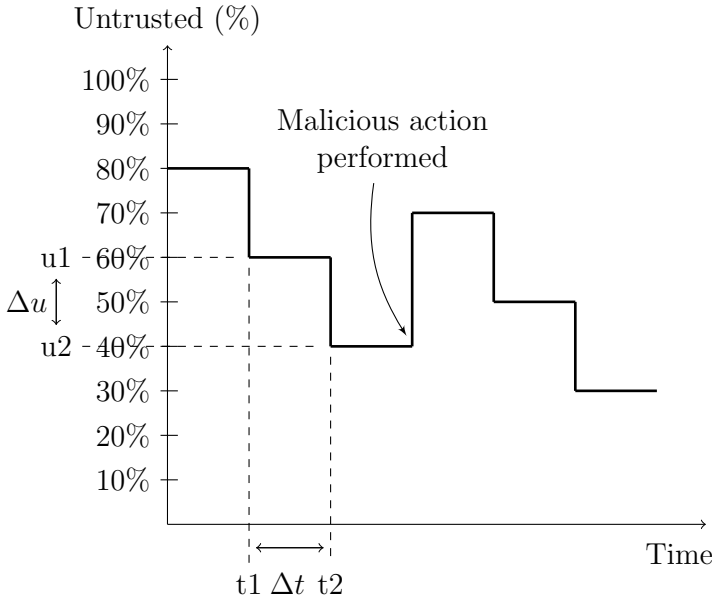

Figure 2.10: Reputation of a Process

If a certain amount of time (Δt) had passed, and that process has not performed any malicious action, its reputation will improve (the untrusted score will decrease) with Δu and the monitoring will be lighter. If the process performs a malicious action, the untrusted score increases with a preset amount associated to the action and that process will be monitored more severely.

Each level of reputation (untrusted score) has an associated level of monitoring

(events and heuristics). Processes with 100% untrusted score are fully monitored. Applications that have a fingerprint with the untrusted score of 60% are monitored by all the events and heuristics corresponding to the level 60% and below. All the events and heuristics in the upper levels are skipped for that process. The actions of processes with 0% untrusted score are not monitored by the behavioral detection solution. In a full security product, these processes may still be evaluated using traditional anti-malware signatures, protected by an anti-exploit module a.s.o.

2.3.4 Experimental Results

To ensure that the strategies proposed to reduce the performance overhead do not have a negative impact on the malware detection rate, we performed detection tests. The results showed that adding asynchronous heuristics and dynamic reputation produced no effect to the detection or false positives rate.

In order to obtain relevant results for the performance overhead tests we attempted to replicate the industry standard testing methodologies of AV-TEST Institute [Insb]. We focused on testing the performance overhead of our solution on every day usage scenarios, like: copying files, executing office tools, installing wide-spread popular applications and opening files with them. For all tested configurations, the solution with the proposed optimizations performed significantly better than the solution without any performance improvements, as illustrated in Figure 2.11. The biggest improvement was for the copy scenario, with a performance overhead decrease from 35% to 4.5%.

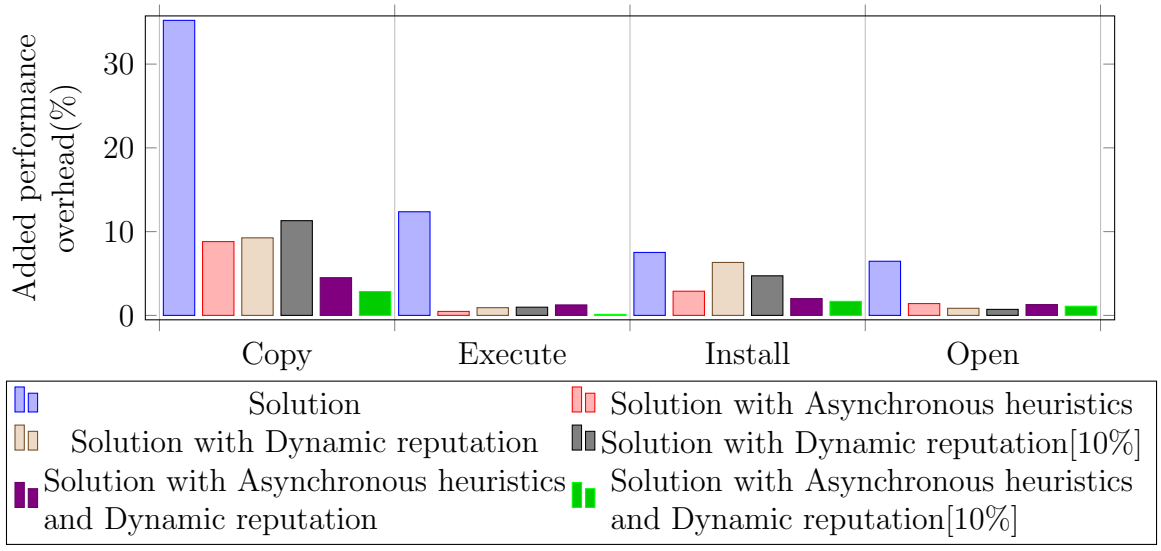

Figure 2.11: Performance Overhead Results

2.4 Bytecode Heuristic Signatures for Detecting Malware Behavior

To reduce the time required by security solutions to respond to new threats, we proposed an innovative approach using an interpreter virtual machine, that we called Behavioral Virtual Machine (BVM), capable of executing behavioral heuristics written in a domain specific language and compiled into bytecode signatures. Using this approach we

reduced by 85% the time required to update a behavior based detection solution to detect new threats, while continuing to benefit from the versatility of behavioral heuristics.

The BVM enables the quick development of new heuristics, which allows for fast prototyping. BVM heuristics can be quickly tested and deployed. If necessary, heuristics can be adapted quickly and easily, in order to correctly detect samples that evolved to be more evasive. The BVM also allows malware researchers without vast knowledge of Operating System (kernel) programming to add heuristics. In addition, the releases can be controlled and tested better, leading to more stable releases of binaries (drivers, DLLs), which means fewer crashes and incompatibilities.

Figure 2.12 illustrates the processing flow of the BVM. The *Routine Dispatcher* loads signature files, located in the *Heuristics Database*. If there are no routines that analyze events, the dispatcher ignores any event notifications and BVM is disabled.

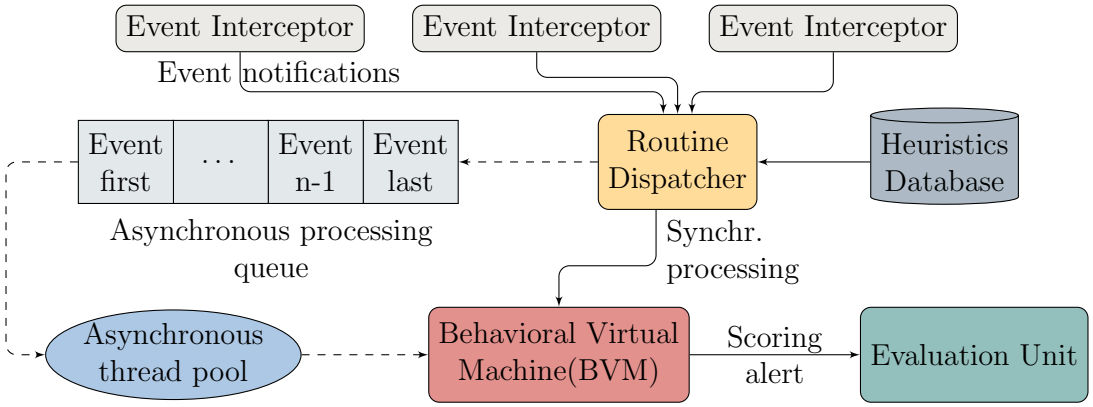

Figure 2.12: BVM Processing

When the dispatcher receives an event notification, it searches the routines that listen for the received event. BVM routines can be either synchronous or asynchronous. The execution of asynchronous routines does not block the thread from which the event originated. Next each synchronous routine is selected and the BVM is called to interpret it. After all the synchronous routines are processed, the dispatcher adds the asynchronous routines into a processing queue. The processing queue is consumed by a pool of asynchronous threads, each thread calling the BVM to interpret the first available routine from the queue. During its execution, each routine may send various information to the rest of the security solution, for example a scoring alert to the *Evaluation Unit*.

The BVM routines are similar to the callback functions in code-based heuristics, as both are called when some event they listen for occurs. The main difference is that the latter are executed directly on the processor while the former are interpreted by the BVM.

2.4.1 Structure of a BVM Signature

BVM routines are written in a domain specific language developed by us, with a similar syntax to that of the C programming language. The operators that can be used to write BVM heuristics are almost identical to the operators found in the C language.

The usefulness of BVM is best described with a practical example. Let's consider that a new malware threat, undetected by the solution, suddenly starts to affect thousands of users. For example, this new threat is a ransomware[1]. Ransomware are a type of

[1]MD5 hash: 1fbd9a11fe96c868b7ff99dbf6507dfd

malware that encrypt the documents on the user's computer and demands a ransom to decrypt them [KRB+15, TN19].

Traditionally, the response to this threat consists of writing a new *code-based* heuristic. This requires updating a complex component of the solution such as the mini-filter driver or a DLL, which is a long and laborious process. The development of code-based heuristics needs to be performed very carefully and the code needs to be validated by thorough testing. Certain certifications also needs to be obtained before the update can finally be deployed to users.

Using BVM, researchers can write a heuristic, compile that heuristic into a bytecode signature and then upload it to the update server. After the security solution is updated, the BVM will decode and execute the instructions within the signature every time a trigger event occurs. The logic of the heuristic could be the following:

```
#define STATE_ENUM_FOLDER              1
#define STATE_WRITE_TARGETED_FILES     2
#define MIN_ENUM_FOLDERS   2
process uint32 state = 0;
process uint32 noEnumFolders = 0;

triggeredby(EVENT_FIND_FIRST_FILE)
void EnumFolders(PEVT_FIND_FIRST_FILE_DATA EvtFindFirstFile) {
  if (0 != state)
    return;
  noEnumFolders++;
  if (MIN_ENUM_FOLDERS == noEnumFolders)
    state |= STATE_ENUM_FOLDER;
}
triggeredby(EVENT_WRITE_FILE)
void FileWritten(PEVT_WRITE_FILE_DATA EvtFileWrite) {
  if (STATE_ENUM_FOLDER != state)
    return;
  if (StrEq(EvtFileWrite->FileExt, "doc")
   || StrEq(EvtFileWrite->FileExt, "pdf")
   || StrEq(EvtFileWrite->FileExt, "jpg")) {
    state |= STATE_WRITE_TARGETED_FILES;
    SendAlert(EvtFileWrite->PID, HEUR_ENUM_WRITE_DOCS, 30);
  }
}
```

Listing 2.4: BVM Heuristic to Detect Ransomware

In this example, the signature contains an area for the definition of variables and two BVM routines. Each routine is called when the event it listens for occurs. For example, the *EnumFolders* routine is interpreted by the BVM only when the EVENT_FIND_FIRST_FILE event is generated (when a call to the *FindFirstFile* API is intercepted). This routine counts the number of enumerated folders and if it reaches a certain number, the *state* variable is changed. This enables the second part of the heuristic, the *FileWritten* routine, that listens for file write events. If the extension of the written file is *doc*, *pdf*, or *jpg*, it triggers an alert for the process with the process id *EvtFileWrite.PID*, a score of 30 points and the heuristic id HEUR_ENUM_WRITE_DOCS.

BVM is also useful for trying new ideas of heuristics, even if they are not related to an urgent threat. Using BVM prototypes of new heuristics can be created rapidly. If

the results of the prototype are unsatisfactory, researchers can decide to either adjust it and try again or to drop it. This way researchers have the advantage of not implementing the heuristic in the driver or DLL until they have its final version. This also makes the product more stable and more efficient in terms of bandwidth consumed with updates.

2.4.2 Experimental Results

To assure that potential bugs that can cause vulnerabilities in the security solution such as those presented in [CCK+13, Orm11] were discovered, we performed tests to validate the stability of the BVM. All erroneous bytecode-heuristics were handled gracefully resulting in no hangs, crashes, or any other similar problems.

We performed tests to measure the performance overhead of the solution containing 5 and 50 BVM heuristics, relative to a baseline of the solution without the BVM. For all the tested scenarios the performance overhead when adding BVM to the security solution is very low, around or under 2%

BVM helps to significantly reduce the *time to market* of a new heuristic, from between approximately 7-25 days for a code-based heuristic to 1-3 days for a BVM heuristic, as presented in Table 2.3. Time to market represents the difference between the moment when a security researcher starts implementing the heuristic until the heuristic is used for malware detection by the security product on the user's machine.

Table 2.3: Time to Market

Step	BVM	Code-based
Developing the heuristic	1 - 2 days	2 - 3 days
Testing (QA)	0 - 1 day	4 - 5 days
WHQL tests	0	1 - 2 days
Scheduled update	0	0 - 15 days
The heuristic is used	0	0 - 1 day
Total	1 - 3 days	7 - 25 days

Developing a BVM heuristic is faster because there are not as many strict rules to follow compared to writing the heuristic in a kernel driver or a DLL. The consequences of a bug in a kernel driver or a DLL can be severe, so the development process is strict and testing must be comprehensive. Compared to all that, a bug in a BVM routine will be detected by the BVM interpretor and that routine will simply be skipped, so there is no need to follow such strict rules when writing BVM code.

3. Conclusions and Contributions

Modern day malware represent a serious security threat to both individual users and organizations. As malware are constantly evolving and their number is continuously increasing, there is a real need for proactive security solutions that can offer real-time protection. This thesis proposes solutions for some of the most significant challenges in behavioral malware detection.

Behavioral malware detection solutions cannot usually classify a sample as malicious based on a single action, because that would lead to an excessive number of false positives. In order to produce accurate detection results, multiple actions need to be evaluated, which performed together indicate the malicious character of a sample. We proposed a novel approach to construct a behavioral malware detection solution based on behavioral heuristics to identify and analyze the actions performed by monitored samples and a scoring engine to evaluate their behavior effectively. The solution implemented using the proposed concepts achieved strong results, with few false positives and a high detection rate. As the malware landscape evolves rapidly, the easily configurable concepts that form the basis of the proposed solution make it simple to adapt in order to meet new challenges.

Another challenge is posed by malware that attempt to evade detection by distributing their payload into multiple distinct processes. To address this challenge, we proposed organizing related processes into groups and evaluating their actions both individually and as a group. We showed that by augmenting a behavioral detection solution with group awareness, the number of detected malware samples increased by over 11%. Several examples of groups created for real-world multiprocess malicious samples were also presented to show how malware attempt to take advantage of the lack of visibility in traditional detection solutions that focus on single processes.

Making sure that a security solution does not represent an inconvenience for users because of the performance overhead is another important aspect in behavioral malware detection. We proposed two approaches to solve this challenge, that do not impact the detection rate. The first approach, designed to take advantage of the prevalence of multi-core systems, is to reduce the perceived performance overhead by analyzing actions performed by processes asynchronously. We also proposed using a dynamic reputation system and monitoring processes differentially, according to their reputation, to reduce the usage of system-wide resources. Compared to other existing solutions that use reputation, the proposed dynamic reputation system has several advantages. First of all, the distributed and layered hierarchy of reputation databases make our approach better suited for networks that have restricted or no access to the Internet, such as enterprise networks. The fact that the reputation of a process is adjusted automatically based on its behavior represents another advantage over other solutions that still struggle with applications that are new, recently released or have few users. The results showed that the detection rate was not affected and a considerable reduction in the performance overhead was achieved.

Malware authors are constantly trying to develop new strategies to evade detection. Therefore, it is essential for a security solution to be able to respond quickly to new, undetected threats. The longer it takes to obtain an updated behavioral model to detect a new threat, the greater the number of users that are exposed and the higher the risk. To address this problem, we augmented our proposed detection solution with an interpreter virtual machine, called Behavioral Virtual Machine (BVM), that interprets behavioral heuristics compiled into bytecode signatures. The BVM allows the quick development,

rapid release and update of new behavioral heuristics, because the bytecode signatures do not need to comply to the rigorous requirements for the development and testing of binary components, such as drivers and DLLs, that make up the security solution. Another advantage of the BVM is that it facilitates the swift development of prototypes, that help security researchers determine if an idea for a new heuristic has a good potential or would lead to too many false positives and needs to be further adjusted and refined. The BVM also enables security researchers with limited knowledge of Operating System kernel programming to implement new heuristics. Using the BVM we were able to reduce by 85% the time needed to update the detection model in response to a new, undetected threat.

3.1 Thesis Contributions

The list of contributions presented in the thesis:

- the architecture of a behavior based security solution using a *minifilter* driver and User Mode hooking;
- the concept of behavioral heuristics used to analyze intercepted actions in order to identify malicious behavior;
- multiple concepts related to the evaluation of a process: process total score, detection threshold, heuristic's points, exceptions, heuristic propagation, heuristic and process flags, re-evaluation rules;
- a method to aggregate scores of heuristics considering the heuristic exceptions and interaction between processes;
- an algorithm to handle process life cycle events in order to maintain a collection of processes and relations between them;
- an algorithm to evaluate heuristic alerts;
- an exemplified guide for implementing new heuristics;
- a method to use the telemetry mechanism to evaluate a heuristic's precision;
- a method to evaluate the performance overhead of a heuristic;
- a method to adjust the points of a heuristic;
- a method to add re-evaluation rules and flag induced exceptions;
- improvements in the architecture of a behavior based security solution that uncouple heuristics from filters;
- multiple concepts used to detect multi-process evasive malware: groups of processes, group creator, group inheritor, entity manager, group heuristics;
- an algorithm to handle process life cycle events in order to construct and maintain the groups of processes;
- an algorithm to evaluate heuristic alerts for processes and groups;
- a method to detect advanced cyberattacks;

- a method to perform system remediation;
- a method to improve performance in behavior based security solutions using asynchronous heuristics;
- a method to improve the testing and stability of a behavior based security solution;
- a method to improve performance using dynamic reputation of a process;
- a distributed method to propagate the reputation of a process;
- a method to adjust the reputation of a process;
- multiple concepts related to dynamic reputation databases: the fingerprint of a process, monitoring levels, untrusted score;
- a method to evaluate the performance overhead of security solutions;
- improvements in the architecture of a behavior based security solution by adding an interpreter virtual machine;
- the concept of bytecode heuristics;
- a method to detect ransomware using BVM signatures;
- a method to reduce time to market and improve the solution's stability using BVM prototypes.

3.2 Practical Value of the Thesis

- The majority of concepts presented in the thesis are successfully integrated in Bitdefender products, contributing to their excellent results and protecting millions of users against cyber-threats.
- Some of the presented concepts represent Intellectual Property and are protected by 6 patents granted by the United States Patent and Trademark Office.
- Some of the concepts described in the thesis have been published and presented at academic conferences, as 4 scientific papers, thus improving the state of the art in the field of behavioral malware detection.

3.3 Conference Papers

1. **Gheorghe Hăjmăşan**, Alexandra Mondoc, and Octavian Creţ. Bytecode Heuristic Signatures for Detecting Malware Behavior. In *2019 Conference on Next Generation Computing Applications (NextComp)*, Mauritius, 2019, pp. 1-6, doi: 10.1109/NEXTCOMP.2019.8883668.
2. **Gheorghe Hăjmăşan**, Alexandra Mondoc, Radu Portase, and Octavian Creţ. Performance Improvements in Behavior Based Malware Detection Solutions. In: Janczewski L., Kutyłowski M. (eds) *ICT Systems Security and Privacy Protection. SEC 2018. IFIP Advances in Information and Communication Technology*, vol 529. Springer, Cham. (book chapter)

3. **Gheorghe Hăjmăşan**, Alexandra Mondoc, and Octavian Creţ. Dynamic behavior evaluation for malware detection. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*, Tirgu Mures, 2017, pp. 1-6, doi: 10.1109/ISDFS.2017.7916495.

4. **Gheorghe Hăjmăşan**, Alexandra Mondoc, Radu Portase, and Octavian Creţ. Evasive Malware Detection Using Groups of Processes. In: De Capitani di Vimercati S., Martinelli F. (eds) *ICT Systems Security and Privacy Protection. SEC 2017. IFIP Advances in Information and Communication Technology*, vol 502. Springer, Cham. (book chapter)

3.4 Patents granted by the United States Patent and Trademark Office

5. **Gheorghe Hajmasan**, and Radu Portase. Systems and methods for tracking malicious behavior across multiple software entities. US Patent: 10706151. Date of Patent: July 7, 2020.

6. **Gheorghe Hajmasan**, Alexandra Mondoc, and Radu Portase. Dynamic reputation indicator for optimizing computer security operations. US Patent: 10237293. Date of Patent: March 19, 2019.

7. **Gheorghe Hajmasan**, and Radu Portase. Systems and methods for tracking malicious behavior across multiple software entities. US Patent: 10089465. Date of Patent: October 2, 2018.

8. **Gheorghe Hajmasan**, Sandor Lukacs, and Botond Fulop. Behavioral malware detection using an interpreter virtual machine. US Patent: 9460284. Date of Patent: October 4, 2016.

9. Sandor Lukacs, Raul Tosa, Paul Boca, **Gheorghe Hajmasan**, Andrei Lutas. Complex scoring for malware detection. US Patent: 9323931. Date of Patent: April 26, 2016.

10. Sandor Lukacs, Raul Tosa, Paul Boca, **Gheorghe Hajmasan**, Andrei Lutas. Process evaluation for malware detection in virtual machines. US Patent: 9117080. Date of Patent: August 25, 2015.

3.5 Citations

According to the Google Scholar profile the previously enumerated published work was **cited 115 times** (until July 21, 2021).

The profile can be consulted at the following page: `https://scholar.google.com/citations?user=s0d_GxYAAAAJ&hl=en`

Bibliography

[ASJ+16] Romanch Agarwal, Prabhat Kumar Singh, Nitin Jyoti, Harinath Ramachetty Vishwanath, and Palasamudram Ramagopal Prashanth. System and method for non-signature based detection of malicious processes, April 26 2016. US Patent 9,323,928.

[BBC] BBC. Cyber-crime profits reached $3.5bn in 2019, says fbi. `https://www.bbc.com/news/technology-51474109`.

[CCK+13] Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pages 1–7, 2013.

[Insa] AV-Test Institute. Malware statistics. `https://www.av-test.org/en/statistics/malware/`.

[Insb] AV-Test Institute. Performance testing. `https://www.av-test.org/en/test-procedures/test-modules/performance/`.

[JHJ+16] Yuede Ji, Yukun He, Xinyang Jiang, Jian Cao, and Qiang Li. Combating the evasion mechanisms of social bots. *Comput. Secur.*, 58(C):230–249, May 2016.

[K+20] Sushil Kumar et al. An emerging threat fileless malware: a survey and research challenges. *Cybersecurity*, 3(1):1–12, 2020.

[KRB+15] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2015.

[McG18] Michael McGuire. Into the web of profit: Understanding the growth of the cybercrime economy. *Bromium Report*, 2018.

[MD18] Steve Mansfield-Devine. The malware arms race. *Computer Fraud & Security*, 2018(2):15–20, 2018.

[MDL+12] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1-2):1–13, 2012.

[MIT] MITRE. Enterprise tactics. `https://attack.mitre.org/tactics/enterprise/`.

[Orm11] Tavis Ormandy. Sophail: A critical analysis of sophos antivirus. *Proc. of Black Hat USA*, 2011.

[TN19] Aditya Tandon and Anand Nayyar. A comprehensive survey on ransomware attack: A growing havoc cyberthreat. In *Data Management, Analytics and Innovation*, pages 403–420. Springer, 2019.

[TZ11] William Scott Treadwell and Mian Zhou. Risk scoring system for the prevention of malware, October 11 2011. US Patent 8,037,536.