Performance Improvements in Behavior Based Malware Detection Solutions

Gheorghe Hăjmăşan, Alexandra Mondoc, Radu Portase, Octavian Creţ

Performance Improvements in Behavior Based Malware Detection Solutions

Gheorghe Hăjmăşan[1,2][0000−0001−8664−8956], Alexandra Mondoc[1,3][0000−0003−2096−3771], Radu Portase[1,2][0000−0001−9008−1462], and Octavian Creţ[2][0000−0002−6657−634X]

[1] Bitdefender, Cluj-Napoca, Romania, {amondoc, rportase}@bitdefender.com
[2] Technical University of Cluj-Napoca, Cluj-Napoca, Romania, {Gheorghe.Hajmasan, Octavian.Cret}@cs.utcluj.ro
[3] Babeş-Bolyai University, Cluj-Napoca, Romania

Abstract. The constant evolution of malware, both in number and complexity, represents a severe threat to individual users and organizations. This is increasing the need for more advanced security solutions, such as dynamic behavior-based malware detection, that monitor and analyze actions performed on a system in real time. However, this approach comes with an intuitive downfall, the performance overhead. For this issue we propose two solutions 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 impact and also a lower false positive rate.

1 Introduction

The need for computer security is now greater than ever with thousands of new malware being released every hour [1]. In order to fight them and offer a reasonable security solution we have to continuously innovate and use more than traditional security technologies. The answer is to use proactive solutions and considering this, dynamic detection technologies are the right choice.

Almost always, more proactive methods mean better security but also a greater performance impact. Dynamic detection is certainly no exception. Monitoring the behavior of processes dynamically, at run-time, by installing filters (such as file system, registry, process and APIs filters) clearly 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 these aspects, it is very important to find ways of improving the performance of behavior based malware detection solutions. We decided to find new ways to limit the performance impact, but at the same time to

keep the detection rate at the same level, which is a significant challenge. We propose two key methods which 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.

The next section presents related work in the area of improving the performance of security solutions. Sect. 3 describes our findings and the proposed solutions. Sect. 4 presents the obtained results, including the known limitations of the implementation and the last section contains the conclusions.

2 Related Work

Behavior-based malware detection solutions focus on analyzing the actions performed by processes in order to identify malicious applications. An approach used by some solutions [6, 18] entails the use of a controlled environment to run the analyzed sample, while monitoring and recording its actions. Because malware may not exhibit their behavior when allowed to run only for a limited time, such solutions should monitor the sample's execution until its termination, which could result in a very long analysis time.

Solutions that are deployed on the end-host represent another approach. Kolbitsch et. al. [11] propose extracting information during the analysis of samples in a controlled environment and later using it for real-time protection. The solution is based on creating fine-grained models representing the behavior of malware, using system calls. These models are then matched against the runtime behavior of applications executing on an end host.

Ji et. al. [10] compare multiple host-based detection solutions to determine potential causes of performance overhead on an end-host. For the analyzed solutions, the major causes of performance overhead were identified as: the mechanism used to intercept the actions performed by processes, which system calls were intercepted and how many, and finally the mechanism used to correlate the intercepted actions in order to detect malware. The overhead caused by some security products is also presented in [16]. Espinoza et. al. [7] also analyze the performance overhead caused by antivirus solutions.

One method of reducing the performance impact of a dynamic malware detection solution running on end hosts is to limit the number of monitored applications. A solution designed to detect bots [14] excludes processes that interact with the user from being monitored, based on the assumption that botnets do not exhibit such behavior. This way the user experience may be improved and the performance impact on the end host may be reduced. However the mechanism used is limited to botnets and cannot be applied to malware in general.

The number of monitored processes may also be reduced by employing an application reputation mechanism. An example of static reputation mechanism, described in [12], indicates for which processes a less strict monitoring protocol should be used, compared to the other unknown processes. The system presented in [13] also helps improve performance, by excluding applications with high rep-

utation from further scans or behavioral profiling. Some of the limitations of this solution are that the reputation is associated to a file, and the reputation score is not updated dynamically based on the behavior of the process.

A reputation system can be integrated with both static and dynamic detection mechanisms and can help improve their detection rates by providing additional information about a file and reduce false positives on popular applications which have a high reputation. The effectiveness of using a reputation system that contributes to malware detection in the real world was confirmed by [13], which employs a system that determines the reputation of executable files and stores it in the Cloud. The solution blocks files with bad reputation from being downloaded. This approach is efficient since most malware reach a system via the Internet. However, because it relies on the Cloud to store the reputation data, this approach is less suitable for an offline environment or an enterprise network.

File reputation is used for malware detection in Polonium [4]. The reputation score for an unknown file is computed based on the reputation of the machines it was found on. The reputation of a machine is in its turn adjusted based on the reputation of all the files on that machine. Another solution based on file reputation is AESOP [15]. In contrast to Polonium, this solution establishes the reputation of a file based on the reputation of related files. These two solutions are effective in identifying the reputation of prevalent files, but find it more difficult to establish the reputation of files that are either infrequent or have been released recently and are new. In our approach the reputation of each application, even if it is less common, is earned in time. In addition, for both Polonium and AESOP, the reputation of files is established in a centralized manner, on the provider's infrastructure and is queried through the Cloud. This approach is not always suitable for isolated endpoints (e.g. critical infrastructure).

A method similar to our approach to parallelize the workload of a security solution is [17]. This solution focuses on reducing the CPU overhead of static signature matching in ClamAV [5]. It uses the high parallelization capacity of GPUs to prefilter signatures that would match on an analyzed sample. Only the signatures that could potentially match on a sample are then matched on the CPU. This way, other tasks are allowed to run on the CPU during the preliminary matching phase and the total analysis time is reduced.

3 Research Description

For our research, we modified the security solutions presented in [8] and [9]. These solutions monitor the actions performed by processes through a set of *filters* and analyze those actions using *behavioral heuristics*. Targeted malicious actions identified by the heuristics are correlated to detect malicious processes.

To illustrate the functioning of a monitoring system, Fig. 1 shows an example of execution flow for a process executing in an instance of the Windows Operating System. The solid arrows represent the execution flow in the absence of filters, while the dotted arrows represent modifications to the execution flow due to

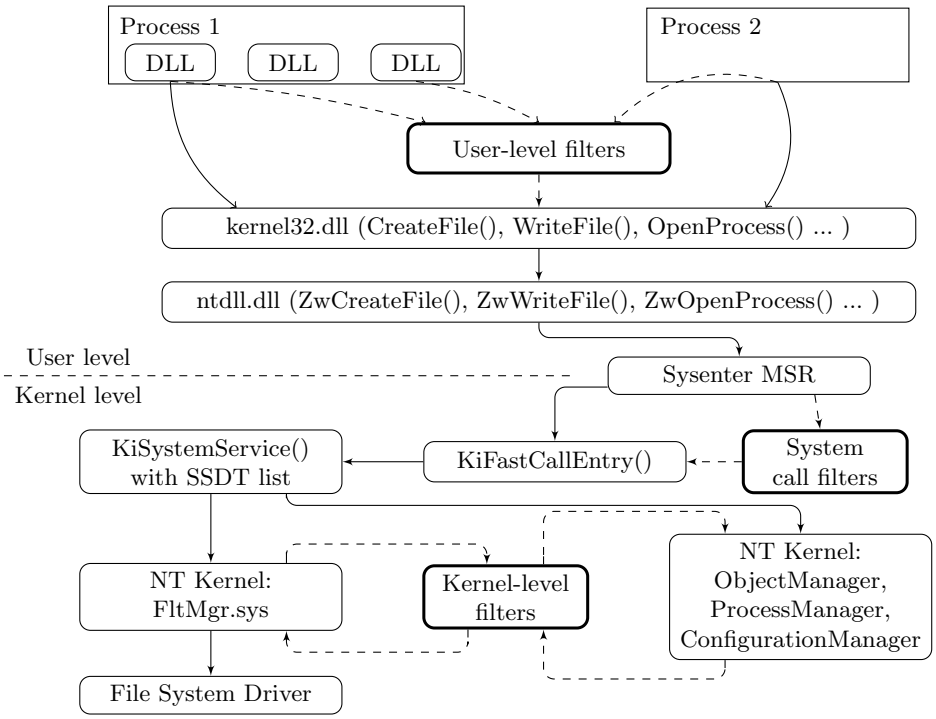

Fig. 1. Monitoring actions in an instance of the Windows Operating System

the presence of filters. The process may load multiple dynamic-link libraries (DLLs), that provide certain functionalities, for example writing to disk files, editing registry keys or launching new processes, through user-mode APIs. Any calls made to the monitored APIs are intercepted and analyzed.

To intercept Win32 API calls, API interception (hooking) through a DLL injection into the monitored process is used [3]. An example of hooking method, depicted in Fig. 1 consists of altering the starting point of a function to redirect the execution to another function, containing the monitoring functionality. The normal execution flow is resumed after the monitoring function is terminated.

Some functionalities provided by the Operating System (OS) can also be accessed by issuing a system call. Such system calls can be intercepted by modifying the associated handler routine [3], for example, by changing the value stored in a model-specific register (MSR) of the processor. This will redirect the execution to the system call filters, as illustrated in Fig. 1.

Certain operations, like actions targeting the file system, registry and the creation or termination of processes, can be intercepted in the Operating System kernel, through a set of filters provided by the OS. In Fig. 1, they are referred to generically as *Kernel-level filters*.

The security solutions presented in [8, 9] rely on **behavioral heuristics**. A heuristic is an algorithm that analyzes certain events to determine whether the occurrence of a set of events within the client system indicates a security threat. Heuristics represent procedures that are notified by the filters whenever an event that they registered for occurs. Usually, the information processed by the heuristics has four main sources: filtering of Win32 APIs, the file system, registry or process filters. The information gathered from one or more of these sources is processed within the heuristic to decide whether a targeted action is being performed.

Some of the actions that can be identified by the heuristics are: creating a copy of the original file, hiding a file, injecting code into another process and creating a startup registry key such that the malicious application will be executed after an operating system restart.

3.1 Main Performance Issues

A behavioral detection solution has many components that may cause performance overhead, such as: hooks, filters, heuristics, communication and evaluation. The components responsible with data acquisition (hooks, filters) have a small performance impact, however there is little room for improvement. Data processing performed by the behavioral heuristics is the most resource intensive and can be improved the most.

3.2 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 multi-processor 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 Fig. 2. The first step indicates that the execution of a monitored application is suspended (hijacked), by a filter installed by the security solution, which can be either a hook, a registry, file system or process filter. The filtered event is processed and if it is not of interest for the security solution, the execution of the suspended application will resume. If the event is of interest, the framework generates an event structure that is passed as a parameter to the synchronous heuristic callbacks. As Fig. 2 illustrates, some synchronous heuristics are still used, because they handle critical events that must be processed as quickly as possible, and an asynchronous approach does not necessarily guarantee that. However, such heuristics should be few and lightweight. For example, some may only generate an event that can be later processed by other asynchronous heuristics.

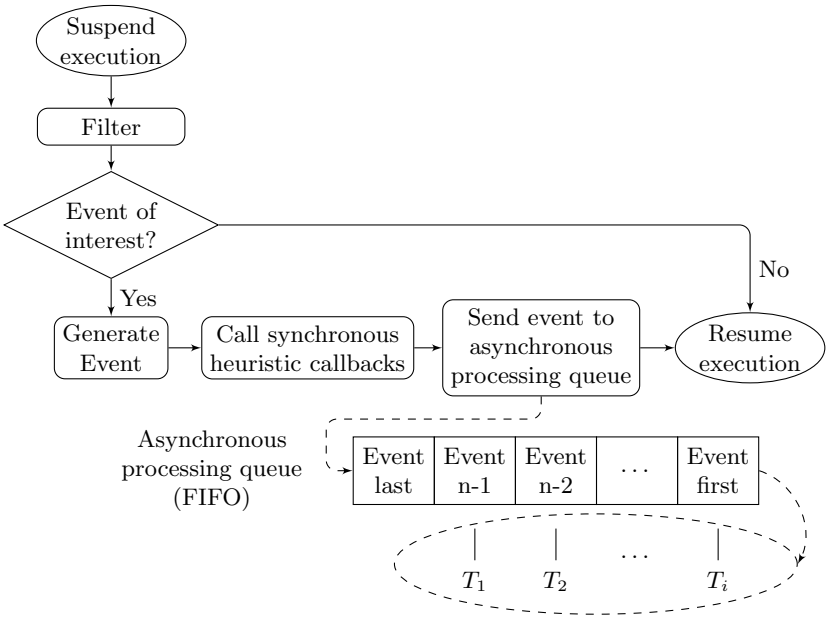

Fig. 2. Asynchronous Processing

After calling the synchronous callbacks (if there are any), the framework will enqueue the event into an asynchronous FIFO processing queue and immediately resume the suspended execution. Essentially, the filters act as producers and the worker threads (denoted T1 ... Ti) as the consumers. The number of worker threads depends on the hardware specifications of the system. This number may also be adjusted dynamically, depending on the number of unprocessed events in the queue. 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.

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). And 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.

Reputation Databases. 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. A summary of the reputation system is illustrated in Fig. 3.

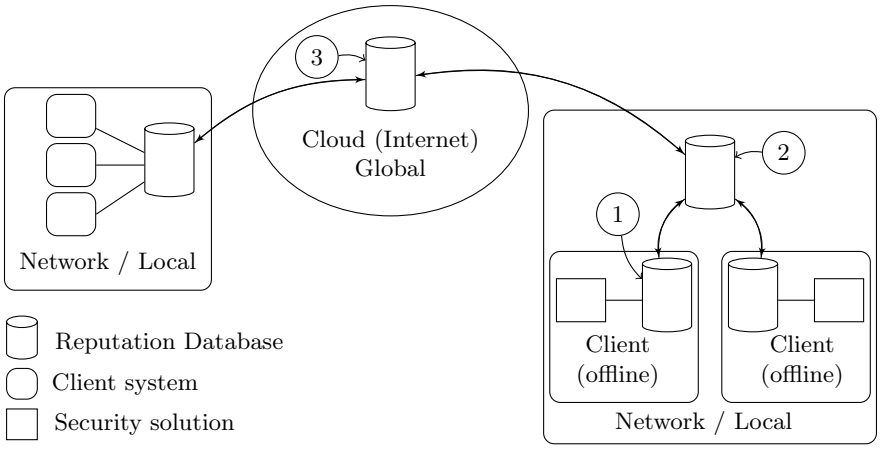

Fig. 3. Dynamic Reputation Servers

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 Fingerprint of a Process. In the case of static reputation there are simple methods to identify a file, for example by computing a hash on the file (i.e. SHA, MD5). In the case of dynamic reputation we must identify a process,

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.

Fig. 4 provides some schematic examples of fingerprints and emphasizes that some fingerprints can be supersets for others, and some can be subsets of a larger fingerprint (both illustrated as dotted rectangles). An important aspect is that a superset will inherit the bad reputation of a subset. For example if the smallest fingerprint in the figure (containing only *A.exe* and *X.dll*) has a reputation (untrusted score) of 60%, all the other supersets will have at least 60% untrusted score. In other words, a superset has the maximum untrusted score of all its subsets.

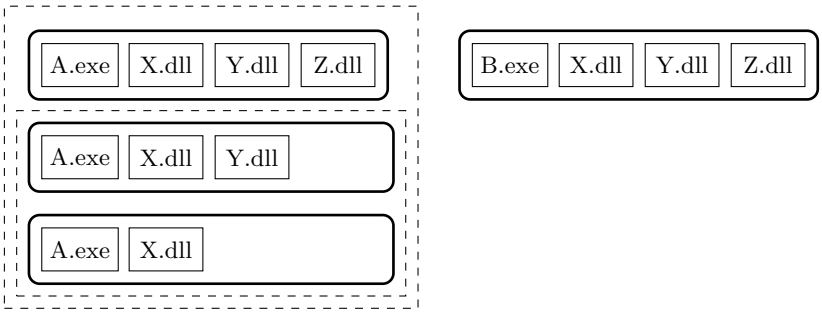

Fig. 4. Process Fingerprints

Levels of Events and Heuristics. Each level of reputation (untrusted score) has an associated level of monitoring (events and heuristics). An example of this association is presented in the following enumeration:

- 100% (untrusted) - Fully monitored
- 90% - Disable a few events and heuristics with high performance impact
- ...
- 20% - Monitor for code injection and drop/copy file
- 10% - Only monitor for code injection events and heuristics
- 0% - Not monitored at all

If an application has a fingerprint with the untrusted score of 60% it will be monitored by all the events and heuristics corresponding to the level 60% and below. All the events and heuristics in the upper levels will be skipped for that process (events will not be considered of interest - as illustrated in Fig. 2 and the callbacks of heuristics will not be called).

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.

Reputation of a Process. When a new process is started, its fingerprint is computed and one or more of the available databases will be queried for the associated untrusted score. If the fingerprint is new (e.g. was not found in any queried database), the score will be set to a default score, according to some rules. For example, if the process does not have any special attributes, the associated score will be 100%. If it has a digital signature, or is executed from a certain path (e.g. *Program Files*), its untrusted score may be lower: 80%. This may be the case for the reputation of the process presented in Fig. 5.

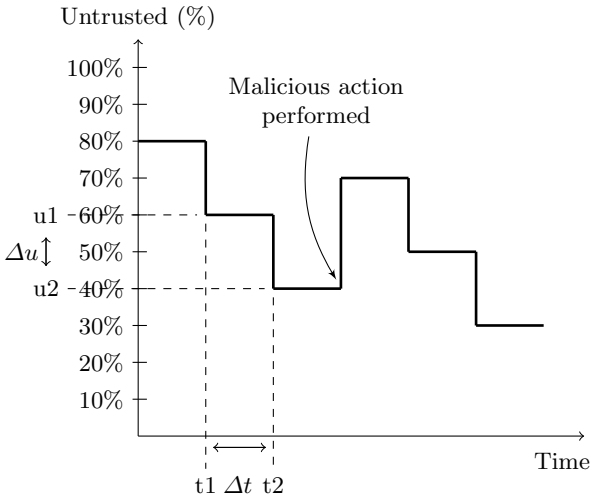

Fig. 5. Reputation of a Process

If the fingerprint is not new, it has one or more associated scores obtained from the databases. The initial score will be the minimum between those scores. 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. The untrusted score may be decreased in time to 0%.

If the process performs a malicious action, the untrusted score increases with a preset amount associated to the action. In addition, that process will be monitored more severely. When a process has performed a malicious action and its untrusted score was raised, all the available databases are notified about this

incident and will adjust the score to be the maximum score (between the new and the old ones).

If the current reputation of a process has improved, the databases (servers) will be notified, but they will not necessarily upgrade the score for the fingerprint right away. They may instead wait a certain amount of time with no incidents to pass for that fingerprint before adjusting the reputation, or will wait for other processes with the same fingerprint to report the same good behavior (similar to a voting process).

3.4 Further Performance Improvements

The previously described improvements have a major and global impact on the security solution. Additionally, we made improvements at a smaller scale. Using Windows Software Trace Preprocessor (WPP) and Windows Performance Toolkit (Xperf) for performance logging we were able to pinpoint other performance issues and solve them in a *traditional* way: costly checks were placed after the ones that are simpler and faster, functions that were called very frequently were transformed into inline functions or the information they provided was cached, slower algorithms were replaced with more suitable ones (e.g binary search tree over list) a.s.o.

4 Results

The solution was tested for functionality, stability and detection on real and virtual machines running 32 and 64 bit versions of Windows 7, 8.1 and 10.

For the performance tests we used a Dell OptiPlex 7040 physical machine equiped with an Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz processor with 8 GB of RAM and a 250 GB SSD drive.

4.1 Detection Tests

Table 1 presents the detection rate of the security solution without any optimizations (our previous work) in the third column. Adding asynchronous heuristics and dynamic reputation produced no effect to the detection or false positives rate. For the detection tests we used the same methodology as in [8] and [9].

Table 1. Detection results for solution with and without improvements

	Total	Detected	%	Detected with improvements	%
Malware set	2283	2255	98.773	2255	98.773
Clean set	680	14	2.058	14	2.058

Malware samples and clean applications were run in virtual machines for a limited time after which we collected the results. The tested malware were proprietary to Bitdefender and were collected from various sources like: honeypots, spam email attachments, URLs used to spread malware, client submissions a.s.o.

4.2 Performance Tests

In order to obtain relevant results we attempted to replicate the industry standard testing methodologies of AV-TEST [2]. We focused our attention on testing the performance impact of our solution on every day user experience. For this we tested copying files from a local disk to another local disk, reading both small and large files with common office tools such as Microsoft Office Word, Powerpoint and Adobe PDF reader, installing wide-spread popular applications (7-Zip, Adobe Reader, Gimp, Skype, Tunderbird, VLC player and Libre Office) and opening files with these applications.

Every test run started with applying a fresh Windows image on the testing machine using a preconfigured golden image containing Windows 10 x64 version 1511 build 10.0.10586 with Windows Updates and Windows Search Indexing Service both disabled. We installed the solution on the clean machine and performed a reboot. Then we tested a single action 12 times and collected the results. The best result and the worst result were discarded and the remaining results were averaged.

For each tested action we performed the corresponding test on the following machine configurations: a clean version of the OS (without our solution installed), the solution without any performance improvements, the solution using asynchronous heuristics (without the dynamic reputation), the solution using only the dynamic reputation (no asychronous heuristics, applications monitored on a level corresponding to real-world trust), the solution using dynamic reputation with all tested applications set to 10% untrusted, the fully optimized solution with asynchronous heuristics and dynamic reputation corresponding to real-world trust and finally the solution using asynchronous heuristics and dynamic reputation with all tested applications set to 10% untrusted.

Table 2 presents the results of the performance tests. The rows in the table correspond to the tested operations and the columns represent the configurations of the solution. Each data cell of the table represents a percentage difference between the average execution time of the solution configuration corresponding to the column and the baseline execution time for the system without the solution. The results are also represented as a bar chart in Fig. 6.

For all of the tests performed both the dynamic reputation and the asynchronous heuristics reduced the overall performance impact of the security solution. As we expected, the best scores were obtained using asynchronous heuristics and dynamic reputation set to 10%. This configuration however has some obvious detection rate issues and is tested only to illustrate an ideal case when all the clean application have only 10% untrusted score. The solution using asynchronous heuristics and real-world dynamic reputation was very close to this ideal solution on all tests while keeping the same detection rate.

For **copy** scenarios, the biggest improvement was obtained by using asynchronous heuristics. We observed an unexpected 2.1% difference between the solution with dynamic reputation set to real world and dynamic reputation set to 10% untrusted. We believe this was caused by the high disk usage involved in the copy test. For **install**, dynamic reputation alone had very little impact, most

Table 2. Performance Impact Results

	S	S+A	S+DR	S+DR[10%]	S+A+DR	S+A+DR[10%]
Copy	35.224	8.808	9.257	11.308	4.501	2.850
Execute	12.372	0.474	0.921	0.979	1.251	0.121
Install	7.517	2.885	6.320	4.725	2.005	1.677
Open	6.463	1.413	0.847	0.719	1.290	1.097

S = security solution, A = asynchronous heuristics, DR = dynamic reputation, DR[10%] = DR set to 10% untrusted for all processes

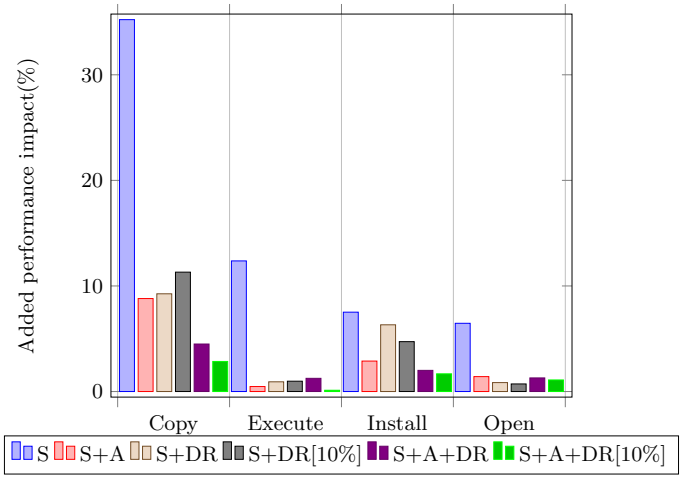

Fig. 6. Performance Impact Results

of the improvements were caused by asynchronous heuristics. Adding dynamic reputation to asynchronous heuristics improved the results slightly.

The **execute** test was affected almost identically by both improvements when used separately. Asynchronous heuristics have a slightly better score. With both improvements used together, the solution actually performed worse than with just one of the optimizations, but not by much. The extra code needed to use both the optimizations was the probable cause of the performance loss. The **open** test results were similar to the **execute** results, with dynamic reputation offering a very small advantage over asynchronous heuristics. The solution with both improvements also performed slightly worse than with each of the improvements.

In some cases, the solution with dynamic reputation performed worse than the solution with asynchronous heuristics only, but this is expected due to fingerprint computation and reputation querying. However, the difference is very small and still better than the solution without any improvements. Even considering this small drop in performance, the reason why the dynamic reputation remains very important is that besides improving the performance it also reduces the probability of false positives.

4.3 Limitations of the Solution

The proposed solution may have certain limitations. The first approach, the use of asynchronous heuristics does not improve performance for single core systems. In addition, the dynamic reputation database needs researchers to supervise the system, and also to add new rules for increasing or decreasing the reputation, according to the latest malware samples. Furthermore, the performance tests have presented a quick view of the improvements made by the proposed solution, but they can not cover all the real world scenarios or user use cases.

There may be malware samples that take advantage of the dynamic reputation system and do not manifest any malicious behavior until their untrusted score is very low and are no longer monitored by the behavioral detection solution. However, the delay of the malicious behavior represents an advantage for the security product which gains time to analyze and detect the sample with other components such as anti-malware signatures. Essentially, through this delay the malware loses the advantage of *zero-day* (unknown) malware, the main target of behavior based detection solutions.

5 Conclusions

We emphasized the need for proactive solutions to combat the latest malware attacks and we consider that dynamic real-time behavioral detection solutions are effective against these advanced threats. However, this approach comes with a significant downfall: the performance overhead. Fixing this critical issue is a progress that opens the possibility for many other improvements in this field.

Our research lists the most costly components in terms of performance overhead of an existing detection solution and focuses on those that can be improved. We determined that the biggest impact is caused by the behavioral heuristics and proposed two different solutions that solve the performance issue globally, at framework level, without having to alter the heuristics. The two proposed solutions are elegant, being practical, logical, straightforward to implement and to use in a commercial solution as well as easy to maintain (the dynamic reputation is self-adjustable according to the defined rules).

The first proposed solution consists of executing most of the behavioral heuristics asynchronously, taking advantage of the advances in hardware (multi-core systems). This solution uses a FIFO queue for the events that need to be processed asynchronously and a thread pool for executing the heuristics that process those events asynchronously.

The second proposed solution is a dynamic behavioral reputation database. Our work describes how a fingerprint of a process can be computed, how the *untrust* score can be updated and stored in reputation databases, how the servers can be distributed and how this dynamic reputation indicator is used to reduce the performance impact and the false positive rate using monitoring levels.

The results of the tests that evaluate these solutions show a considerable performance improvement for each approach individually and also for the combination of both approaches, without any impact on the malware detection rate.

References

1. AV-Test: Malware statistics, http://www.av-test.org/en/statistics/malware/
2. AV-Test: Performance testing, https://www.av-test.org/en/test-procedures/test-modules/performance/
3. Blunden, B.: The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System. Jones and Bartlett Publishers, Inc., USA (2009)
4. Chau, D.H., Nachenberg, C., Wilhelm, J., Wright, A., Faloutsos, C.: Polonium: Tera-scale graph mining and inference for malware detection. In: SIAM INTERNATIONAL CONFERENCE ON DATA MINING (SDM). pp. 131–142 (2011)
5. ClamAV: https://www.clamav.net
6. Elhadi, A.A.E., Maarof, M.A., Barry, B.I.: Improving the detection of malware behaviour using simplified data dependent api call graph. International Journal of Security and Its Applications 7(5), 29–42 (2013)
7. Espinoza, A.M., Al-Saleh, M.I., Crandall, J.R.: Antivirus performance characterisation: system-wide view. IET Information Security 7 (06 2013)
8. Hăjmăşan, G., Mondoc, A., Creţ, O.: Dynamic behavior evaluation for malware detection. In: 2017 5th International Symposium on Digital Forensic and Security (ISDFS). pp. 1–6 (April 2017)
9. Hăjmăşan, G., Mondoc, A., Portase, R., Creţ, O.: Evasive Malware Detection Using Groups of Processes, pp. 32–45. Springer International Publishing, Cham (2017)
10. Ji, Y., Li, Q., He, Y., Guo, D.: Overhead analysis and evaluation of approaches to host-based bot detection. International Journal of Distributed Sensor Networks 11(5), 524627 (2015)
11. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: Proceedings of the 18th Conference on USENIX Security Symposium. pp. 351–366. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)
12. Mircescu, D.: Systems and methods for using a reputation indicator to facilitate malware scanning (august 2015), https://www.google.com/patents/US9117077, US Patent 9,117,077
13. Ramzan, Z., Seshadri, V., Nachenberg, C.: Reputation-based security an analysis of real world effectiveness. Symantec Security Response, https://www.symantec.com/content/dam/symantec/docs/white-papers/reputation-based-security-en.pdf
14. Shin, S., Xu, Z., Gu, G.: Effort: A new host-network cooperated framework for efficient and effective bot malware detection. Comput. Netw. 57(13) (2013)
15. Tamersoy, A., Roundy, K., Chau, D.H.: Guilt by association: Large scale malware detection by mining file-relation graphs. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1524–1533. KDD '14, ACM, New York, NY, USA (2014)
16. Uluski, D., Moffie, M., Kaeli, D.: Characterizing antivirus workload execution 33, 90–98 (03 2005)
17. Vasiliadis, G., Ioannidis, S.: GrAVity: A Massively Parallel Antivirus Engine, pp. 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
18. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 116–127. CCS '07, ACM, New York, NY, USA (2007)