*Journal of Universal Computer Science, vol. 25, no. 9 (2019), 1089-1108*
*submitted: 12/1/19, accepted: 9/7/19, appeared: 28/9/19* © *J.UCS*

# Precise Performance Characterization of Antivirus on the File System Operations

**Mohammed I. Al-Saleh**
(Jordan University of Science and Technology
Department of Computer Science
P.O. Box 3030
Irbid, Jordan 22110
misaleh@just.edu.jo)

**Hanan M. Hamdan**
(Jordan University of Science and Technology
Department of Computer Science
P.O. Box 3030
Irbid, Jordan 22110
hmhamdan12@cit.just.edu.jo)

**Abstract:** The Antivirus (AV) is of an important concern to the end-users community. Mainly, the AV achieves security by scanning data against its database of virus signatures. In addition, the AV tries to reach a pleasant balance between security and usability. When to scan data is an important design decision an AV has to make. Because AVs are equipped with on-access scanners that scan files when necessary, we want to have a fine-grained approach that provides us with high precision explanation of the performance impact of the AVs on different file system operations. Microsofts minifilter driver technology helps us achieve exactly what we want. By deploying a minifilter driver, we show that most overhead of the tested AVs are greatly imposed on the OPEN operation. Interestingly, we also show that the AV greatly enhances the timing for the READ operation. Finally, the WRITE and CLEANUP operations show almost no differences in terms of performance.
**Key Words:** Antivirus, Performance, File system, Minifilter driver
**Category:** D.4.3, D.4.6, D.4.8

## 1 Introduction

Computer security is being integrated as part of the business model of companies. Companies store customers private data and credentials in their machines. Consequently, they have to protect their machines against penetration. In addition, many online stores have to find ways to assure secure transactions. Any mistake or vulnerability in the system leads to money loss and, more importantly, reputation damage. To counter attacks, several security controls have been applied. Such controls try to prevent, deter, detect, and recover from attacks. In many cases, data encryption comes with the solution. Data confidentiality and integrity can be greatly obtained through different provably-strong encryption techniques. However, not all security problems can be solved through encryption. For example, data has to be in plain before it is encrypted or after it is decrypted. An intruder might find a way to such plain data. Firewalls, intrusion detection systems, intrusion prevention systems, anti-spyware, and anti-adware are examples of security technologies to fight malware and attacks. One of the widespread security tools is the Antivirus (AV). According to a study [Richardson, 2011], the AV is being used by 94% of the surveyed parties. This indicates how mature, effective, and affordable the AV is. Basically, the AV keeps signatures for the viruses it already knows and stores them in its database. The AV scans data of interest against its virus signatures. If a match is found, then an extra action is required to stop and recover from the attack.

Most AVs are being developed by profitable, competitive companies that do not share their protection techniques with the security researchers and practitioners. There are two ways to study such AVs, namely reverse engineering and black-box testing. Researchers test the AV from either security or performance perspectives. However, most AV performance studies are technically superficial. They mainly discuss the overhead of the AV on systems without reasoning. Deeper studies on the performance impact of the AV on systems [Al-Saleh et al., 2013, Uluski et al., 2005] report the overhead of the AV and try to explain such overhead by the extra events that happen during experiments. Such events could be hardware events (such as cache hits and memory accesses), or software events (such as system calls). Going beyond this to give a higher-level explanation about such overhead has not been approached yet.

At the heart of the AV is the file system protection against reading or writing infected files. In order to provide such protection, the AV needs to intercept the file system operations and inspect the involved data. This study measures the performance impact of the AV on the main file system operations. To get a precise measurement, we have to get as close as possible to the AV scanning components and the file system operations. Modern AVs use Microsoft Windows minifilter driver technology to gain control over file system operations. Our approach to measure the AV performance impact is also through utilizing the technology of the minifilter drivers. In this study, we implemented a minifilter that is stacked on top of the AVs minifilter to measure the performance impact of the AV. We intercepted the file system operations using the same way the AV itself does and reported the impact of the AV.

This paper first discusses related work. Then, we give background on minifilter device drivers in Section 3. Section 4 illustrates methodology and experimental setup. Our results are shown in Section 5. Then, discussion and future work are covered in Section 6. This is followed by the conclusion.

## 2 Related work

Unfortunately, the topic of measuring the impact of the AV software on the underlying systems has not received enough attention from the research community. Currently, popular AVs are managed by commercial vendors who understandably do not expose the structures of their products. Such profit-driven products missed intellectual thoughts and suggestions from security research communities. Researchers speculate or reverse-engineer AVs in order to understand what exactly they do [Post and Kagan, 1998]. Black-box testing is another option for researchers to reason about AVs. Generally, AVs are being studied from two perspectives, namely performance and security.

A vital concern for both AV vendors and users is the performance impact of AVs on the systems they protect. The security vs. usability argument always comes into the fore. Users undoubtedly are not willing to live with performance-killing AVs. Consequently, AV vendors work hard to attain a good compromise between the achievable security and the performance impact. Performance studies either test how much overhead the AVs incur on systems or try to suggest methods that can make AVs more efficient.

Al-Saleh et. al. [Al-Saleh et al., 2013] study the overhead of AVs on systems. Several experiments which included common user activities such as Internet browsing and document editing were designed to check how these activities can be affected by AVs from performance perspective. An overhead is obviously shown on all of the activities. Then, the authors tried to reason about this overhead by studying the events generated by the Operating System (OS). The events include system calls, page faults, Input/Output operations, and process/thread creations. Event Tracing for Windows (ETW) was utilized to log events. In case of an AV, these events outnumbered that of no installed AV. They finally showed that processes spend more time in event waiting queues in case there exists an AV. This suggests that processes wait more for events because of the AV. This study is the closest to ours. However, it only explains the overhead by the very general system symptoms (events) that were indirectly tied to the AV. In this study, we try to work as closely as possible to the AV. Moreover, we want to tie the overhead to the AV more accurately by sandwiching the kernel components of the AV. This position gives us direct and precise analysis of where overhead is being spent. Both of this work and [Al-Saleh et al., 2013] are complementary and concentrate on a different angle of the problem. While in [Al-Saleh et al., 2013] the overhead is attributed to the AV due to various system operations without going deeper to explain further, this work, however, attributes the overhead to specific file system operations. To put both results together, we can say that the events that the authors in [Al-Saleh et al., 2013] were talking about take place inside the file system operations that are being discussed in this paper.

A hardware-level performance characterization study [Uluski et al., 2005] showed that AVs incur extra CPU cycles, machine instructions, cache misses, and memory accesses. These extra events can be attributed to the AV. Both of [Al-Saleh et al., 2013] and [Uluski et al., 2005] are complementary.

In [Al-Saleh et al., 2013] the authors focus on the OS view of events that might cause performance overhead. These events include file system operations, processes and threads creations, various system calls, and page faults. While in [Uluski et al., 2005], the authors focus on the hardware kind of events, such as number of CPU cycles, cache hits/misses rates, and memory accesses.

Al-Saleh et al. [Al-Saleh et al., 2015] tested the trade-off of security and performance of AVs. AVs were put under excessively concurrent attacks, where many malware samples were read/written simultaneously. Interestingly, the study shows that malware samples can evade the detection of some AVs.

Because pattern matching is at the heart of most AVs, enhancing the scanning process is essential. Several studies propose to enhance the AV performance [Hellal and Romdhane, 2016, Vasiliadis and Ioannidis, 2010, Jang et al., 2016].

Graphics Processing Units have been utilized to enhance the performance of the AV detection [Cheng, 2010, Post and Kagan, 1998]. Thousands of threads can be used in the scanning process. In [Vasiliadis and Ioannidis, 2010], ClamAV, a popular open-source AV, is modified to utilize the GPU and enhance the performance. Also, in [Lin et al., 2011] ClamAVs Wu-Manber and Aho-Corasick pattern matching algorithms are modified to achieve a better performance.

AV security research concerns about enhancing malware detection by either finding vulnerabilities or proposing effective detection techniques. An on-access scanner that is capable of detecting malware written to disk by modifying the open-source AV, ClamAV [Kojm, 2004] ,is developed [Miretskiy et al., 2004]. Extracting virus signatures out of an AV has been proven to be a real threat [Christodorescu and Jha, 2004]. A semantic-aware malware detection algorithm was developed [Christodorescu et al., 2005] to consider metamorphic viruses. They solved the problem by integrating the semantics of instructions to detect malicious instances.

Creating timing channel attacks against AVs to check how updated they are is another attack dimension [Al-Saleh and Crandall, 2011]. Enhancing the AV capabilities to scan network data was also developed [Al-Saleh and Shebaro, 2016]. In [Bayer et al., 2009], the researchers studied the behavior of malware by collecting one million malware instances. Then, they analyze them in order to help improve the AV products. Finally, certain criteria that can help users assess AV products were studied [Josse, 2006]. Finally, the impact of the AV on digital evidence has been tackled [Al-Saleh, 2013].

## 3 Background: minifilter device drivers

A file system filter driver is a kernel-mode module that is capable of inspecting and (possibly) modifying the operations of the file system. The Windows I/O manager passes the I/O request to the filter manager, which in turn passes it to the file system filter driver before sending it to the file system drivers for completion. Windows filter drivers are currently implemented as minifilter drivers. Minifilter drivers are managed by the filter manager. The filter manager is positioned on top of the file system drivers so it can intercept file system operations before the file system driver does (see Figure 1). Each minifilter driver has an altitude value that determines its position in the stack of the minifilter drivers; the higher the altitude, the higher in the stack. Consequently, the minifilter with a higher altitude can intercept I/O operations before others in the stack. In Microsoft Windows, the altitude values can be reserved and managed into groups, each with a specific purpose.

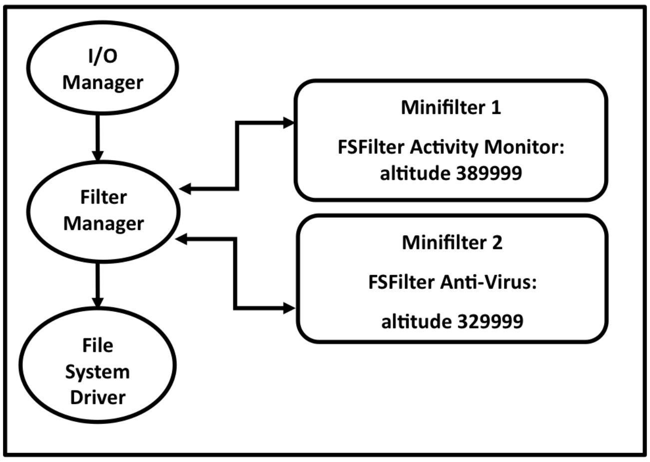



**Figure 1:** Relationship between the different I/O components.

According to their functionalities, Microsoft Windows organizes the altitude values into what is called load order group. A range of altitude values is reserved for each group. Table 1 shows some of load order groups, short descriptions of them, and their altitude range of values.

As in other types of Windows device drivers, a minifilter driver must have *DriverEntry()* routine, which is called when the minifilter driver is loaded. In this routine, the minifilter driver registers the file system operations (in which it is interested) with the filter manager by calling *FltRegisterFilter()* routine. Filter-

| **Load order group** | **Altitude range** | **Purpose** |
|---|---|---|
| FSFilter Activity Monitor | 360000-389999 | Observing and reporting file I/O operations. |
| FSFilter Anti-Virus | 320000-329999 | Detecting viruses. |
| FSFilter Content Screener | 260000-269999 | Preventing certain contents or specific files creation. |
| FSFilter Encryption | 140000-149999 | Encrypting and decrypting data. |

**Table 1:** Some load order groups as described by Microsoft.

ing takes place after calling *FltStartFiltering()* routine. If *DriverEntry()* routine works normally, then it returns SUCCESS as a status value. A minifilter driver selects which I/O operations to filter by registering PRE and POST callback routines. Only one PRE callback routine can be registered for an I/O operation. The same applies for the POST callback routine. Upon an I/O operation, the filter manager passes the I/O operation to the top minifilter driver (i.e., with the highest altitude), where the PRE callback routine of the minifilter driver for that operation will be called. After the PRE callback routine is returned, the filter manager takes the control again. The next PRE callback routine of the next minifilter driver (if exists) on the stack is called, and so on until all PRE callback routines of all minifilter drivers are called. After that, the filter manager passes the I/O operation to the file system driver to process it. When the file system driver completes the I/O operation, the filter manager takes the control again and passes the completed I/O operation through the POST callback routines of the minifilter drivers from the lowest altitude in the minifilter stack up to the highest. After finishing each POST callback routine, the filter manager takes control and calls the next POST callback routine, and so on until all POST callback routines are called. Figure 2 explains the flow of this process.

A PRE callback routine has three return cases. If the PRE callback routine returns a status of *FLT_PREOP_SUCCESS_WITH_CALLBACK*, then this tells the filter manager to call the POST callback routine for this operation. Similarly, if it returns *FLT_PREOP_SUCCESS_NO_CALLBACK* value, then this tells the filter manager not to call the POST callback routine for this operation. Finally, the third case is to return *FLT_PREOP_SYNCHRONIZE* status. This tells the filter manager to call the POST callback routine of this operation but in the same context of the PRE callback routine. After a POST callback routine is done, the status of *FLT_POSTOP_FINISHED_PROCESSING* is returned as an indication of a completed job.

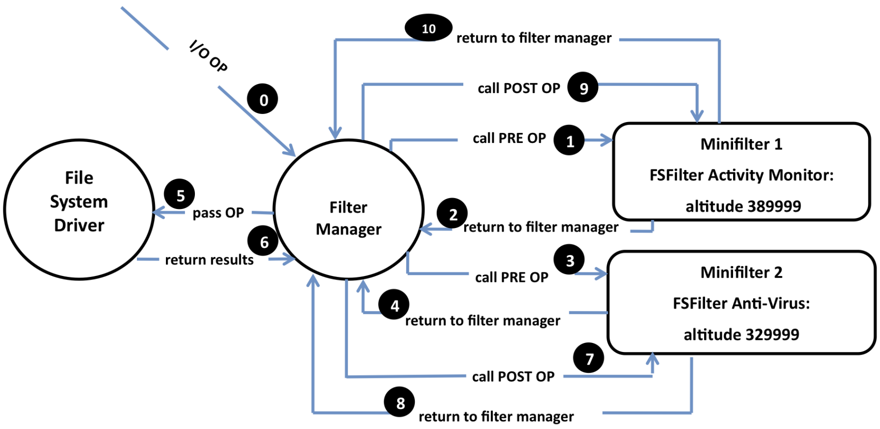



Figure 2: I/O request flow between the filter manager, minifilter drivers, and file system drivers.

## 4 Methodology

Basically, this paper tries to answer the following question: **How much overhead do the kernel components of AVs impose on the file system operations? Measuring the performance overhead of AVs by getting as closely as possible to such components is our goal.**

Consequently, we design the following experiment to answer the above question.

### 4.1 Experiment: Measuring the AV performance overhead on the file system operations

#### 4.1.1 Our approach

We want a way that measures the performance impact of the AV on the main file system operations, namely, CREATE, READ,WRITE and CLEANUP, at the kernel-level. Deeper investigation shows that such AVs have special device drivers, called minifilter drivers (see Section 3). In order to precisely measure such overhead, we need to get as closely as possible to the minifilter device driver of an AV. This can be perfectly implemented using the minifilter device driver technology itself for our purposes. As previously explained, minifilter drivers can be stacked on top of each other according to their altitude values. In this study, we build a minifilter driver with a higher altitude than that of the AV in order to intercept the I/O request before the AV does and after the AV complete its scan using the PRE and POST operations, respectively. This enables us to compute the time that is being spent on the file system operations. Figure 3 illustrates our approach to measure the AV performance overhead imposed on the file system operations.

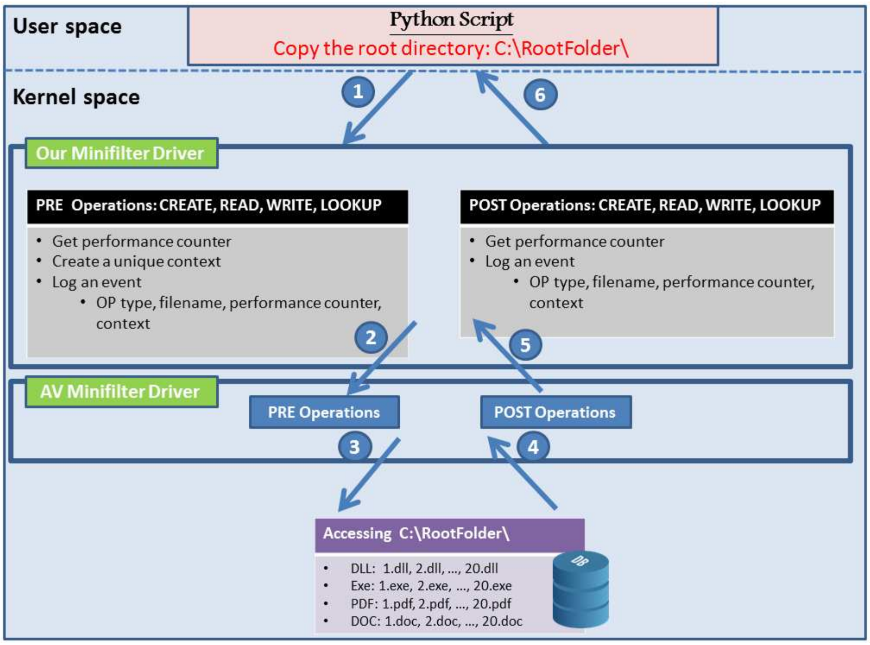



Figure 3: Our approach for measuring the AV performance overhead on the file system operations.

There are many types of I/O Request Packets (IRP) that can be intercepted by minifilter drivers. However, we are interested in four of them, which are: IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MJ_CLEANUP. An IRP_MJ_CREATE request is sent by the I/O manager upon file/directory creation as a result of calling one of the following routines: *CreateFile()* (user mode) *IoCreateFile()*, *IoCreateFileSpecifyDeviceObjectHint()*, *ZwCreateFile()*, or *ZwOpenFile()*(kernel mode).An IRP_MJ_READ request is sent when reading from files through *ReadFile()* routine (user mode) or *ZwReadFile()* routine (kernel mode). In addition, an IRP_MJ_WRITE is sent when writing to files through *WriteFile()* routine (user mode) or *ZwWriteFile()* routine(kernel mode). Finally, an IRP_MJ_CLEANUP is sent when all handles of a file object are closed by calling *CloseHandle()* routine (user mode) or *ZwClose()* routine(kernel mode).

All operations of interest are required to be registered as callbacks in the minifilter driver. Consequently, we register PreCreate and PostCreate callbacks for the following operations: IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MJ_CLEANUP.

In our minifilter driver, we use *KeQueryPerformanceCounter()* kernel routine, which returns a high resolution performance counter along with its frequency that can be optionally set in the routine's parameter. This performance counter is updated based on the frequency that is measured in ticks/sec. The frequency resolution of this counter is less than a micro second. The frequency is set at system boot and then the performance counter is incrementally updated according to the frequency. Time intervals can first be computed using this routine in ticks and then converted into time by dividing a time interval (in ticks) over the frequency (ticks/sec). We use this routine in PRE and POST filter operations upon several file system operations. We get the value of the performance counter in PRE operation. Then, we pass through the operation to the next minifilter on the stack (probably the AV). Afterwards, we gain the control again in our POST operation, after the AV passes the operation of interest to the upper minifilter drivers, where our minifilter sits. We take the value of the performance counter again and compute the elapsed time. This elapsed time can be used to check the overhead of an AV installed on the system. One important issue here is how we can distinguish operations from each other. For example, more than read operation can be requested on the same file simultaneously. Consequently, two or more PRE filter operations might be hit for the same file at very close times. The same applies to the POST filter operations. The problem here is the need to exactly pair each PRE operation with its exact POST peer, or otherwise the results will be erroneous. To solve this problem, we observe that each filter operation can be accompanied with a context which can be created when needed. The filter manager passes the context through from a PRE filter operation and returns it back to its POST counterpart, where things can be matched up. So, we create a context in each PRE operation and store in that context a unique value (we used the performance counter itself) which we should receive again in the POST operation to complete matching.

All performance counter readings are printed out to the default kernel log buffer that can be viewed using the WinDbg system program. Logging into a buffer storage is much faster that logging into an external file. We use the *DbgPrint()* kernel function for logging. The following information is logged at every PRE or POST opertion: the name of the file that is being manipulated, the type of operation, and the current performance counter value. After the experiment is completed, we copy all such information from WinDbg into an output file that is then populated into a database to be analyzed with a query system.

#### 4.1.2 Experiment design

To test the effectiveness of our approach explained, we designed the following experiment. We went through the following steps:

- We prepared six identical machines with the following specifications:
  1. OS: Microsoft Windows 7 Professional edition with Service Pack 1.
  2. RAM: 4GB.
  3. Hard Drive: 500GB.
  4. CPU: Core i7-860 at 2.80GHZ.
- On each machine, we copied a directory (called RootFolder) into the C drive of each machine. The directory contains four subdirectories, each of which contains 20 files of the same type, but with sizes ranging from 1KBto 10MB. The file types we used are DLL, EXE, DOC and PDF. So, we have 20 DLL files (named 1.dll, 2.dll, etc.) in one subdirectory, 20 EXE files (named 1.exe, 2.exe, etc.) in another, and so forth. These files are collected from the Internet in an ad hoc manner.
- We copied our minifilter device driver we designed into each machine. We gave our minifilter an altitude value equals to 389999, which is greater than all of that of the AVs to make sure that ours is stacked on top of the AV's minifilter driver. We tested this using the fltmc command that can be executed on a command line with a privileged account.
- Python 2.7 is installed on all machines.
- We copied a python script that programmatically copies our root directory into another. This step should trigger all of the file system operations we are interested in (CREATE, READ, WRITE, CLEANUP).
- We installed five well-known AVs on five of the six prepared machines. We left the remaining machine with NO-AV so it will be considered as a reference for later comparisons. More information about the AVs we installed are shown in Table 2. Every AV is updated with its latest updates regarding its signatures and software.
- All machines are restarted to obtain a fresh start point.
- An image of every machine is separately taken to enable us consistently repeat the experiment from the very same state.
- Now, every machine is ready for the experiment, which proceeds as follows:

1. Our minifilter is loaded and started.
2. The python script that copies the root directory is executed.
3. Enough time is given before stopping the minifilter to make sure that all logging statements pass through.
4. After stopping the minifilter, our log file is copied into an external storage and named appropriately (run1, run2, ).
5. Steps 1 through 4 are repeated for ten times on each machine.

– By now, each machine produces 10 logs for 10 different runs. We parse such logs and save them into an SQLite database.

– We design different SQL queries to collect the time spent on every operation type (CREATE, READ, WRITE, CLEANUP) on a file type (DLL, EXE, DOC, PDF) for all files (1, 2, 3, ..., 20) on a different machine for different runs (run1, run2, run3, ..., run10). For example, we query the database for the average of all CREATE operations on all files of EXE type on all 10 runs for the NO-AV machine. The returned value forms the base for our comparison to the cases where we have an installed AV. For example, to get the overhead of an AV (call it AV1) on the CREATE operation of EXE file types, we compare the average of the CREATE operation of AV1 on EXE files and compare it to that of NO-AV case. The difference is the overhead incurred by AV1 on CREATE operations of EXE files. The same argument applies for all other operations, file types, and other AVs.

## 5 Results

In this section, we present our results for the experiment that is explained in Section 4. Our aim is to check how much performance overhead the AV incurs on the main file system operations.

Microsoft's minifilter driver technique enables us to achieve our aim. File system operations can be wrapped in a PRE and POST operations that are called before an operation takes place and right after the operation, respectively. It is not a surprise that the AV itself utilizes such capabilities and implements its own minifilter drivers. As previously shown, minifilter drivers can be stacked on top of each other based on their altitude values. Given that, we gave our minifilter driver the max possible altitude value to make sure that it is going to be stacked on top of that of the AV. Because we record a high-precision time resolution at the PRE and POST operations, we are able to compute the elapsed time of the main file system operations. This elapsed time includes the normal

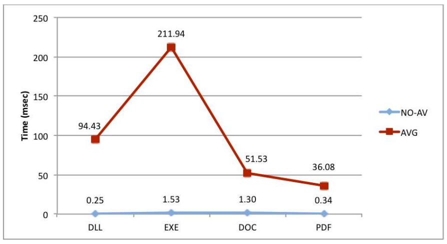
| AV name | Version | Filter altitude |
|---|---|---|
| AVG | 2015.0.6030 | 325000 |
| AVIRA | 15.0.11.574 | 320500 |
| COMODO | 8.2.0.4591 | 321200 |
| KASPERSKY | 15.0.1.415 (c) | 320400 |
| SYMANTEC | 22.0.0.110 | 365100, 260600, 329000 |

Table 2: The AVs we experimented with in this paper. Also, detailed information about them along with minifilter device driver information for each AV. An AV might have more than one minifilter driver. We obtained altitudes using *fltmc* command-line.

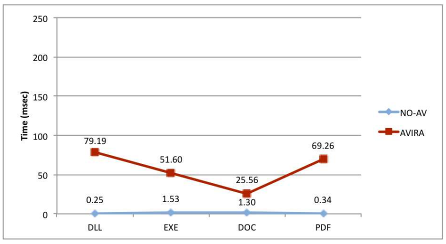
time that is spent on the file system operation itself and the overhead (if there is any) the AV imposes on the operation. In the discussions below, we always compare and highlight the difference between the NO-AV case and the AV case. Because in each file type (DLL, EXE, DOC, PDF) we have 20 files and we repeat each experiment 10 times. What we report is the average time of operations of interest that take place over 20 files for 10 runs. For example, we compute the average time for all CREATE operations on all EXE files over 10 runs. We do the same thing for the other file types.

Figure 4 shows the average times for the NO-AV case and different AVs for the CREATE (IRP_MJ_CREATE) operation on different file types. The increase in the average CREATE time in all file types is obvious in the case of the AV. Even though the CREATE (an open request is mapped to a create request by the OS) caller only wants a handle to the target file, the AV is expected to scan the target file in advance before successfully returning a handle to the CREATE caller. Consequently, we expect that the AVs take extra time in reading and scanning target files from the Hard Drive. One important highlight here is the synchronous nature of the CREATE operation versus the asynchronous nature of the WRITE operation. That is, the caller has to block until the whole operation returns. However, with the existence of the AV, the operation does not return until the AV is satisfied with the contents, which, in turn, imposes the extra time. Different AVs have different times on different file types.

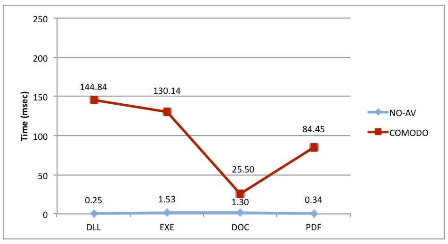
Figure 5 shows the average times for the NO-AV case and different AVs for the READ (IRP_MJ_READ) operation on different file types. Interestingly, in all cases, the average timing for the AV case is less than that of the NO-AV case. To



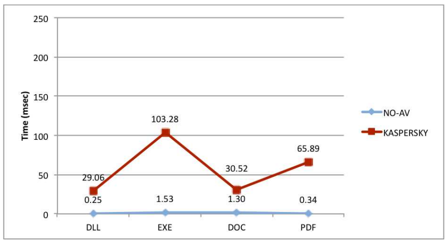

(a) AVG case



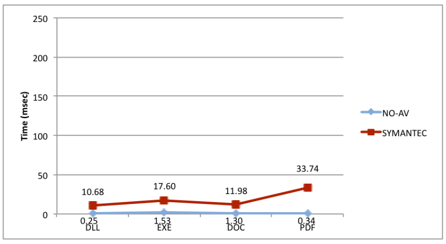

(b) AVIRA case




(c) COMODO case



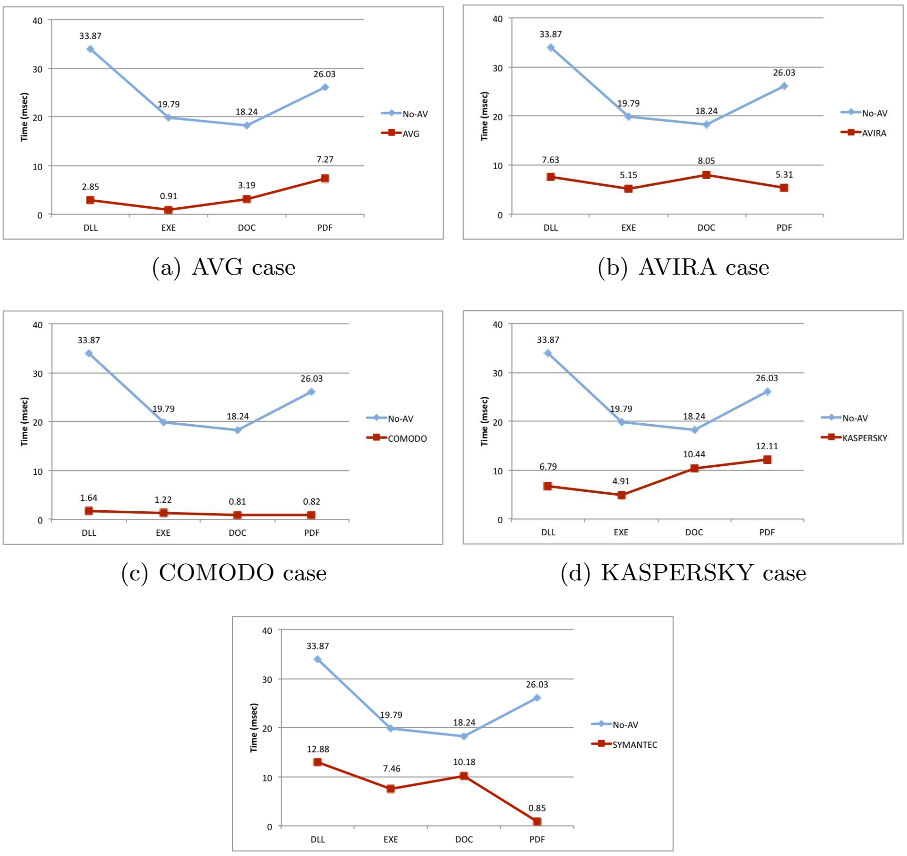

(d) KASPERSKY case




(e) SYMANTEC case

**Figure 4:** Average CREATE times for the AV cases and NO-AV case.

some extent, this confirms our conclusions for the CREATE operation. Because the AV reads data in advance for scanning, afterwards such data will be available in RAM, where it can be read from there so fast. For example, in Figure 4(a), the READ operation for the DLL files are averaged to 2.85 milliseconds in the case of AVG AV, while it is averaged to 33.87 milliseconds in the case of NO-AV, making a difference of 31.02 milliseconds. However, if we compare the difference between the overhead imposed by the AVG on the CREATE operation for DLL files (94.43 milliseconds as in Figure 5) and the CREATE operation for the NO-AV case (0.25 millisecond as in Figure 5), we can see the overhead on the CREATE operation is much more than the enhancement on the READ operation. Moreover, we also found out that the number of READ operations on the tested files is significantly more than that of the NO-AV case. For example, AVG AV makes an average of 1172 READ operations on the EXE files. On the other hand, that number is only 70 in case of NO-AV. The other AVs have similar behavior. For example, Symantec AV makes an average of 1551 READ operations on the PDF files. However, that number is only 68 in case of NO-AV.




(a) AVG case

(b) AVIRA case

(c) COMODO case

(d) KASPERSKY case

(e) SYMANTEC case

**Figure 5:** Average READ times for the AV cases and NO-AV case.

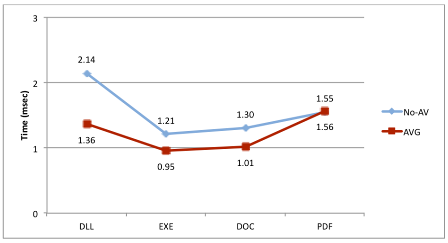
Figure 6 shows the average times for the NO-AV case and different AVs for WRITE (IRP_MJ_WRITE) operation on different file types. We have two important observations. First, the WRITE operation is generally very fast whether with or without an AV (less than 3 milliseconds in all cases). This is because the WRITE operation is asynchronous operation. This means that the operation returns right after the request is grabbed by the Operating System (OS), not until the data is written to the Hard Drive. The OS usually postpones writing data to the Hard Drive for performance issues. Second, in case of AVG, COMODO and SYMANTEC (Figures 6(a), 6(c), and 6(e), respectively), it seems that the WRITE operation takes slightly less time than that of the NO-AV case. Even though the timing difference is not significant, we speculate that this difference is due to the caching issues applied by the OS on data being used more frequently. In other words, the Cache Manager and the Memory Manager might observe different usage patterns in case of the AV, which, in turn, is reflected on how readily the data to be written is. In case of AVIRA and KASPERSKY (Figures 6(b) and 6(d), respectively), there is no common pattern to conclude a difference.

Figure 7 shows the average times for the NO-AV case and different AVs for CLEANUP (IRP_MJ_CLEANUP) operation on different file types. It is obvious that the CLEANUP operation is called when the number of handles on a file object has reached zero (i.e., the file is being closed). This might be a reason for the AV to scan files upon being closed to check if the target file has malicious contents. Our results show that, in some cases, there is no difference between an AV case and the NO-AV case. In others, there is a slight difference. Our conclusions about the CLEANUP operation are very similar to those we already made for the WRITE operation. That is, the CLEANUP operation is very fast operation and data to be scanned upon CLEANUP has already been scanned when being read.

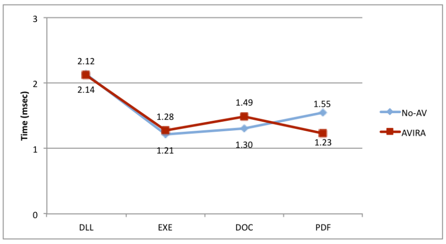
**More Insights into the Results:** Several parameters could interfere with our results. For example, popped up, concurrent operations might affect the timing measurements. To countermeasure such impact, we repeat the experiment 10 times and we report the average over these runs. This way we can eradicate any anomalous readings. Furthermore, AVs might be more aggressive to certain file types than they are to others. For example, EXE and DLL files are more dangerous than DOC and TXT file types. This is obvious in both AVG and AVIRA antiviruses for the CREATE operation in Figures 4. Finally, because we CREATE operation is considered the most effective operation on the performance, Symantec AV performs the best among the studied AVs.

## 6 Discussion and future work

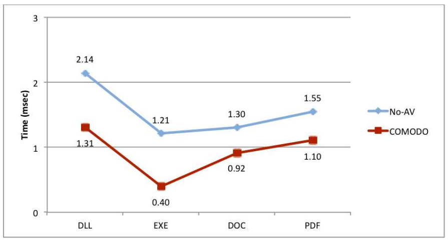
To get an accurate performance impact of an AV on the main file system operations, we need to be as close as possible to the scanning components of the AV. However, in theory, doing so does not completely assure us that an AV is not triggering scanning activities that might not take place in the same sequential



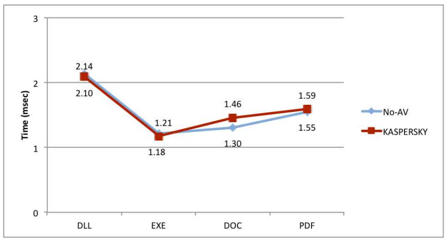

(a) AVG case



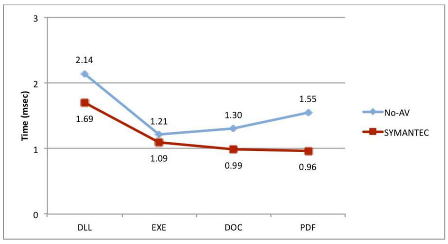

(b) AVIRA case




(c) COMODO case




(d) KASPERSKY case

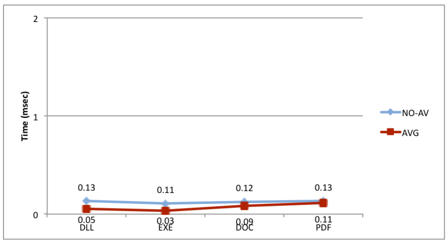



(e) SYMANTEC case

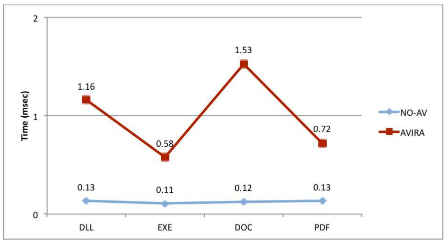
**Figure 6:** Average WRITE times for the AV cases and NO-AV case.

line on which we start and end our timing measurements. For example, in its minifilter driver, an AV might be sending a signal to some other AV components to do some scanning and let the current operation proceeds. This theoretical scenario does not affect the accuracy of our approach because we only compute the elapsed times for file system operations. In other words, we do not compute all execution times of AVs; we just compute the time spent on file system operations.

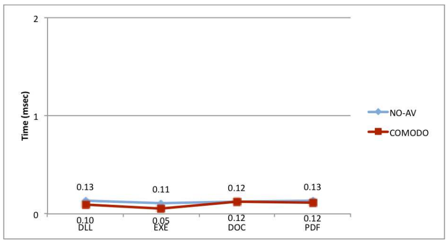
In this paper, we want to highlight the AV activities on normal systems (i.e., those that are not compromised). Consequently, all the files we experimented with are benign ones. Experimenting with malicious files, on the other hand, does not provide much information. Extra actions will be triggered by the AV



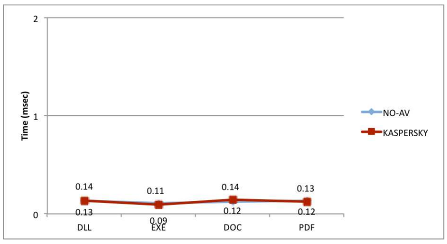

(a) AVG case



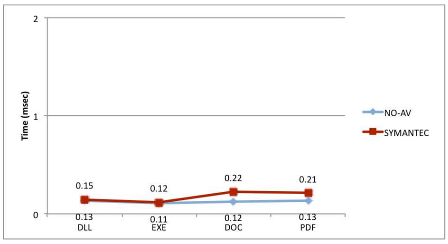

(b) AVIRA case




(c) COMODO case




(d) KASPERSKY case




(e) SYMANTEC case

**Figure 7:** Average CLEANUP times for the AV cases and NO-AV case.

to prevent any harm on the system and complete the clean up process. The amount of time that AV takes to complete such a job is meaningless to the end user from two perspectives. First, end users deal with malicious files very occasionally compared with benign files. Second, it is well expected that the AV will take much more time in case it finds malicious contents.

Another issue is that an AV might issue new file system operations that do not go through our minifilter driver. Microsoft Windows has a set of routines that can be called from a minifilter driver such as *FltCreateFile()*, *FltReadFile()*, and *FltWriteFile()*. These routines can only be intercepted by minifilter drivers with altitude values less than that of the calling minifilter driver. Even with that, the extra times these FltX() routines are adding will be included in our calculations. In our experiments, we measured IRP_MJ_CLEANUP operation but not IRP_MJ_CLOSE one. This is because, in practice, the IRP_MJ_CLOSE request might be delayed for a long time (hours or even days) after the last handle on a file is removed. This might have to do with the Memory and Cache Managers. However, accounting for IRP_MJ_CLEANUP request is good enough for our purposes. Furthermore, our results for both WRITE and CLEANUP operations show that no significant difference is observed. As explained, this is mainly because of the fact that the data to be written is the same data that has been previously read (and probably scanned). Thus, there is no need to scan it again. Designing another experiment to elaborate on this issue is a future work.

In this paper, we only concentrate on measuring the performance overhead of AVs on the file system operations. AVs leverage various approaches to gain scanning capabilities. In the case of file systems, modern AVs implant minifilter device drivers to gain control and thus they impose performance overhead on the critical path of the file system operations. Consequently, this overhead can be measured pretty accurately. Nonetheless, AVs also have performance impact on memory, network, and CPU operations. For example, modern AVs also utilize various behavior-based or heuristic-based scanning techniques to recognize malicious activities. In these cases, however, an AV might be doing its computation as a separate process that is running in parallel with the monitored process (*i.e.,* not in the critical path of the monitored process.), making the performance measurement a little more complicated. In their AV performance measurement work [Al-Saleh et al., 2013], the authors show that the performance impact of AVs can occur from the fact that AVs enforce processes to create more memory page faults, wait more in the system queues, or make more system calls. That empirical study, however, does not explain precisely how these are incurred. This paper is complementary to that one and tries to get as low as possible to where exactly AVs are imposing the overhead. Exploring the other dimensions of this issue is a future work.

## 7 Conclusion

Testing different aspects of AVs is important in order to understand their internal functionalities and enhance their performance. This paper measures the performance impact of AVs on the main file system operations in a novel way. Modern AVs implement minifilter drivers to intercept file system operations in order to scan contents before an operation proceeds. To measure the performance impact precisely, we use minifilter drivers in the same way the AVs do to take control before and after the AVs do so. This paper shows that most overhead is imposed on the CREATE operation, where AVs usually scan data and prevent opening malicious files. Furthermore, as a side effect of reading the contents of the files in the CREATE operation in advance for scanning, the READ operation takes less time in the presence of AVs because such data will already be in RAM when the READ operation is triggered. Finally, regarding the WRITE and CLEANUP operations, we show that they are fast and that no difference is intrinsically observed with existence of AVs. This is because we request to write data that is obtained from the same files which the AVs have already scanned. This suggests that the AVs apply clever ways not to scan the same data twice.

## References

[Al-Saleh, 2013] Al-Saleh, M. I. (2013). The impact of the antivirus on the digital evidence. *International Journal of Electronic Security and Digital Forensics*, 5(3-4):229–240.

[Al-Saleh et al., 2015] Al-Saleh, M. I., AbuHjeela, F. M., and Al-Sharif, Z. A. (2015). Investigating the detection capabilities of antiviruses under concurrent attacks. *International Journal of Information Security*, 14(4):387–396.

[Al-Saleh and Crandall, 2011] Al-Saleh, M. I. and Crandall, J. R. (2011). Application-level reconnaissance: Timing channel attacks against antivirus software. In *LEET*.

[Al-Saleh et al., 2013] Al-Saleh, M. I., Espinoza, A. M., and Crandall, J. R. (2013). Antivirus performance characterisation: system-wide view. *IET Information Security*, 7(2):126–133.

[Al-Saleh and Shebaro, 2016] Al-Saleh, M. I. and Shebaro, B. (2016). Enhancing malware detection: clients deserve more protection. *International Journal of Electronic Security and Digital Forensics*, 8(1):1–16.

[Bayer et al., 2009] Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. (2009). A view on current malware behaviors. In *LEET*.

[Cheng, 2010] Cheng, Y. W. (2010). Fast virus signature matching based on the high performance computing of gpu. In *Communication Software and Networks, 2010. ICCSN'10. Second International Conference on*, pages 513–515. IEEE.

[Christodorescu and Jha, 2004] Christodorescu, M. and Jha, S. (2004). Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44.

[Christodorescu et al., 2005] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. (2005). Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, Oakland, CA, USA.

[Hellal and Romdhane, 2016] Hellal, A. and Romdhane, L. B. (2016). Minimal contrast frequent pattern mining for malware detection. *Computers & Security*, 62:19–32.

[Jang et al., 2016] Jang, J.-w., Kang, H., Woo, J., Mohaisen, A., and Kim, H. K. (2016). Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. *computers & security*, 58:125–138.

[Josse, 2006] Josse, S. (2006). How to assess the effectiveness of your anti-virus? *Journal in Computer Virology*, 2(1):51–65.

[Kojm, 2004] Kojm, T. (2004). Clamav. *URL http://www. clamav. net*.

[Lin et al., 2011] Lin, P.-C., Lin, Y.-D., and Lai, Y.-C. (2011). A hybrid algorithm of backward hashing and automaton tracking for virus scanning. *IEEE transactions on computers*, 60(4):594–601.

[Miretskiy et al., 2004] Miretskiy, Y., Das, A., Wright, C. P., and Zadok, E. (2004). Avfs: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium (Security 2004*, pages 73–88. USENIX Association.

[Post and Kagan, 1998] Post, G. and Kagan, A. (1998). The use and effectiveness of anti-virus software. *Computers & Security*, 17(7):589–599.

[Richardson, 2011] Richardson, R. (2011). 15th annual 2010/2011 computer crime and security survey. *Computer Security Institute*, pages 1–44.

[Uluski et al., 2005] Uluski, D., Moffie, M., and Kaeli, D. (2005). Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33:90–98.

[Vasiliadis and Ioannidis, 2010] Vasiliadis, G. and Ioannidis, S. (2010). Gravity: a massively parallel antivirus engine. pages 79–96.