Antivirus Performance Characterization: System-Wide View

Mohammed I. Al-Saleh
University of New Mexico
Department of Computer Science
Mail stop: MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131
alsaleh@cs.unm.edu

Jedidiah R. Crandall
University of New Mexico
Department of Computer Science
Mail stop: MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131
crandall@cs.unm.edu

Abstract

Cyber security threats are still big concerns of the cyber world. Even though many defense techniques have been proposed and used so far, the antivirus (AV) software is very widely used and recommended for the end-users-PC community. Most effective AV products are commercial and thus competitive and it is not obvious for security researchers or system developers how exactly the AV works or how it affects the whole system. The AV adds layers of complications over the already layered, complicated systems. Because there is very little effort in the literature to provide a way for understanding the AV functionality and its performance impact, in this paper we want to shed some light on that direction.

To the best of our knowledge, we are the first to present an OS-aware approach to analyse and reason about the AV performance impact. Our results show that the main reason of performance degradation the tasks have with the existence of the AV software is that they mainly spend the extra time waiting on events. Also, the AV in most of our experiments enforces the tasks to spend more time using the CPU. Although there is an overhead from the competition between the tasks and the AV on the CPU, this competition is not a main factor of the overall overhead. Because of the AV intrusiveness, the tasks in our experiments are caused to create more file IO operations, page faults, system calls, and threads.

1 Introduction

The Antivirus software (AV), even vulnerable for new attacks, is still widely used and recommended for that it can detect a wide range of known malware. The AV is claimed to detect some of the unknown malware through heuristic algorithms [11] it runs looking for malware-similar behaviors. According to two studies [2, 5], 80-81% of end-users have an AV installed on their machines. Also, Microsoft Windows keeps alerting a user who has no installed AV as if it is so dangerous not to have an AV.

The AV has an OS property for that it intercepts and inspects system-wide operations. It is not just a user-space process with a bunch of DLL (Dynamic Link Libraries) files linked to it; it is more complicated software to understand. Users care about buying machines with decent hardware hoping to accomplish their work fast. However, they are less aware of installing software that might render the new hardware unutilized. System designers care about designing efficient, reliable systems with no care of software that could render their system inefficient. Also, software developers and security administrators might be unaware of what the AV could cause to the debugging process or the intrusion detection system given that the AV might change processes' behaviors.

Most AVs are commercial and their scanning algorithms are not revealed to the public. Even though this strategy is good for the AV venders to compete, it is not good for security researchers who will not be able to assess, if even possible, the AV without suffering. There is no effort in the security research literature that specifically targets analyzing the commercial AV software.

The AV performance impact has not been well studied. Some studies [19, 6, 3] had conducted several experiments aiming to show the overhead (extra time or instructions) the AV adds while performing specific tasks. Even though these studies did well in characterizing the AV performance, the main question would still be what exactly causes this overhead! Because we can consider the AV as a property of the whole system, we need a system-wide inspection approach to characterize its performance. Although monitoring the system from the hardware-level [19] is useful and the hardware performance counters could be queried, the hardware view is limited in terms of information it can provide. A better approach is to have a system-wide, OS-aware instrumentation scheme that is able to provide information at different levels.

In this paper, we examine the performance issues caused by the AV from the OS point of view. Fortunately, we utilized an instrumentation tool to inspect the whole system; a Windows built-in technology, called Event Tracing for Windows (ETW). This technology is integrated with Microsoft Windows kernel to log events of interest very efficiently (when enabled). More details about this technology are coming in Section 2. We designed several experiments that represent common end-users tasks W/A the AV being installed to see how intrusive the AV is to these tasks and thus to pinpoint its performance impact. To enrich the study, we pursued two AVs, Symantec and Sophos.

This paper is organized as follows. First, we give a background on ETW in Section 2. This is followed by Section 3 that explains our experimental setup, and then our results in Section 4. A discussion and future work are in Section 5. Then related works and the conclusion follow.

2 Event Tracing for Windows (ETW)

Because the AV intercepts and inspects system-wide operations, we need a system-wide tool to be able to understand the AV behavior. Event Tracing for Windows (ETW) is a low-overhead, system-wide instrumentation technology that comes with Microsoft Windows starting from Windows 2000. ETW is integrated into the kernel so that it can capture most kinds of the OS events users are interested in such as process-related, CPU-related, IO-related, and memory-related events. ETW can be enabled/disabled on the fly without a need to restart the machine. It produces binary files with .etl extension that can be converted to CSV or XML formats using tools such as **tracerpt.exe**. In the XML format, the trace file consists of events, each starts with <Event> tag and ends with </Event> tag. Every event consists of header and body. The header contains fixed and general information about the event and is common to all events, while the body contains specific information based on the event type. **Listing 1** shows an XML representation for Process Start event. The header starts with <System> section, while the body starts with <EventData>.

ETW architecture consists of four components: controllers, providers, sessions, and consumers. See Figure 2.

1. **Controller :** its main job is to start/stop tracing.
2. **Provider :** it is the source of events. Whenever an event happens, it sends it out to one of the sessions.
3. **Session :** it manages buffers and logs events into trace files.
4. **Consumer :** it interprets the trace files produced by sessions.

xperf is a powerful tool that comes with the Windows Performance Tools, which can be used as a controller to start ETW. It also can be used as consumer of the log files. Figure 1 shows a graph captured from xperf as a consumer.

The ”Windows Kernel Trace” provider is responsible for sending the OS kernel events to the ”NT Kernel Logger” session which in turn logs the events into trace files. In this project, we used xperf to enable The ”Windows Kernel Trace” with many flags that represent all kinds of events.

```
<Event xmlns=”http://schemas.microsoft.com/win↩
     /2004/08/events/event”>
    <System>
        <Provider Guid=”{9e814aad−3204−11d2−9a82↩
             −006008a86939}” />
        <EventID>0</EventID>
        <Version>3</Version>
        <Level>0</Level>
        <Task>0</Task>
        <Opcode>1</Opcode>
        <Keywords>0x0</Keywords>
        <TimeCreated SystemTime=”2011−03−21T21↩
             :37:15.770988400Z” />
        <Correlation ActivityID↩
             =”{00000000−0000−0000−0000−000000000000}”↩
              />
        <Execution ProcessID=”2120” ThreadID=”6592”↩
              ProcessorID=”0” KernelTime=”0” ↩
             UserTime=”15” />
        <Channel />
        <Computer />
    </System>
    <EventData>
        <Data Name=”UniqueProcessKey”>0↩
             xFFFFFA800E071060</Data>
        <Data Name=”ProcessId”>0x77C</Data>
        <Data Name=”ParentId”>0x848</Data>
        <Data Name=”SessionId”>        1</Data>
        <Data Name=”ExitStatus”>259</Data>
        <Data Name=”DirectoryTableBase”>0x14E594000↩
             </Data>
        <Data Name=”UserSID”>\\alsaleh−hpdv6\↩
             alsaleh</Data>
        <Data Name=”ImageFileName”>calc.exe</Data>
        <Data Name=”CommandLine”>&quot;C:\Windows\↩
             system32\calc.exe&quot; </Data>
    </EventData>
    <RenderingInfo Culture=”en−US”>
        <Opcode>Start</Opcode>
        <Provider>MSNT_SystemTrace</Provider>
        <EventName xmlns=”http://schemas.microsoft.↩
             com/win/2004/08/events/trace”>Process↩
             </EventName>
    </RenderingInfo>
    <ExtendedTracingInfo xmlns=”http://schemas.↩
         microsoft.com/win/2004/08/events/trace”>
        <EventGuid>{3d6fa8d0−fe05−11d0−9dda−00↩
             c04fd7ba7c}</EventGuid>
    </ExtendedTracingInfo>
</Event>
```

Listing 1

3 Experimental setup

We designed our experiments to investigate and characterize the performance issues caused by AVs on specific

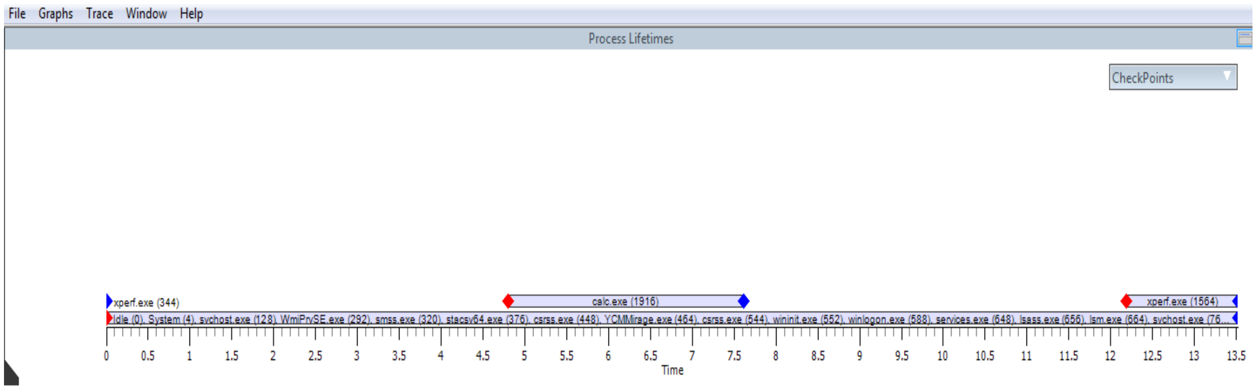
File Graphs Trace Window Help
Process Lifetimes
CheckPoints
xperf.exe (344)
calc.exe (1916)
xperf.exe (1564)
Idle (0), System (4), svchost.exe (128), WmiPrvSE.exe (292), smss.exe (320), stacsv64.exe (376), csrss.exe (448), YCMMirage.exe (464), csrss.exe (544), wininit.exe (552), winlogon.exe (588), services.exe (648), lsass.exe (656), lsm.exe (664), svchost.exe (76...
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10 10.5 11 11.5 12 12.5 13 13.5
Time

Figure 1: **xperf as a consumer.**

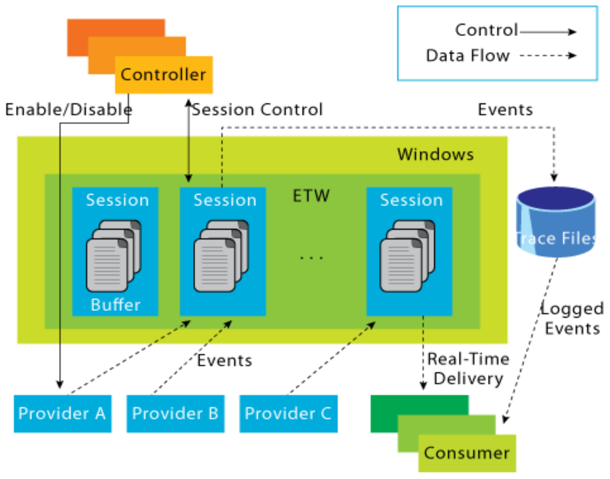

Figure 2: **ETW architecture, reproduced from [16].**

tasks from a system-wide view. We designed four experiments that represent common end-user tasks. All of our experiments were scripted in PowerShell to prevent any user intervention during experiments. We ran the experiments on a machine that has Windows 7 OS, Intel Dual Core Atom processor at 1.66 GHz, 4 GB RAM, and 250 GB of hard disk. The hard disk is partitioned into three partitions almost equally likely. The machine has triple boot on Windows 7. All partitions have the same exact software except that the second partition has Sophos AV installed, and the third partition has Symantec AV installed. Besides Windows 7, each partition has MS Office 2010, Windows SDK 7.1 (contains Windows Performance Tools 4.7), and Python. The Windows update service and the indexing services had been disabled to prevent accidental events from taking place in the middle of the experiments. All the experiments in the following section had been conducted on every partition separately and the machine has been rebooted after every experiment.

3.1 Experiments

1. **Client-Server**: a PowerShell script starts the ETW logging, and then it starts a Python client that connects to a server on another machine using sockets. Then the client receives a zipped file from the server that is password-protected. The zipped file has putty.exe, the popular SSH client. After receiving the file, putty.a, the script unzips the file using 7za.exe, an unzipping program. Once the unzipping is done, the logging is disabled and the log file is taken. This experiment involves file IO operations, system calls, CPU operations, and networking.

2. **Write to Microsoft Word and save**: a PowerShell script starts the ETW logging, creates a new Microsoft Word COM object, writes a short sentence to the object, saves and closes the object into word.doc file, and then stops logging. In this experiment we want to see how the AV interacts with such popular staff end-users frequently do. Memory and file IO operations are involved in this experiment.

3. **Copy from Microsoft Word to Microsoft PowerPoint**: a PowerShell script starts the ETW logging, creates a new Microsoft Word COM object, opens a pre-created word file (wordsourcedoc.doc) into the new object, copies its content into the clipboard (only has a short sentence), creates a new Microsoft PowerPoint COM object, creates a new presentation in the PowerPoint object, adds a new slide to the created presentation, pastes the copied sentence into the slide, saves the PowerPoint object, closes both objects, and stops logging. This experi-

ment shows a frequent act in which end users copy data from application into another. The question is how the AV interacts with this process that involves COM objects creations and data transfer between different applications.

4. **YouTube**: a PowerShell script starts the ETW logging and creates an Internet Explorer COM object and makes it navigate into a particular video in www.youtube.com. Then the script sleeps for five seconds, letting the browser to start the video and then it closes the browser and stops logging. This experiment involves using the Internet Explorer that is considered one of main gates usually threats come through. Also, the experiment involves running flash videos over the internet.

We convert all experiments' log files into a CSV files and dumped the data into PostgreSQL [4] database. Then we designed SQL queries to retrieve information we care about. We present our findings in the next Section.

4 Results

In this section we present the results for the experiments we explained their methodology in Section 3. The goal is to show as many as differences (in terms of OS point of view) between running an experiment with and without an installed AV. In all of our experiments we care about the processes and files directly involved in the experiment. The OS metrics we examine are: File IO operations, page faults, system calls, threads and processes creations, and CPU scheduling.

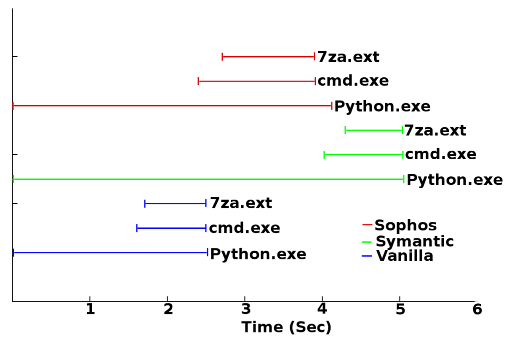

Figure 3: **Client-Server experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The processes we care about are: python.exe, cmd.exe, and 7za.exe.**

Figures 3, 4, 5, 6 show the lifetimes of processes we care about in all experiments. The overhead caused by Symantec and Sophos is obvious in all experiments.

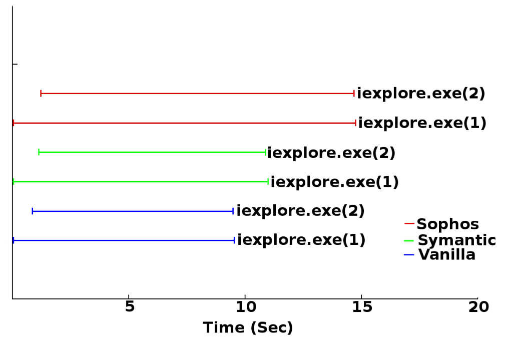

Figure 4: **YouTube experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The processes we care about are two processes of the same image: iexplore.exe.**

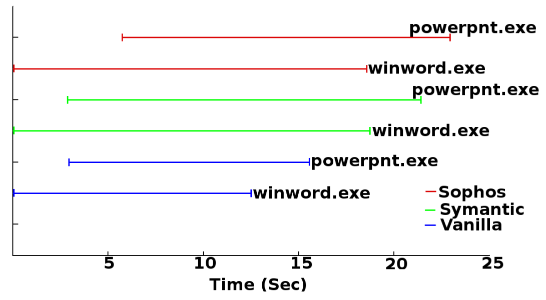

Figure 5: **Copy from Microsoft Word to Microsoft PowerPoint experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The processes we care about are: winword.exe and powerpnt.exe.**

Not only that, but also if a process start time depends on another process, the other process would delay the start time of its dependent.

After it gets executed, a process is in one of three states: executing, waiting in the ready queue to be picked up by the scheduler, or waiting for an event that, when happens, puts it back in the ready queue. Figure 7 shows the lifetime of the processes divided between the three states. What is clear here is that in both Symantec and Sophos the processes spend more time on waiting for events than in the Vanilla case. Also, what is surprising in the figure is that in Sophos case, the total wait time is about 17 seconds while the whole execution is about 4.1 seconds. We found that 7za.exe process has three threads; the wait time is accumulated for the all three, see figure 11. However, that total time for all 7za.exe threads is about 2.8 seconds out of the 17 seconds. We found that python.exe process creates five threads in case of Sophos, while it is only one thread in case of Vanilla and Symantec. Figure 12 shows the waiting times for the

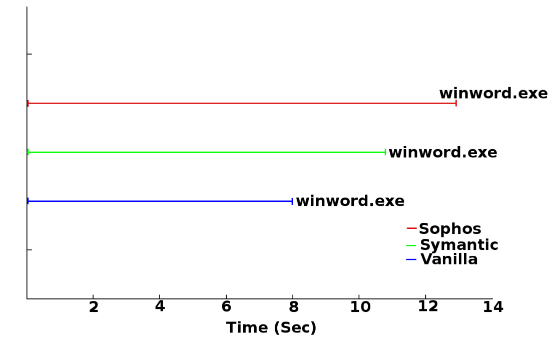

Figure 6: **Write to Microsoft Word experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The process we care about is winword.exe.**

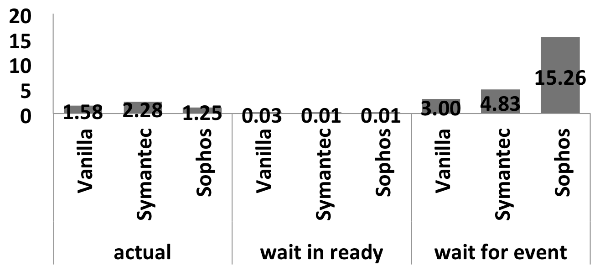

Figure 7: **Client-Server experiment: processes' time (in sec) distributed between execute (actual), ready (wait in ready), and wait (wait for event) states.**

python.exe threads in case of Sophos. We ran the experiment again and found out that the same thing is happening again, which means that Sophos makes python.exe to create the extra four threads.

Figure 8 shows the excessive amount of the waiting times for iexplore.exe processes in the YouTube experiment. More threads are created for the iexplore.exe processes in case of Sophos (74 threads) and Symantec (64 threads) than in Vanilla (60 threads). Also, figures 9 and 10 show the extra time the processes spend waiting on events.

It is obvious from figures 7, 8, 9, and 10 that the main reason of the overhead caused by the AV on the running processes come from the waiting time they spend on events. Also, it is obvious from the figures that the AV in most of the experiments enforce the tasks to spend more time on using the CPU (execute state). Although there is an overhead from the competition between the tasks and the AV on the CPU, this competition is not a main factor of the overall overhead because those tasks spend negligible time in the scheduler's ready queue.

A process' lifetime is affected by the system (hard-

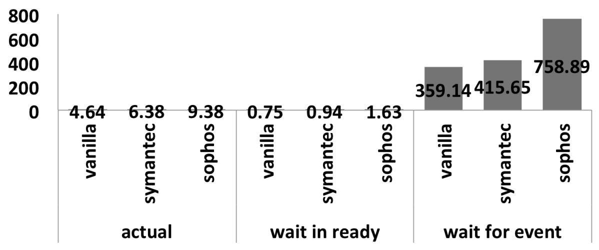

Figure 8: **YouTube experiment: processes' time (in sec) distributed between execute (actual), ready (wait in ready), and wait (wait for event) states.**

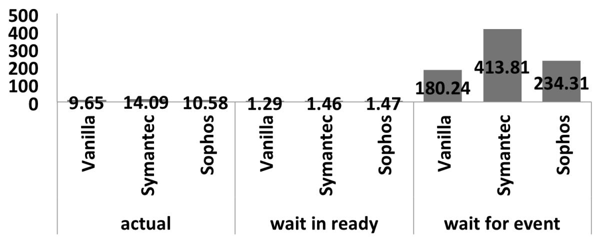

Figure 9: **Copy from Microsoft Word to Microsoft PowerPoint experiment: processes' time (in sec) distributed between execute (actual), ready (wait in ready), and wait (wait for event) states.**

ware and software) it is running on. Because we have fixed the hardware for all experiments, it is only the software that takes the role. Figure 13 shows the number of processes, threads, and images that were created/loaded before and during running the Client-Server experiment. The figure shows that the AV creates threads and loads images during the experiments to achieve something. Figure 14 shows the same thing happening in the YouTube experiment. Although this is not a direct performance implication, we can predict some intrusiveness from the AV to the tasks as the coming figures will show. Also, those extra threads could compete with the tasks on some resources like the hard drive and memory.

Figures 15 and 16 show a comparison between the numbers of file IO operations on the files involved in two different experiments by whatever process, including the AV. The intrusiveness of both AVs is obvious.

Figure 17 shows the total number of file IO operations directly made by the processes we care about. The figure shows the two different approaches Symantec and Sophos take to scan the task. Symantec approach is to enforce the running process to do more operations that is not originally designed to do. For example, python.exe did 128 file IO operations on "ProgramData/Symantec/Definitions/VirusDefs/20110313.002/VIRSCAN7.DAT" file. It is obvious that python.exe is not supposed to read Symantec's

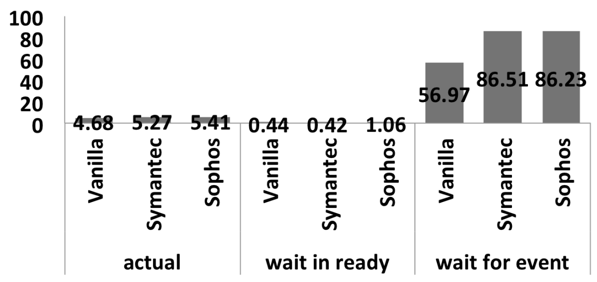

Figure 10: **Write to Microsoft Word Experiment: experiment: processes' time (in sec) distributed between execute (actual), ready (wait in ready), and wait (wait for event) states.**

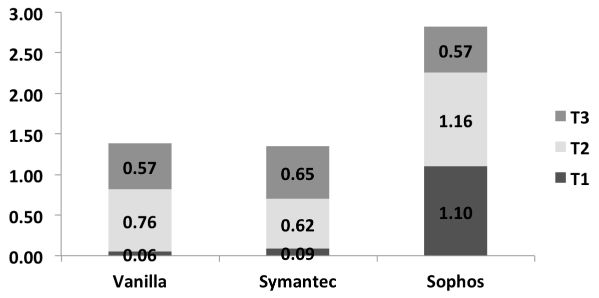

Figure 11: **Client-Server experiment: the wait time (in sec) for the 7za.exe threads.**

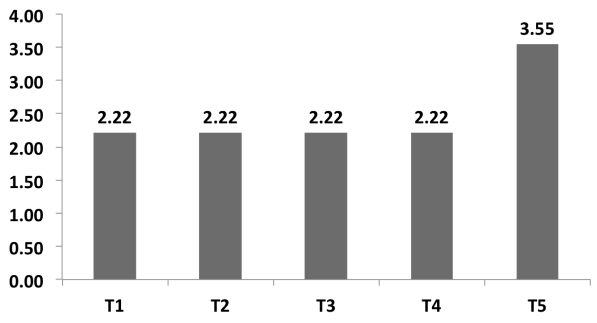

Figure 12: **Client-Server Experiment: the wait time (in sec) for python.exe threads created in case of Sophos. In the Vanilla and Symantec cases, python.exe has only one thread.**

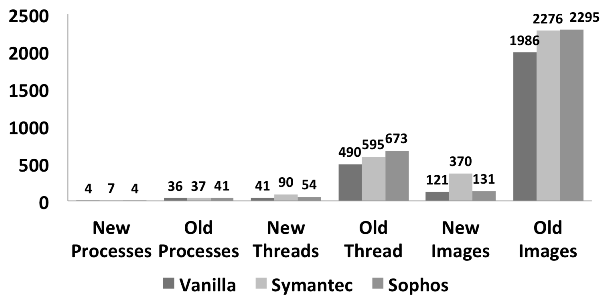

Figure 13: **Client-Server experiment: the total number of processes, threads, and images in the system before and during the experiment.**

signatures! This explains why in case of Symantec the number of file IO operations made by the processes we care about is much more than the Vanilla and Sophos cases. Sophos on the other hand does part of its job through detouring the execution of the running process. This is clear when the processes make file IO operations on sophos_detoured.dll, sophos∼1.dll, and swi_lsp.dll (Sophos Web Intelligence), so the load could be detoured to Sophos processes.

Figures 18 and 19 show the number of caused page faults. Hard faults are the ones that need hard disk access, while others are other kinds of faults like copy-on-write and demand-zero faults. When a process causes a hard fault, it needs to wait until the page is brought in memory. The high increase of hard faults in both Symantec and Sophos experiments is correlated to the overhead they add to the processes' lifetimes. Again, these faults are only accumulated for the processes we care about.

A system call proceeds from the user space (unprotected mode) to kernel space (protected mode) for the kernel to achieve a task on behalf of the user process and then get back to the user process. Increasing the number of system calls decreases performance. Figures 20 , 21, and 22 show the increase in the number of system calls made by the processes we care about in case of Symantec and Sophos, compared with the Vanilla case.

5 Discussion and future work

Although we claim that our approach is useful in characterizing the effect of the AVs on the running processes in terms of performance from the OS point of view, it is not perfectly accurate. We depend on finding differences between the Vanilla and the AV cases to conclude, which is loosely connected to the AV. Some operations might happen or not based on the system state or the state of the task that is about to execute. For example, if an AV uses the same DLLs the task will be using, then the task would not cause a hard fault in case of the AV because the AV might have already brought the DLLs before even the task starts. Also, a program could do some staff based on its last run or configuration which might make its execution to look different.

In this paper, we started from the files and processes we care about to find out what operations they do or are done on them. Another approach, which is a future work, is to start from the AV components to directly find out

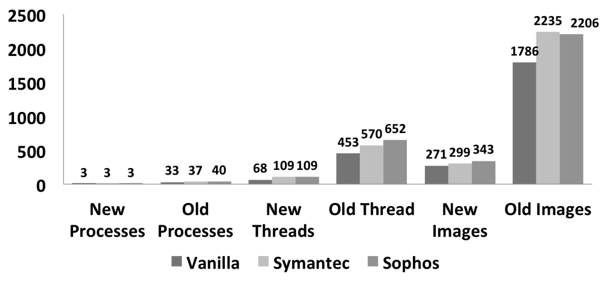

Figure 14: **YouTube experiment: the total number of processes, threads, and images in the system before and during the experiment.**

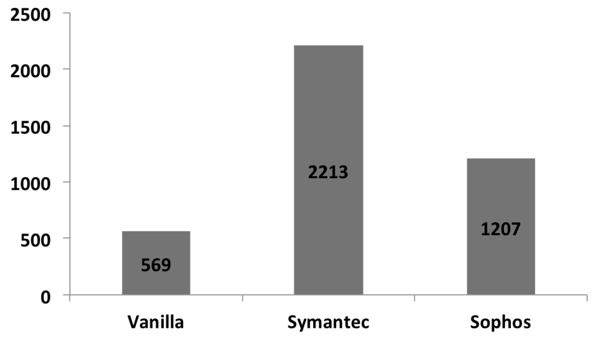

Figure 15: **Client-Server experiment: the total number of file IO operations with respect to the files involved in the experiment (python.exe, 7za.exe, cmd.exe, client.py, putty.a, putty.exe).**

what they are doing exactly. This approach is challenging, though, for that we need to know all the changes and the components the AV adds to the system when it gets installed.

6 Related work

The AV software is well known in the literature to prevent viruses and worms from spreading. Surfing the internet without having an AV is not safe. Although pattern matching is in the heart of the AV scanning engine, other techniques such as heuristics [11], code emulation, and algorithmic scanning are essential parts of modern AVs [18]. Little research has been conducted towards analyzing and improving AVs, mainly because most AVs are closed source. Few papers [13, 14] improved the scanning engine of the open source AV, ClamAV [1]. Al-Saleh *et al.* [7] shows that it is possible to create timing channel attacks against AVs. Christodorescu *et al.* [10] shows that it is possible to extract the signature for a specific virus that the antivirus is using to detect that virus. The closest to our work is [19], however they studied the performance of the AV purely at the hardware level while our approach is to characterize the AV performance from the OS point of view. Naive ways [6, 3] of measuring the AV performance are conducted by running a task and then compare the total time while running different AVs.

Software instrumentation [12, 8, 15, 9, 17] is a powerful technique to analyze programs' behaviors through adding extra code at certain positions of the instrumented programs. The purpose of instrumentation could be measuring performance, proving correctness, or assuring security of programs. Pin [12] is a dynamic binary instrumentation tool that can instrument binaries at the instruction level without modifying them. DTrace [9] is whole-system instrumentation tool for Linux. ETW is similar to DTrace, but ETW is integrated within Microsoft Windows kernel and it is very light, efficient, and easy to use.

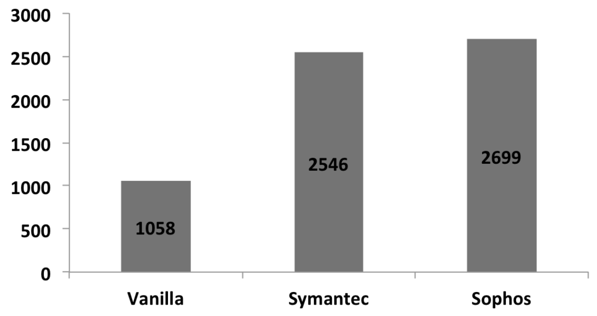

Figure 16: **Copy from Microsoft Word to Microsoft PowerPoint experiment: the total number of file IO operations with respect to the files involved in the experiment (winword.exe, powerpnt.exe, wordsourcedoc.doc, pres.ppt).**

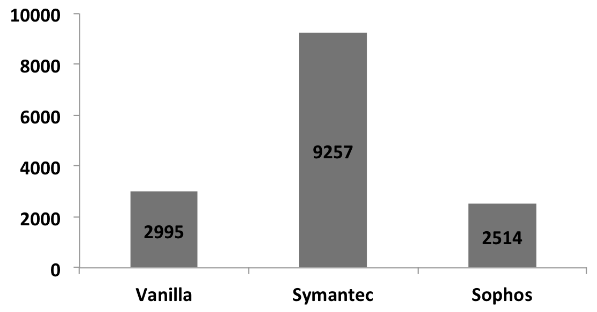

Figure 17: **Client-Server Experiment: the total number of file IO operations made by the processes involved in the experiment.**

7 Conclusion

Less care is given to study the functionality and the performance of the AV software despite how important and

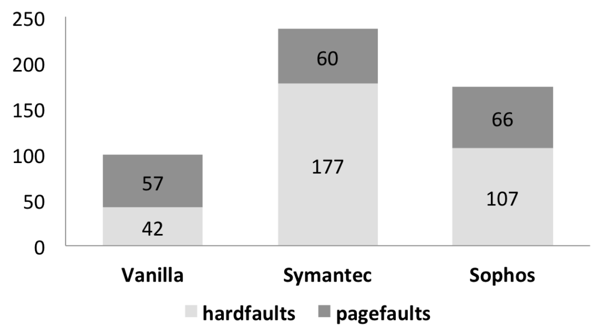

Figure 18: **Client-Server experiment: the total number of page faults made by the processes involved in the experiment.**

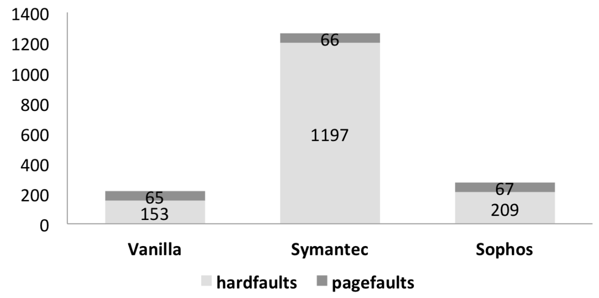

Figure 19: **Write to Microsoft Word experiment: the total number of page faults made by the processes involved in the experiment.**

widely used it is. In this paper, we investigated the performance issues for two commercial AVs. While naive techniques concentrate on the elapsed times of running tasks with the existence of AVs and others look at it from a purely hardware point of view, we focused on OS-aware approach to give more meaningful and accurate results. We used the Event Tracing for Windows instrumentation technology (a system-wide, kernel-integrated tool), xperf (a performance tool), and PowerShell (an automation and scripting framework in Windows) to design our experiments. Our experiments were designed to simulate common end-users tasks. Our results show that a considerable amount of performance overhead is added by the AVs because of the AV intrusiveness. The main impact of the AV on tasks is that they spend extra time waiting on events or using the CPU. The AV changes a task behavior by enforcing it do more file IO operations, page faults, system calls, and threads. We also show that a process behavior is changed with the existence of the AV. So, software development process (design, test, debug) and intrusion detection systems (which might look at a process changing its behavior as anomalous) should be aware of the AV existence.

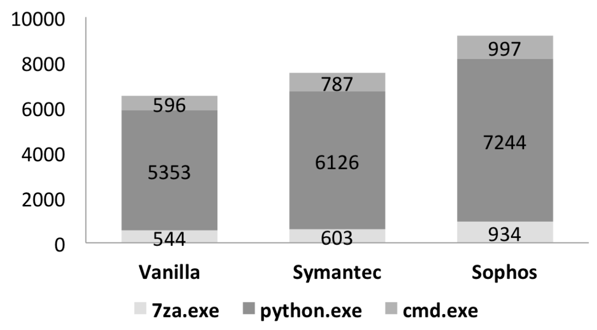

Figure 20: **Client-Server experiment: the total number of system calls made by the processes involved in the experiment.**

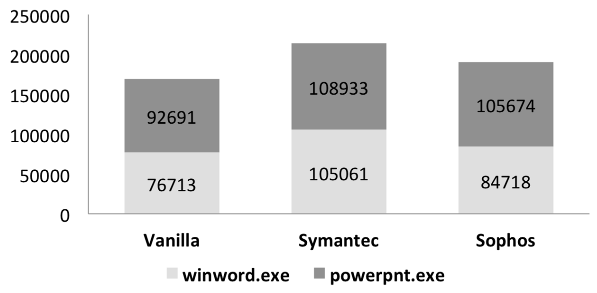

Figure 21: **Copy from Microsoft Word to Microsoft PowerPoint experiment: the total number of system calls made by the processes involved in the experiment.**

References

[1] Clam antivirus. `http://www.clamav.net`.

[2] Internet security threats will affect u.s. consumers holiday shopping online. `http://www.bsacybersafety.com/news/2005-Holiday-Online-Shopping.cfm`.

[3] Passmark software. `http://www.passmark.com/benchmark-reports/`.

[4] Postgresql: The world's most advanced open source database. `http://www.postgresql.org/`.

[5] Small and medium size businesses are vulnerable. `http://www.staysafeonline.org/blog/small-and-medium-size-businesses-are-vulnerabl`

[6] Tests of anti-virus and security software. `http://www.av-test.org/`.

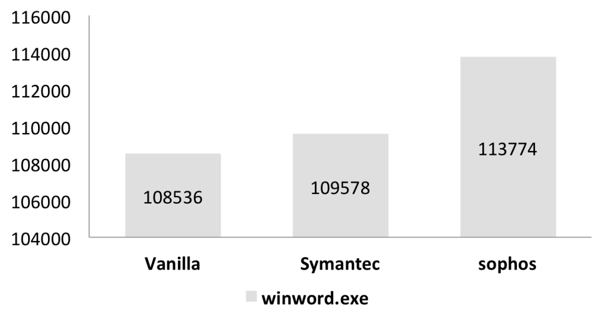

Figure 22: **Write to Microsoft Word experiment: the total number of system calls made by the processes involved in the experiment.**

[7] M. I. Al-Saleh and J. R. Crandall. Application-level reconnaissance: Timing channel attacks against antivirus software. In *LEET 2011: 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2011. To appear.

[8] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4*, 2000.

[9] B. M. Cantrill, M. W. Shapiro, A. H. Leventhal, and S. Microsystems. Dynamic instrumentation of production systems. pages 15–28, 2004.

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

[11] J. Eisner. Understanding heuristics: Symantecs bloodhound technology. symantec white paper series volume xxxiv. `http://www.symantec.com/avcenter/reference/heuristc.pdf`, 1997.

[12] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[13] P.-C. Lin, Y.-D. Lin, and Y.-C. Lai. A hybrid algorithm of backward hashing and automaton tracking for virus scanning. *IEEE Transactions on Computers*, 60:594–601, 2011.

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

[15] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV03*, 2003.

[16] D. I. Park and R. Buch. Improve debugging and performance tuning with etw. `http://msdn.microsoft.com/en-us/magazine/cc163437.aspx/`.

[17] A. Skaletsky, T. Devor, N. Chachmon, R. S. Cohn, K. M. Hazelwood, V. Vladimirov, and M. Bach. Dynamic program analysis of microsoft windows applications. In *ISPASS*, pages 2–12, 2010.

[18] P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

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