IRis

Northeastern University

Electrical and Computer Engineering Master's Theses

Department of Electrical and Computer Engineering

January 01, 2008

Real time anti-virus for a virtualized environment

Derek Uluski
Northeastern University

Recommended Citation

Uluski, Derek, "Real time anti-virus for a virtualized environment" (2008). *Electrical and Computer Engineering Master's Theses.* Paper 21. http://hdl.handle.net/2047/d1001715x

This work is available open access, hosted by Northeastern University.

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Real Time Anti-Virus for a Virtualized Environment

Author: Derek Uluski

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree:

Thesis Advisor: Date

Thesis Committee: Date

Thesis Committee: Date

Head of Department: Date

Graduate School Notified of Acceptance:

Associate Dean of Engineering: Yaman Yener Date

Copy Deposited in Library:

Reference Librarian Date

REAL TIME ANTI-VIRUS FOR A VIRTUALIZED ENVIRONMENT

A Thesis Presented

by

Derek Uluski

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University

Boston, Massachusetts

May 2008

© Copyright 2008 by Derek Uluski

All Rights Reserved

Abstract

Despite the pervasive use of anti-virus (AV) software, there has not been a systematic study of the characteristics of the execution of this workload. In this work we begin by presenting a characterization of four commonly used anti-virus software packages. Using the Virtutech Simics toolset, we profile the behavior of four popular anti-virus packages as run on an Intel PentiumIV platform running Microsoft Windows-XP.

In our study, we focus on the overhead introduced by the anti-virus software during *on-access* execution. The overhead associated with anti-virus execution can dominate overall performance. The AV-Test group has already reported that this overhead can range from 23-129% on live systems running *on-access* experiments [9]. [1] The performance impact of the anti-virus execution is clearly an important issue, and we present the first quantitative study of the characteristics of this workload. Our study includes the impact of both operating system execution and system call execution.

Prior work has quantified how much overhead is introduced by the execution of a real-time anti-virus scanner and have indicated that significant amounts of pressure is placed on not only the central processor, but the memory subsystem. Many solutions have been proposed to potentially offset or alleviate this overhead; however few have actually been implemented.

[1] Comparison tests were done during 2001-02 on earlier versions of the anti-virus packages. We are using more recent versions of these packages.

The issue of pressure on the physical system becomes more immense in a virtualization environment, where multiple virtual machines are executed on one physical machine, and each virtual machine running the windows operating system requires its own instance of an anti-virus program.

The purpose of this work is to take one step forward in addressing the growing issue of real time anti-virus execution based overhead in the context of application consolidation. The VMWare ESX architecture provides the ideal environment for a distributed real-time scanning process to be executed amongst all machines, consolidating physical memory consumption and exploiting thread-level parallelism. We implemented a prototype of such a model, and study the performance of the prototype design.

We found that as well as designing a real-time anti-virus system that has scalable performance, we can alleviate up to 78% of the overhead introduced by commercial Anti-Virus packages, and in the worst case, consolidate up to 260 MB of memory consumed by anti-virus packages.

Contents

Abstract **iv**

1 Introduction **1**

1.1 Contributions 2

2 Related Work **4**

3 Anti-virus Software **10**

3.1 Simulation framework 13

3.2 Anti-Virus Characterization 16

3.2.1 Memory Accesses 16

3.2.2 Instruction memory footprint 18

3.2.3 File Types 21

3.2.4 Discussion 21

4 Virtualization and Anti-Virus **23**

4.1 Virtualization Technology 23

4.2 VMWare ESX Server 25

4.3 ESX Anti-Virus Overview 25

5 ESX Anti-Virus **28**

5.1 Development History 28

5.2 On-Demand System Results 32

5.3 ESX Anti-Virus Prototype 38

5.4 General Design Model 40

5.5 AV Client 42

5.5.1 System Service Table 43

5.5.2 Invoking the Scan 46

5.6 AV Server 48

5.6.1 AV Server Initialization 49

5.6.2 AV Server Daemon 51

5.7 Synchronization Model 53

5.7.1 Monitor 54

5.7.2 VMKernel 58

6 Results and Analysis **61**

6.1 Experimentation Setup 61

6.2 Results 65

7 Summary **73**

7.1 Conclusions 74

7.2 Future Work 75

Bibliography **78**

A IO-Stressing Benchmark **83**

List of Figures

3.1 Anti-virus performance degradation. 12

3.2 Anti-virus overhead. 13

3.3 Multi-level architectural & micro-architectural simulation environment. 14

3.4 Cumulative memory accesses during execution of the copy test. . . . 17

3.5 Cumulative memory accesses during execution of calc. 17

3.6 Cumulative memory accesses during execution of wordpad. 18

3.7 Cache hit ratio for the copy test. 19

3.8 Cache hit ratio for calc. 19

3.9 Cache hit ratio for wordpad. 20

3.10 Anti-virus instruction memory footprint while executing Helloworld. . 20

3.11 Overhead for different file types. 21

4.1 High level Abstraction of the ESX Server running a common anti-virus process communicating with contained virtual machines 26

5.1 Scanning performance of McAfee Enterprise 8 scanner and prototype system against a wall clock (On-Demand Scanning) 33

5.2 Memory consumed by entire system attributed to Anti-Virus execution (On Demand) of prototype system and McAfee Enterprise virus scanner 35

5.3 Comparison of various cases of realtime execution against the business winstone 2004 benchmark 36
5.4 Representation of the amount of increased pressure exhibited on the translation cache by the execution of various Anti-Virus packages due to increased guest kernel level activity. The workload used is discussed further in section 6.1 39
5.5 Abstraction of the ESX server architecture, summarizing the entire AV-Prototype configuration 42
5.6 Abstraction of the AV Client summarizing step by step the process that the client goes through to invoke a scan up to the monitor component 44
5.7 Abstraction of the AV Client summarizing step by step the process that the client goes through when scanning results are returned from the monitor. 49
5.8 Abstraction of the AV Server illustrating the initialization process as well as the task of a worker thread 54
5.9 Abstraction of syncronization between the AV Client and AV Server via the VMKernel. 55

6.1 Real time scanning performance measurement of Trend Micro PC-Cillin, McAfee, and AV-Prototype using the Business Winstone benchmarking suite - Higher is better 66
6.2 Real time scanning performance measurement of Trend Micro PC-Cillin, McAfee, and AV-Prototype using the IO-stressing synthetic benchmark for different levels of concurrency. All scores are based on real time (wall clock) - lower is better. 67

6.3 Real time scanning memory consumption measurement of Trend Micro PC-Cillin, McAfee, and AV-Prototype using the IO-stressing synthetic benchmark for different levels of concurrency. All measurements are presented in MB - lower is better. 70

6.4 Representation of the amount of increased pressure exhibited on the translation cache by the execution of various Anti-Virus packages due to increased guest kernel level activity. 72

List of Tables

3.1 Structure of the P4 microarchitecture used in this work. 14

3.2 Five environments evaluated: Base has no anti-virus software running. 15

Chapter 1

Introduction

Security is an important issue for all computer users. A significant amount of overhead is introduced if we enable anti-virus scanning. Many users are unhappy with the performance penalty they must pay for security. The amount of overhead introduced can be so significant that many users will defer virus scanning or totally disable their anti-virus software. Then their system will be vulnerable to viruses. Thus, it is important to address the performance overhead associated with anti-virus software execution.

Most anti-virus software packages employ a range of scanning techniques to decide whether or not a given file is infected. More complex techniques also exist such as: sandboxing, digital watermarking, and heuristic-based techniques [46].

There are two main usage models when running anti-virus software, 1) *on-demand*, and 2) *on-access*. The *on-demand* model involves the user specifying which files to scan. In this case, the anti-virus software will usually be running for a period of time, scanning numerous files. *On-demand* scanning is usually performed offline, when the user does not use the computer. The *on-access* model can be thought of as a daemon

process that monitors system-level and user-level operations and intervenes (scans) when a predefined event occurs. Most AV software is configured to run in *on-access* mode. In this study we will focus on execution overhead associated with an *on-access* model.

1.1 Contributions

This thesis attempts to build on several ideas that have been proposed as the field of data integrity has matured. One of the fundamental building blocks of virus prevention is the antivirus scanner itself [35, 34, 4, 5]. The idea of a simple process scanning files on a single system has been extended to a distributed model, where the antivirus process is responsible for several systems [13, 31]. This thesis takes these concepts and applies them to the *Virtualization* execution model. The contributions of this work include an implementation of a signature based scanning engine tailored to a system running with a hypervisor, a syncronization model for scanning processes to pass data amongst system threads, and a lightweight windows based driver that is capable of communicating with the hypervisor and the scanning process that the hypervisor manages.

The rest of this work is organized as follows. First we present other works related to the performance and effectiveness to anti-virus and other such malware prevention. Then, we present data showing the performance penalty due to anti-virus execution in section 3. In section 3.1 we discuss our Simics environment and in section 3.2 we present some results from our workload characterization. We propose some methods to potentially alleviate some of the overhead in section 3.2.4. We then discuss one

implemented solution for the VMWare ESX platform in 4.3. The history of development is shown in section 5.1. Some analysis of the previous implementations is shown in section 5.2. An overview of the final version of the prototype is discussed in section 5.3. The details of each component of the final prototype are discussed in sections 5.4, 5.5, 5.6. A discussion of the experimentation methodology is presented in 6.1. Analysis of the results are shown in section 6.2. Finally, conclusions are drawn off of all the work in 7.1.

Chapter 2

Related Work

Many methods exist today that are used to guard against virus attacks. Anti-virus packages are commonly used to guard against know viruses. Most anti-virus software packages employ signature matching as the main mechanism to identify viruses [46], such as fprot [5], clamAV [4], Trend Micro PC-Cillin [35], Symantec Norton Anti-Virus [17], and McAfee [34]. Due to a heavy reliance on a signature database and signature matching, there have been other efforts to leverage the performance of this type of anti-virus package. Since the signature databases are ever-growing in these packages, Badhusha et al [3] propose a system for active packet-oriented signature file updating in real time.

An alternative strategy involves *behavior blocking*, wherein the behavior of a binary is analyzed and the rate of connections to a new host is limited [52]. Mechanisms that execute untrusted software in a sandbox, while monitoring behavior, are described in [46]. Nachenberg et al [39] propose a state-based cache for antivirus software in which a CPU emulator is executed upon the access of a file, where the execution

of the file is emulated in a contained environment. In such a system, the state of the machine at the time of execution must be maintained. Such sandboxing systems introduce large amounts of overhead and would benefit in an environment with dedcated resources to performing sandbox execution and heuristical analysis.

Other forms of heuristical analysis have been drawn from the area of biology. The term *neural network* refers to a network of neurons, or in the context of computer science, artificial nodes. Such a artificial network is used to gain an understanding of artificial intelligence systems without creating a full scale model [50]. Tesauro et al [23] propose a neural network system for generic detection of boot sector viruses, which are viruses that reside in the boot parition of a given disk and contain code that is executed when the master boot record (MBR) of a disk is accessed. The neural network system is used to simulate the behavior of a given piece of data and learn from its execution patterns, similar to a combination of heuristical analysis and sandboxing of data.

Other proposals that draw relations between biological viruses and computer viruses have been proposed as well. Kephart et al [30] propose an entire immune system for a system, involving the capturing of a sample of given data for analysis. The analysis includes two main components, a disinfection heuristic, and a disinfection algorithm. This method is to prevent malware from destroying data, using a series of execution states and cyclic redundancy check (CRC) checksums. If a particular piece of code attmepts to destroy legitimate data, the immune system can use a calculated CRC checksum to reconstruct the destroyed data. This system also works well for viruses that attempt to hide themselves by making their injected malicious

code appear to be the same size as that of the data in which it destroyed.

Other methods of antivirus have been proposed as the filesystem or disk block level. Miretskiy et al [53] proposes a stackable file system where incremental scanning takes place on potentially infected data before it is committed to disk. This method allows for the added flexibility of creating versions of potentially infected files that can be recovered or isolated before the data is commited to the disk. Miretskiy's approach is based on the clamAV [4], modified for logarithmic scalining as the number of signatures in the antivirus database increases. Lettvin et al [32] propose a hidden partition, where storage space is provided to a special antivirus mechanism at the disk block level to detect malware prior to post-bootstrap operations, indepedent of the execution of any operating system. Such a model is desirable for detecting boot partition viruses.

The idea of dedicating resources have been proposed, and applied to a distributed model. Chen et al [13] propose an Antivirus system where the server computer, containing whatever database it needs to protect (such as email), has a module that contains the routines encapsulated in an antivirus package. When a client makes a request to the server, before distributing the data to fulfill the client request, the server antivirus module will scan the file. Taking this to the next level, Oberheide et al [31] state that the antivirus process be offloaded to a series of collaborating services that, from the client point of view, resides in the network as a service. Each client participating on the network has a small lightweight client that will intercept any incoming data and send it for verification to the network services available to it on the network. The efficiency of these network services is increased by increasing

the number of services that are collaborating with each other on the network. Furthermore in this proposed model, specific scanning duties or entire separate scanning engines can be contained as heteorgeneous scanning nodes in the network.

An important class of software-based intrusions include stack smashing attacks [14, 54]. This class of attacks enables an intruder to redirect execution to malicious code by overwriting the return address that is stored on the program call stack. Stack smashing attacks can be addressed in several ways. StackGuard [18] is a compiler-based approach which places a canary key next to the return address on the program stack and validates the integrity of the return address. LibSave [10] presents a method where special libraries are loaded dynamically that intercept calls to known, unsafe functions.

Hardware-based solutions for stack smashing also exist. StackGhost [22] provides a hardware-based stack protection; the hardware is responsible for encrypting and decrypting return addresses. Another approach described in [54] enhances the return stack address to detect buffer overflow attacks. Wambach et al [33] propose a hardware solution where memory controllers have mechanisms on them in hardware to prevent data from being written to, providing methods that can be controlled by the operating system or manual protection mechanisms at the circuitry level. Additionally, this protected memory can be used contain lists of physical addresses that the computer can not write to.

In the area of anti-virus software characterization, the AV-Test group has published online results of measuring the overhead associated with different anti-virus

softwares [9]. They compared the impact of running a range of anti-virus scenarios. Another comparison of different anti-virus software can be found in [12]. [8] also contains accuracy testing on all of the major commercial signature based antivirus packages available today.

Basu et al [43] propose a tool that makes uses of a systems performance counters to analyze all the running processes of a system at any given point in time. The data collected from the performance counters are used to create a statistical model that breaks down running processes and their corresponding threads and sorts all of the entities by their degree of anomaly. Each entity, or feature in the context of the tool, is treated as a univariate gaussian. Each feature is tested against common features found in the analysis and then computed against the *average likelihood per feature*, yielding their degree of anomaly. This tool could potentially be used to identify bottlenecks in an entire system, particularly one succesptible to multiple attacks.

There have been a few studies that have proposed solutions to overcome anti-virus execution overhead. In [44], the authors analyze the underlying algorithms of open source anti-virus projects [4, 6] and propose a *CAM-based* co-processor for boosting anti-virus software execution performance. In [45], Symantec (the developers of Norton Anti-Virus) describe a anti-virus scanning hardware mechanism that would exist on a telecommunications network. They suggest using a finite state machine to match multiple signatures.

Tatari [47] describes the implementation of a co-processor that is capable of simultaneously matching complex regular expressions. Similarly, Nachenberg [40] proposes

an anti-virus co-processor that contains volatile memory that serves as a cache for the on chip antivirus processing module. The purpose of the non volatile memory is to preserve cache locality for the execution of the on-chip antivirus module, as well as cache for frequently accessed data. Offloading the duties of an antivirus program from the central processor also significantly increases the overall performance of the system [20]. Waldin [42] encapsulates this technology in a processor that can process data through a network controller.

Next, we will present a number of characteristics of anti-virus software execution.

Chapter 3

Anti-virus Software

The foundation of all antivirus software is based on the concept of signature matching. Essentially, any company that releases an antivirus package has their own set of analysis tools used to uniquely identify a particular piece of data. Customers, or end users, will report suspicious behavior on their systems and submit any new data committed to their systems to the antivirus companies for consideration of a newly created virus. Once the behavior of the submitted data is considered to be malicious, through various proprietary methods, the unique identifier will be extracted.

This unique identifier is referred to as the signature of the malicious data in question, and is usually represented by a sequence of binary codes. Antivirus packages will commit this signature to a self-managed and contained database that it refers to when analyzing files in question. When a new file is accessed, before the user is allowed to take control of the file (and execute it), the antivirus will intercept the user request, analyze relevant portions of the file in question, attempting to match a sequence of bytes contained in the file to that of the many signatures contained

within its signature database. If a signature matches a sequence of bytes in the file in question, it is considered a virus and access to the file is prohibited. The methods for which this regular expression matching algorithms are implemented varies amongst the different antivirus packages. Conceptually, they all perform the same problem. Since these packages all rely on a signature database that only grows as more viruses are discovered, they will all suffer from a larger execution time as they intercept user IO requests.

Next we will quantify the amount of overhead introduced by anti-virus software. We will defer a discussion of the details of our evaluation framework until section 3.1. Figure 3.1 plots the increase in execution time due to anti-virus overhead. We study three different test scenarios: 1) copying a small executable from the CDROM to the hard disk, 2) executing calc.exe, and 3) executing wordpad.exe. All of this execution is running under Windows XP professional. The value shown in each bar is the percent increase in execution time relative to a base case (the base case is the same scenario run without any anti-virus software present).

We conducted a second experiment to determine the number of extra instruction executed while performing file system operations and while loading/executing a binary. Both scenarios involve a small Helloworld binary of 28KB in size.

Most of the anti-virus code executed is located in tight loops that perform string scans. We have found that anti-virus execution is dominated by a very small number of very hot basic blocks in each anti-virus package: 3 basic blocks for Cillin and F-Prot, and less than 20 basic blocks for McAfee and Norton (containing 109 and 226 instructions total, respectively).

In figure 3.2, we plot the number of dynamic instructions executed. We show the total number of instructions executed (total) and also the number instructions

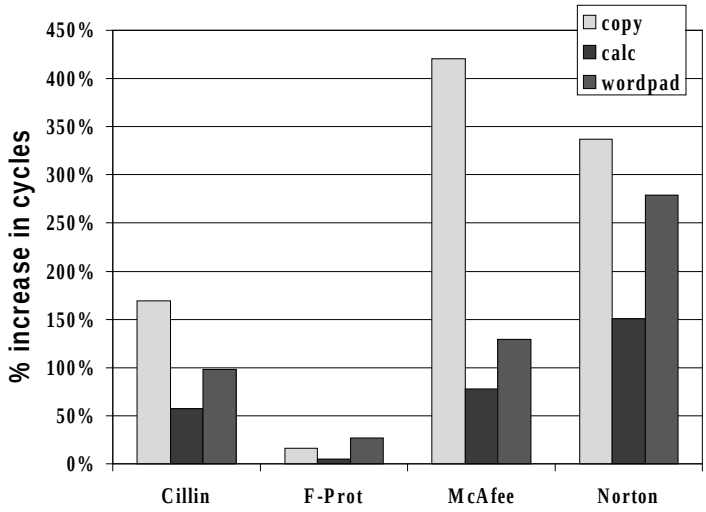

Figure 3.1: Anti-virus performance degradation.

executed that reside in *hot* basic blocks. We consider a basic block as hot if it is visited more than 50,000 times. We collect all the virtual addresses, labeling each basic block as hot and cold, and compute the percentage of instructions executed that reside in hot basic blocks.

For Cillin, McAfee and Norton, the scanning algorithm used has a relatively small footprint and is frequently revisited. This opens the door for optimizing the most frequent basic blocks, which may lead to a significant reduction in the performance penalty introduced by anti-virus execution.

Next, we will discuss our simulation environment for this characterization work.

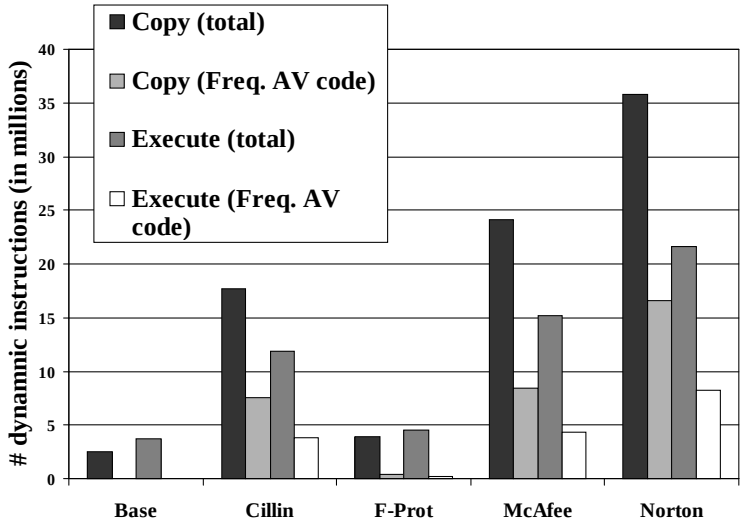

Figure 3.2: Anti-virus overhead.

3.1 Simulation framework

To study anti-virus behavior, it only makes sense to use a platform where a majority of the virus attacks have been targeted, and where there exist a number of commercial anti-virus packages available. We have chosen to build our studies on top of the Virtutech Simics toolset [48], a full machine-state architectural simulator that can emulate a faithful model of a large number of micro-architectures. Simics allows us to profile the complete instruction stream executed by the processor (including operating system and library execution), as well as capture all memory and I/O activity. The Simics toolset also includes a cycle-accurate micro-architectural model which we use to obtain cycle-accurate performance numbers.

The Simics model we are using is known as the *Dredd model*, a 2GHz Intel PentiumIV with 256MB of memory. This model contains a generic motherboard containing

a model of the Intel 440BX chipset. The goal in modeling this class of machine is to capture the execution of an anti-virus software on a representative system. In order to obtain performance metrics, the instruction stream executed is passed to the micro-architectural simulator. We configure Simics to simulate a current Intel PentiumIV microprocessor.

Processor Model	Intel Pentium 4 2.0A
Processor Operating Frequency	2GHz
L1 Trace Cache	12K entry
L1 Data Cache	8KB
L2 Cache	512KB
Main Memory	256MB

Table 3.1: Structure of the P4 microarchitecture used in this work.

Figure 3.3 shows the organization of our evaluation environment. Our simulated host (Dredd) is executing Windows XP (loaded from a simulated harddrive). On top of Windows XP we install and run the anti-virus software, as well as our test scenarios. We compare execution taken from a baseline configuration (without any anti-virus

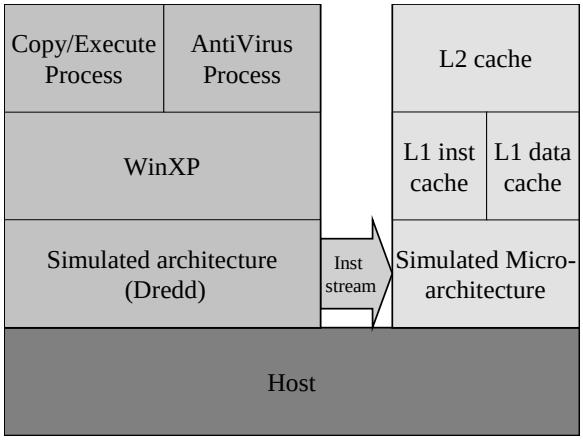

Figure 3.3: Multi-level architectural & micro-architectural simulation environment.

software installed), as well as systems that have 4 different anti-virus packages installed. For our initial configuration, we have installed Windows XP professional (2002). This is the *Base* configuration and it has no anti-virus software installed. We then created four more configurations on top of the Base configuration, one for each anti-virus software package. In order to minimize the interference of background processes, we collect profiles after rebooting Windows XP and simulating for 100 billion simulation steps (Windows XP boots in less than 7.17 billion steps. A step in the simulator is an execution of an instruction, an exception or an external interrupt).

Table 3.2 summarizes the 5 different configurations:

Configuration	Anti-Virus edition	version
Base	-	-
NAV	Norton Anti-Virus Professional 2004	10.0.0.109 [1]
PC-Cillin	Trend Micro Internet Security	11.0.0.1253
McAfee	McAfee Virus Scan Professional	8.0.20
F-Prot	F-Prot Anti-virus for Windows	3.14b

Table 3.2: Five environments evaluated: Base has no anti-virus software running.

For each experiment we created an image file that is loaded as a CDROM inside the emulated machine. In order to facilitate accurate profile collection, we execute a utility at the start and the end of each collection. This utility contains a special instruction (interpreted by Simics as a breakpoint) which allows us to turn on and off profiling as needed.

We study three different operations that invoke anti-virus scanning. In the first, we copy a file from the CDROM to the harddrive. In the next two scenarios, we study two Windows XP accessories: calculator, and wordpad. We run these applications by accessing them through a shortcut. [2] Each experiment is run multiple times to check for reproducibility. We use the same image for all profiles. We captured at least

[2] running the shortcut has a similar effect as running a program in the background

5 profiles per scenario and found less than a 1% difference in most of the workload parameters studied across profiling runs.

It is important to note that the statistics gathered include all execution between the two breakpoints. The data collected includes more than our test case and the anti-virus program. There is some overhead introduced by the breakpoint utility, the test case command shell, and a number of operating system background processes. Note also that the utility program executed has a prefetching effect: The AV program will scan it too, thus *prefetching* the anti-virus code and signature database.

3.2 Anti-Virus Characterization

Next, we present a sample of different memory access patterns and cache hit ratios obtained in our study. We also analyze the instruction memory footprint and the impact of scanning different file types.

3.2.1 Memory Accesses

In the following results, we consider our 3 scenarios of a copy, and 2 executions of Windows-XP utility programs (calc and wordpad). In figures 3.4, 3.5, and 3.6 we show the cumulative number of memory accesses executed for the 3 scenarios. We present statistics for the number of L1 instruction and data cache references, as well as L2 cache references.

We can see some clear trends across all applications. We see a consistent increase in the cache activity for each of the anti-virus workloads. This overhead is smallest for F-Prot, which performs the least amount of scanning. Norton introduces the most overhead. It is interesting to see that the impact to the L1 data cache and the L2 is

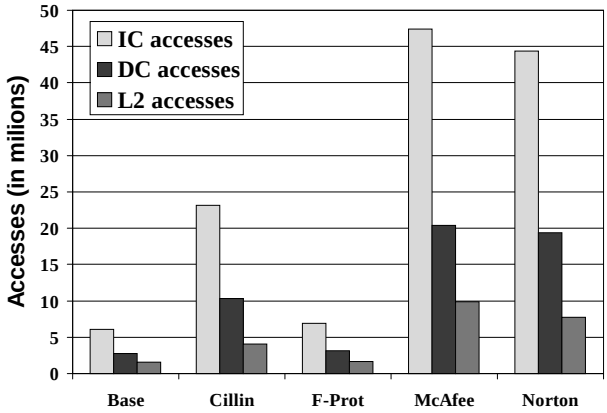

Figure 3.4: Cumulative memory accesses during execution of the copy test.

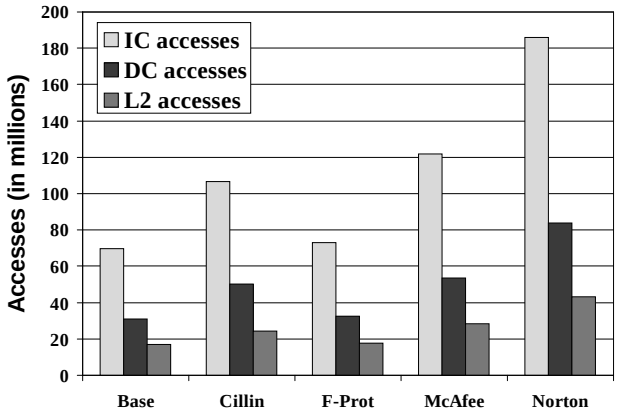

Figure 3.5: Cumulative memory accesses during execution of calc.

directly proportional to the number of accesses in the L1 instruction cache. The L2 impact shows that the L1 miss rate scales linearly with the number of references to the L1 instruction cache. We can attribute most of this overhead to capacity misses caused by the anti-virus execution.

We present cache hit rates in figures 3.7, 3.8, and 3.9. We break down read

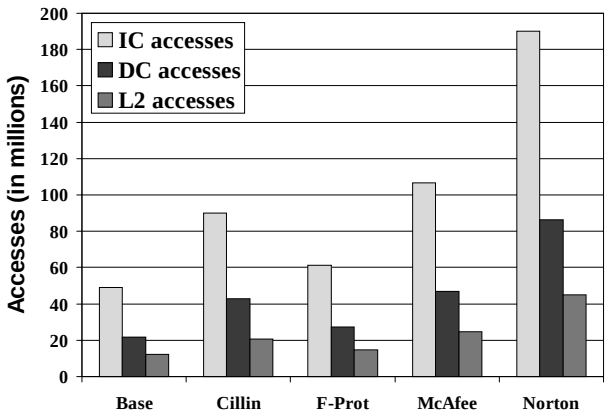

Figure 3.6: Cumulative memory accesses during execution of wordpad.

accesses to L1 and L2 for instructions and data. Note that L2 is shared, while we have separate L1 instruction and data caches. We see fairly consistent results for the 3 scenarios except for Norton, where the L2 hit rate is much higher. We can see that we capture a lot of the working set associated with the anti-virus execution that falls out of L1 and resides in the L2 cache. Norton possesses the largest working set of all the programs, so it make sense that the L2 cache should provide more of an advantage to Norton than to the other anti-virus packages.

3.2.2 Instruction memory footprint

In figure 3.10, we show the instruction memory footprint for each anti-virus program while copying the Helloworld binary. The graph shows the cumulative number of unique instruction addresses touched over time. The results show that anti-virus software packages (in particular Norton and McAfee) have a somewhat larger footprint than the Base case. The additional addresses that need to be fetched impact cache

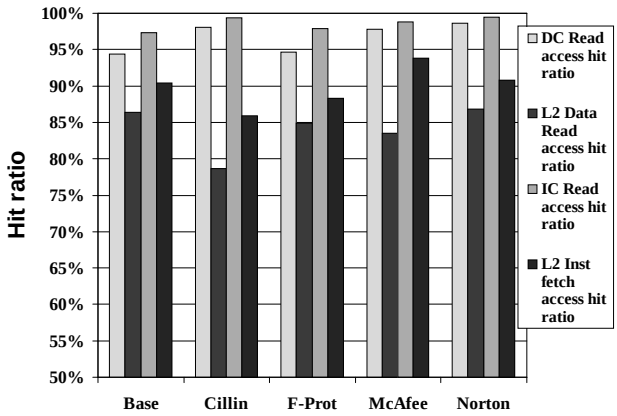

Figure 3.7: Cache hit ratio for the copy test.

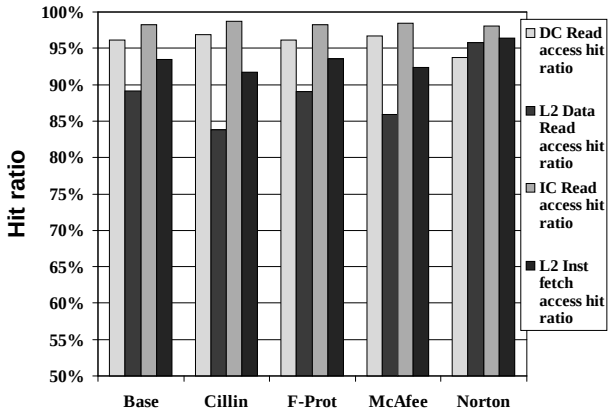

Figure 3.8: Cache hit ratio for calc.

performance.

In this figure we can also see three distinct *spikes* occurring during the execution. These spikes represent the copy process, the anti-virus process and the utility code process. The middle spike represents the anti-virus software and shows an increase of approximately 40,000 instructions (which is on the same order of magnitude as the

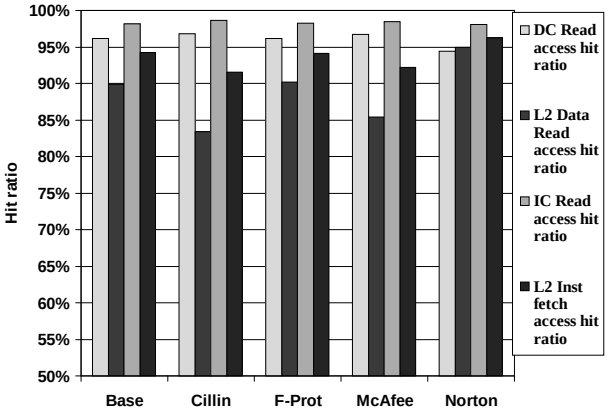

Figure 3.9: Cache hit ratio for wordpad.

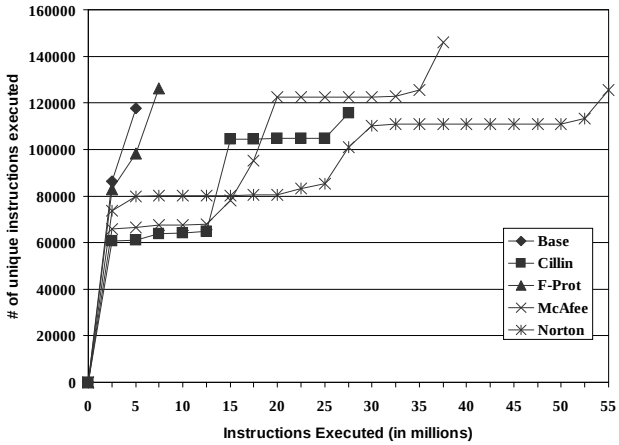

Figure 3.10: Anti-virus instruction memory footprint while executing Helloworld.

footprint of the copy process.)

3.2.3 File Types

Since anti-virus programs use different algorithms to scan different file formats, we ran experiments that perform copies of different file types. The files types include: .dll, .doc, .exe, .html, .jpg, .mp3, .ppt, .sys, .xls. All files are 128KB in size. We measured the number of dynamic instructions associated with each AV when the files are copied. We show results in figure 3.11.

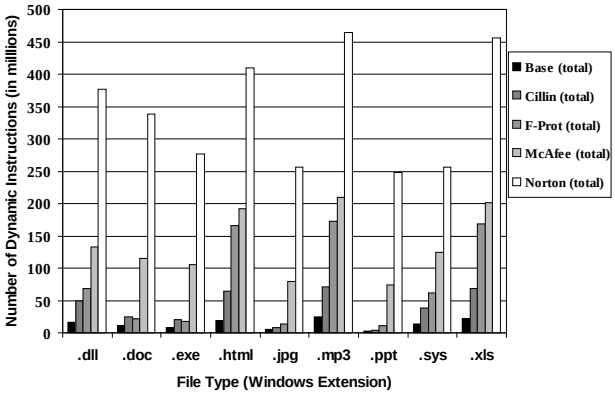

Figure 3.11: Overhead for different file types.

3.2.4 Discussion

Based on the some of the data collected during our characterization study, we have begun to develop hardware-based solutions to reducing anti-virus execution overhead. Our initial idea was to extend the ISA by adding new *fused* instructions that would execute a fixed sequence of (2/3/4) instructions that occurred frequency in hot portions of anti-virus execution. Those sequences could potentially be fused, reducing the overhead of the scanning operations (think of these as customized string operations).

The only problem with this approach is that we found different sequences present in different anti-virus software. If anti-virus software used a common set of libraries for scanning, then we may be able to employ this kind of accelerator. This solution can potentially reduce a significant amount of the overhead, it does not completely hide the overhead associated with scanning.

Another approach is to design an anti-virus co-processor, with the co-processing running all the scanning algorithms, alleviating the main processor of the arduous task of scanning a binary. We are proposing to extend the ISA to allow a program or operating system to control the operation of the co-processor. The co-processor would scan the binary while the main processor continues with normal execution.

Another approach is to incorporate additional functionality into a memory controller that can scan instructions and data as they are being fetched. Essentially, the anti-virus software equipped with a virus definition or signature database, is executing on the memory controller and acts as a filter that is transparent to the main processor and the user. The main problem with this solution is maintainability. Anti-Virus software is only effective if it is continuously updated. Any solution incorporating the anti-virus in hardware would have to take into account frequent updates both to the signature database and possibly the software and/or algorithms.

This work is intended to be a first study of anti-virus execution behavior, performance and benchmarking. We intend to continue studying the effects of anti-virus software execution on performance, and suggest new mechanism making use of virtualization techniques to reduce this overhead in the subsequent parts of this paper.

Chapter 4

Virtualization and Anti-Virus

4.1 Virtualization Technology

There exist several definitions to the term virtualization. The general idea behind the term virtualization is to hide the physical characteristics of a system so that other systems and applciations can interact without knowledge of the abstraction of those resources [7]. More specifically, there exist several types of virtualization. They can be divided into two main categories: *Platform Virtualization* and *Resource Virtualization* [51].

Resource Virtualization involves the virtualization of specific resources of a system, such as storage devices and network resources. Platform Virtualization is a technique which presents multiple uses with seemingly independent systems [19]. In the context of this work, the VMWare ESX [25] is the implemented virtualization platform. Therefore, this work pertains to the platform virtualization model, and will only consider such subsequently.

In the context of platform virtualization, there exist several subcategories. Specifically, there exists *emulation/simulation*, *native/full virtualization*, *paravirtualization*, and *application virtualization* [51]. In the context of emulation virtualization, a particular platform is entirely simulated in software. A good example of an emulation platform would be the bochs simulator, a program that emulates the IA-32 architecture [11].

Paravirtualization entails offering a special API in the operating system that communicates with a resource manager, typically called a hypervisor [41]. An example of this type of implementation is the Xen virtual machine monitor [38].

Application virtualization essentially entails containing the necessary resources in a software application to create an environment for common execution for a set of applications. Such an implementation that is used commonly today is the Sun Java Virtual Machine [37].

Native or full virtualization is of particular interest in the context of this work. This virtualization model provides an abstraction of resources necessary to run unmodified software in a fully contained environment. In this model, the software is run on the same processor as the native hardware. Although Xen [38] supports this, the primary focus will be on the VMWare ESX [25], discussed in the next section.

4.2 VMWare ESX Server

VMWare has a slightly different approach to native virtualization. Essentially it contains the properties of native virtualization, encapsulating various resources of an entire platform. VMWare refers to this as a *virtual infrastructure* [29]. VMWare takes this one step further by introducing the *Bare-Metal Architecture*, also referred to as *Hypervisor architecture* [29]. This essentially is an operating system in itself that runs in the background, consuming as little resources as possible in order to deliver the maximum amount of resources available for the virtual machines themselves.

There are several resources that are managed under the hypervisor model. Generally speaking, the ESX server is responsible for the management of CPU utilization, memory, and all other I/O devices [28]. The monitor has the ability to distribute equal priority across all virtual machines for physical and virtualized resources, although it also has the ability to distribute these resources as the administrator sees fit [26].

The VMWare ESX server is the chosen virtualization platform to implement an anti-virus architecture distributed amongst virtual machines, as discussed further in the subsequent sessions.

4.3 ESX Anti-Virus Overview

The purpose of this prototype is to take the necessary components of an on-access anti-virus system and distribute its use across all the virtual machines being executed on the ESX server. There are multiple reasons for the design of this prototype.

First and foremost, the ESX server is capable of running multiple operating systems concurrently. The primary focus with respect to operating systems is the Windows Operating system [15]. For each windows based virtual machine, an instance of anti-virus must be run on each. Since the workload for anti-virus will be the same across all virtual machines, it is a workload that can be considated.

The idea is to have one anti-virus process running a layer above the virtualization layer for the virtual machines and have each virtual machine communicate directly with this process, as depicted in figure 4.1.

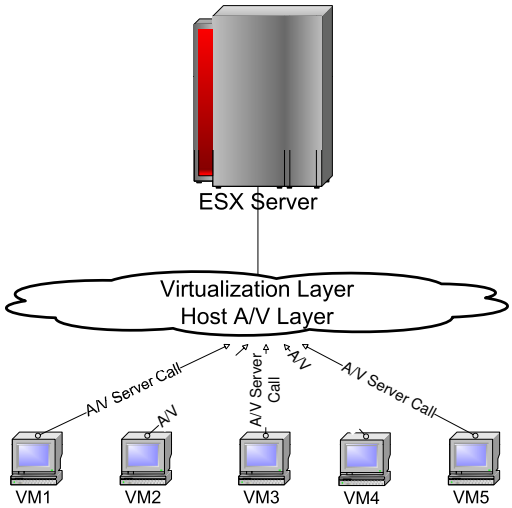

Figure 4.1: High level Abstraction of the ESX Server running a common anti-virus process communicating with contained virtual machines

There are two general components to the design of the prototype. The first of which is the scanner backend (designated the *Server Daemon*), which is responsible for performing the actual scanning process, and the client front end (designated the *Client*), which is responsible for making the necessary data available to the scanning engine

residing in the scanner backend.

The server daemon is a POSIX [24] based application, which is software that is compatible with variants of the Unix operating system, that previously ran from within the Console Operating System of the ESX Server and has also been optimally compiled on a VSMP virtual machine for optimized parallel execution. The core of the server daemon is its scanning engine, which is a Trend Micro [35] designed engine running lower level API calls to minimize execution overhead. This scanning engine performs the duties of the conventional scanning engine with no specialized features, and makes use of the commercially available signature databases provided by Trend Micro.

Chapter 5

ESX Anti-Virus

5.1 Development History

The server daemon and the client originally had an INET Socket [21] [24] based communications medium. It effectively initializes the core scanning engine and then continuously waits for client connections. Upon a connection request from the client, the server spawned a detached POSIX thread that shares a potion of memory with the parent process, which contains the mapped signature database (denoted "Pattern File" by Trend Micro). This memory sharing between the parent process and each thread was and still is one of the major memory-saving design components of the prototype.

In the current design, the use of INET socket based communication has been discontinued. A new communication protocol is being designed that is based upon a shared memory model. Consequently, many of the mechanisms designed in the prior model need to be updated to support the shared memory model. Most notable of

these updates is the configuration manager and the scanned file caches.

There are a few optimizations built into the server daemon that are switched on and off via the configuration manager, which essentially is a file that instructs both client and server to take or not to take certain logical paths. The most notable of these optimizations is the "already scanned" cache. In this implementation, any file from one virtual machine that has been positively identified as legitimate will have a checksum computed against the file, and that value will be stored in a hash local to the server daemon that has the file name as the key to the checksum value from within the hash.

When a client submits a file to the server daemon for checking, the server looks this filename up in its already scanned cache to see if it has already positively been scanned to avoid repetition. This mechanism will impact the overall system performance, although a thorough performance study had not been conducted, and will not be conducted until further development in the new model has taken place. The intentions of this are to avoid scanning the same file across multiple virtual machines, and will increase the overall performance of the system as more virtual machines are included in the system.

The configuration also had a component that will communicate to the client that it should behave a certain way. If there is an updated configuration file for the client, the next time that it attempted to connect to the server daemon, the server daemon instructed the client to take this new configuration file and refresh its local logic paths.

The prior client component of the prototype had a few sub-components. The first of which is the actual user-level code that serves as a front end to the application as well as a communications medium to the second component, the kernel level system filter driver. Previously there was actually 3 separate implementations of the client model. The original model was preliminary and resembles more of an On-Demand scanning model, in which files are explicitly defined to be scanned, queued, and sent to the server serially. The model has no interaction with the system filter driver.

The second model of the client makes use of the Trend Micro based system filter driver, simply denoted Filter. Filter serves as the bridge between the Trend Micro virus scanning API and the user level application, and calls are made to Filter via the IOTrap statically linked library. This model is currently operational and performs all the duties necessary to maintain data integrity. There is an additional functionality, that of mapping a file to main memory, to support the next functional change in the overall system of the prototype (discussed later).

There are a few issues with this implementation: The first of which is how the user application is interfaced with the filter driver. Because the IOTrap libraries only provide a means to control the Filter driver through the virus scanning engine, the client model must include the scanning engine in kernel space, even if scanning is not performed. This consumes extra unnecessary overhead, mainly in the context of memory consumption. Having a scanning engine running in the kernel space of the virtual machine creates undesired redundancy, since the scanning will actually be offloaded to the server daemon.

A third version of the client was in development, one that makes use of a variation of an example driver contained in the Microsoft Driver Development Kit, called FileSpy. FileSpy is an example tracer driver that logs and reports all the specified file transactions that are executed in the file system. In the Windows Driver Model [36], the fundamental communications primative is the I/O request packet (IRP) [49]. An IRP is essentially a kernel structure that describes I/O request from the I/O manager of the windows kernel. The relevant IRPs in this context are the IRP_MJ_CREATE [2] and IRP_MJ_CLOSE [1]. IRP_MJ_CREATE is sent when a new or existing file, directory, device, or volume is being opened. The IRP_MJ_CLOSE IRP is sent when the reference count of the file, directory, device, or volume has reached 0, signifying that the kernel I/O manager has completed operation on the file, directory, device, or volume in question.

The client call code was injected into the user portion of this application, and upon every IRP file transaction (ignoring FastIO cached file transactions), a call to the server daemon was made. One of the main issues with this implementation currently is that data integrity is not preserved presently. In this model, the file access is not blocked until positively verified as legitimate code. This could potentially lead to a user accessing a file and not having the virus scanning system intervene until after the code has already been executed.

One of the significant developmental steps involved in progressing is the implementation of a stable IO blocking system. In the source of the FileSpy kernel routines, this involves putting a sort of semaphore on a file access, and would be associated with the events IRP_MJ_CREATE and IRP_MJ_CLOSE, for when a file is created

and when a file is modified.

The idea is that when the driver interprets an IRP_MJ_CREATE I/O event, a new kernel event is created that represents the portion of the code that would perform the call to the scanner. Then, the function KeWaitForSingleObject(...), which will put the current thread (in this case the I/O transaction) into a wait state until the given dispatcher object (in this case the scanning event) is set to the signaled state. Once the server daemon has sent back a positive response, this event is set to the signaled state, and the kernel will continue the thread execution.

If there exists a straightforward way to decouple the scanning engine from the trend micro filter driver, then development in the second variation of the client will continue and the third variation will be displaced. This requires correspondence with partners at Trend Micro. It is also possible that Sophos could provide such a driver to offer similar functionality. However, in either case, such routines would not give full control over the behavior of the driver, as they will not provide source. This is another reason why the FileSpy implementation has been chosen.

5.2 On-Demand System Results

Shown in figure 5.1 are some of the results for the on demand scanning model. A workload of 70 binary files containing approximately 263 MB data was processed by the system. The results of the average time to scan across 5 trials are shown in figure 5.1.

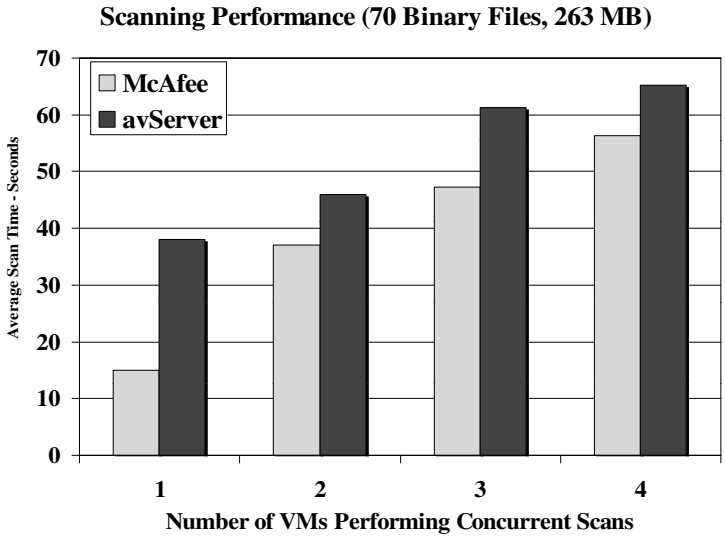

Figure 5.1: Scanning performance of McAfee Enterprise 8 scanner and prototype system against a wall clock (On-Demand Scanning)

This data presents two cases: the average scan time of the prototype on-demand model and the average scan time of the commercial McAfee 8.0 Enterprise on-demand scanner. The data on the X axis represents the concurrency level (i.e. the number of virtual machines concurrently running a scan). The data suggests that the McAfee application takes less time to scan the files across all cases. It is noteworthy that the gap between the time taken by the prototype versus the commercial scanner decreases as the level of concurrency goes up. The main contribution to the degredation in performance in the prototype on-demand model is the need to make a copy of the file and send it the server daemon. Eliminating this portion of the execution will result in much better performance from the prototype.

Memory consumption is another concern when executing anti-virus software. The

prototype model provides a good opportunity to significantly reduce the amount of memory consumed by the execution of anti-virus software through the consolidation of resources. One of the major memory consumers in the anti-virus model is the signature database, consuming anywhere on average from 20 to 40 MB, depending on the implementation. The size of these signature databases will continue to grow with time as new viruses are discovered and their signatures contained in the databases. In the ESX server, running multiple machines with anti-virus software means multiple instances of signature databases, which consumes significant amounts of memory. The prototype consolidates this memory by sharing the signature database, among other configuration parameters, amongst the virtual machines that are connecting to it. Measurements were taken, comparing the McAfee Enterprise 8.0 real time anti-virus package to the prototype execution. These measurements across various levels of concurrency are shown in figure 5.2.

The first level of concurrency, 0, is the case where no scan is actually taking place, and the anti-virus system is simply waiting for a request with the signature database and configuration parameters loaded. Because of the lightweight nature of the prototype design, this case consumes less overall memory than the enterprise package. The gap increases as you increase the level of concurrency. In the prototype system, for each new virtual machine introduced, a detached thread is created that consumes roughly 10 MB memory for stack space of the thread.

In order to characterize the performance of an on-access virus scanner, it is necessary to generate a workload that models something of typical use. The benchmark employed for testing is Business Winstone 2004, running a standard sequence of 5

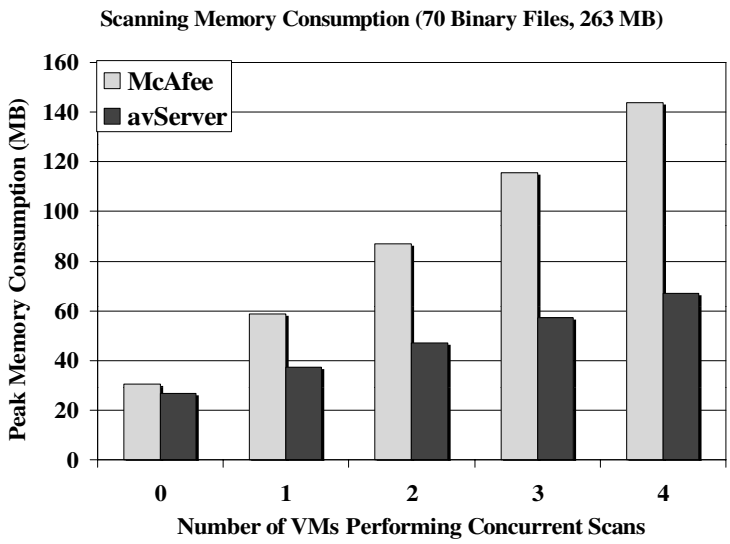

Figure 5.2: Memory consumed by entire system attributed to Anti-Virus execution (On Demand) of prototype system and McAfee Enterprise virus scanner

tests. The top score from the 5 tests is taken as the score, only under the condition that the variance amongst the 5 trials is less than 5%. If variance is higher than 5%, then the sequence of tests are repeated.

There are several different sets of results shown in figure 5.3. All data was collected on a single virtual machine with 1GB main memory running in the Windows XP operating system. The physical system is a HP xw9300 workstation with two AMD Opteron 252 processors, with an overall system memory of 4.5GB.

Descriptions of the cases are as follows: Base represents the case where the virtual machine is running no software in the background and the benchmarks are performed. McAfee is the case where the commercially available McAfee VirusScan 8.0 real-time

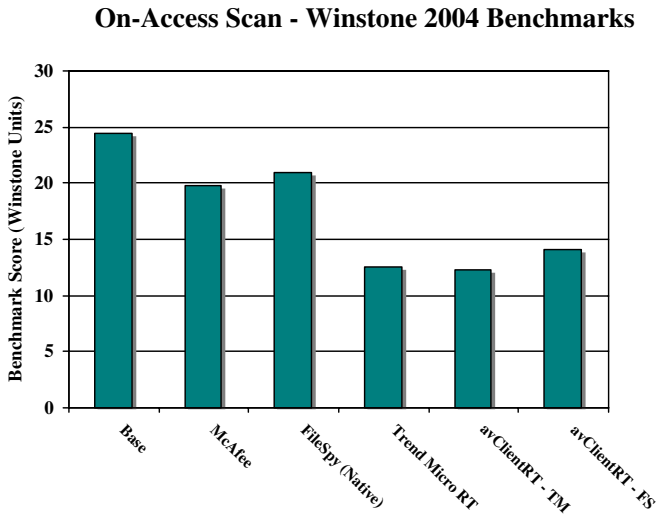

Figure 5.3: Comparison of various cases of realtime execution against the business winstone 2004 benchmark

scanner is being run. FileSpy is the case where no virus scanning functionality is implemented in the example Microsoft FileSpy driver, but the filter is running in the background, simply logging the files. This case was recorded as grounds for proving that the basic functionality of FileSpy does not produce a significant amount of system overhead. TMRT is the example real-time scanned provided in the Trend Micro API, where scanning is performed locally, in kernel space. AvClient-TM and AvClientRT-FS are the second and third implementation of the client, respectively.

These results show that when running only one instance of an anti-virus engine, the commercial McAfee scanner outperforms the other systems that actually perform virus scanning duties. As stated above, the purpose for showing results for FileSpy

and TMRT is to demonstrate how these filter driver systems work by default, performance wise.

The results suggest that FileSpy does not produce a significant amount of overhead in its duties, whereas the TMRT implementation does. This is the result of executing the scanning routines in the kernel, where the most binary translation takes place in the ESX architecture; the more code that must be translated, the faster the translation cache fills up, and consequently, the poorer the overall performance. This poor performance is also seen in the avClientRT-TM implementation. Eliminating the need to include the Trend Micro virus scanning API from the kernel code space would increase the performance of the execution of the client side of the system from within a virtual machine.

The avClientRT-FS implementation performs better than the TM based implementation, however, the comparison is unfair because currently no I/O blocking takes place in the driver for FileSpy, which has a significant impact on performance. The implementation of I/O blocking within the driver will most likely yield similar results to the TM implementation.

One of the most significant roadblocks of the overall system performance is the prior communication medium used between the client and the server. The use of INET sockets as well as making a copy of the file between the two entities generates a significant amount of overhead that does not have to be paid in the traditional on-access model.

5.3 ESX Anti-Virus Prototype

A major objective of this thesis is to allow computer systems that utilize software-based anti-virus applications, to run more efficiently. Previous studies [20] have taken measures to quantify how much overhead is introduced by the execution of a real-time anti-virus scanner and have indicated that significant amounts of pressure is place on not only the central processor, but the memory subsystem. Many solutions have been proposed to potentially offset or alleviate this overhead; however few have actually been implemented.

The issue of pressure on the physical system becomes more immense in the ESX server environment, where multiple virtual machines are executed on one physical machine, and each virtual machine running the windows operating system requires its own instance of an anti-virus program.

In the particular case of the VMWare ESX server, there is the issue of binary translation to consider. Since most commercial anti-virus packages will add a system driver and scanning code to the guest kernel space, this places more pressure on the translation cache of the virtual machine monitor.
The translation cache is responsible for intercepting all system calls within a guest virtual machine and translating certain instructions to their user-level instruction counter part. The binary translation that takes place introduces a significant amount of overhead. Measurements taken from commercial anti-viruses using 1000 binary file workload demonstrate the increased amount of pressure on the translation cache. These measurements are shown in figure 5.4. Alleviating this overhead is a matter of offloading as much code as possible from the guest kernel.

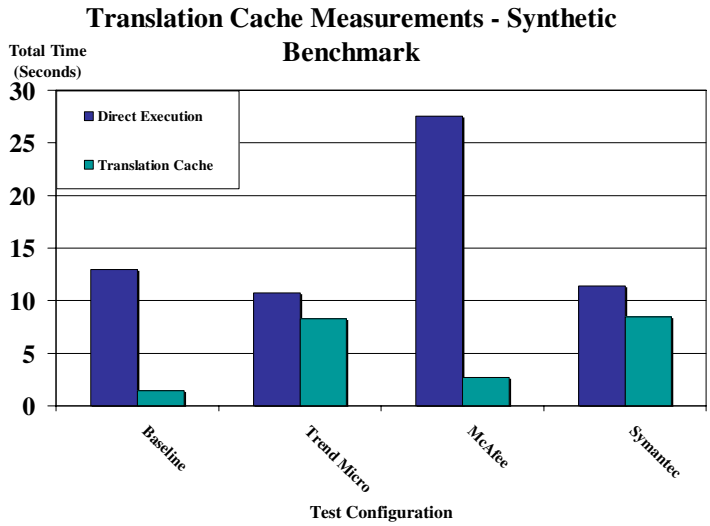

Figure 5.4: Representation of the amount of increased pressure exhibited on the translation cache by the execution of various Anti-Virus packages due to increased guest kernel level activity. The workload used is discussed further in section 6.1

The purpose of this thesis is to take one step forward in addressing the growing issue of real time anti-virus execution based overhead in the context of application consolidation. The ESX architecture provides the ideal environment for a distributed real-time scanning process to be executed amongst all machines, consolidating physical memory consumption and exploiting thread-level parallelism.

The goal of this thesis is to design a prototype of a real time anti-virus application that outperforms commercially available scanners when multiple virtual machines are running and being used in the context of memory consumption and processing power and time, without losing any degree of data integrity.

5.4 General Design Model

There are three major components to the design of the prototype. The first of which is the scanner backend (designated the *Server Daemon*), the client front end (designated the *Client*), and the communications medium between the two.

The server daemon is a POSIX based application that presently runs as a UserWorlds application compiled with UWGlibC-2.2.5, as this library was determined to work properly with the scanning routines. UWGlibC-2.2.5 is an implementation of the POSIX standard glibC-2.2.5 library ported to work with the ESX VMKernel. The core of the server daemon is its scanning engine, which is a Trend Micro designed engine running lower level API calls to minimize execution overhead. This scanning engine performs the duties of the conventional scanning engine with no specialized features, and makes use of the commercially available signature databases provided by Trend Micro.

The client is a lightweight Microsoft Windows system driver that is designed to intercept file I/O calls by rewriting values to the system dispatch table. Conceptually, all a real time anti-virus scanning mechanism does is perform the duties of a regular, on-demand scanner transparently before a new file is being accessed. In the context of the operating system, a "new" file is any file that is either created or opened.

There are two functions that are executed in the windows kernel that perform these duties are the ZwOpenFile(...) and ZwCreateFile(...) functions. It is possible for a driver to "intercept" these calls by overwriting the place in memory that is pointed to by the kernel to make these calls. What the driver does is first, make note of where in

memory the functions that open and create files are in memory. Then, it overwrites that portion of the memory with the address of the driver's "Hook" functions. The actual kernel functions are executed inside these hook functions under their control.

The communications medium is essentially two modules that have been added to the monitor, a subset of the virtual machine manager, and the virtual machine kernel (denoted VMKernel). The client passes information to the virtual machine monitor via virtual registers. This module in the monitor communicates with the VMKernel, which coordinates the remainder of the communication between the guest to monitor and the server daemon.

The system essentially works by having the guest map the file that is being accessed into memory. Then, that memory segment is mapped into the virtual address space of the Server Daemon, which performs the scan using its routines. The Server daemon will then return the results.

An overview of the components involved in the prototype are shown in an abstraction of the ESX server architecture, shown in figure 5.5 . In this figure, we depict N virtual machines running on the ESX server. Each VM is running Windows XP, and has the Client installed as a lightweight system driver, depicted in the Windows kernel partition of the VMs. The AVClient will communicate with the VMM, which will in turn communicate with the VMKernel. The VMKernel then will pass data to the AVServer application and wait for a response. Upon response, a message is passed back from the VMKernel through the VMM to the guest kernel.

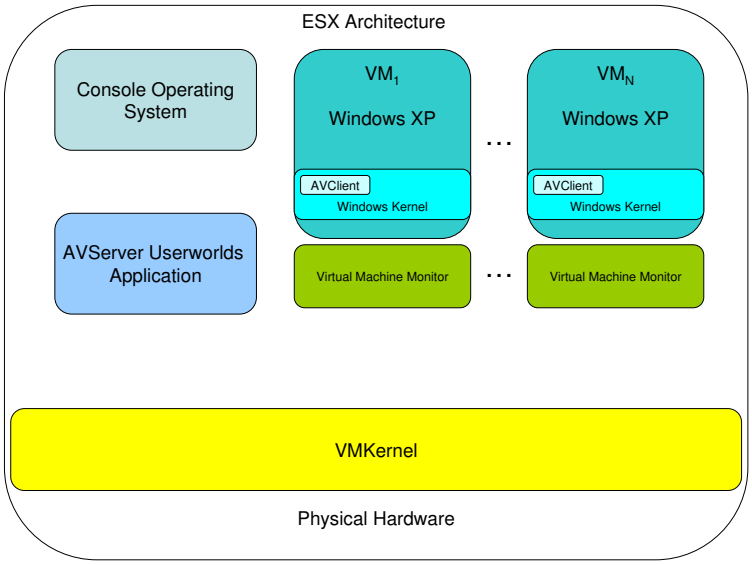

Figure 5.5: Abstraction of the ESX server architecture, summarizing the entire AV-Prototype configuration

5.5 AV Client

As stated before, all the client needs to do to behave like a standard on-access virus scanner is to run the scanning routines on files that are actively being accessed from user interaction. The two functions in the Windows kernel that are executed from this behavior are the OpenFile() and CreateFile() functions. The idea is to trigger a scan whenever a file on the filesystem is accessed using either of these functions. This calls for a need to inject code in the execution path of the OpenFile and CreateFile functions. This can be done by adding hook functions into the windows kernel that perform the necessary tasks to invoke a scan before allowing the user to obtain control of the file being accessed. The way to do this is to modify values contained on the system service table in the Windows kernel, which is discussed in more detail in section 5.5.1.

At the heart of the client is code that will map the file being accessed to guest physical memory. When the client has successfully mapped the file into memory, it must instruct the VMKernel that a file is ready to be scanned so that VMKernel can invoke the proper scanning routines. This instruction to run the scanning routines from inside the guest is referred to as *Backdoor*, and it is the communications medium for the guest to talk to the VMKernel, discussed further in section 5.5.2.

In order for the VMKernel to be able to invoke the proper scanning routines, it needs certain information from the guest. There are two buffers in the client that are responsible for this information. The first buffer, referred to as the *File Buffer*, is the buffer containing a mapping of the file that is currently being accessed by the windows Kernel. This bufer has been fixed to a size of 4MB for two reasons: (1)To prevent memory exhaustion inside the guest kernel, and (2) so that the list of all physical pages numbers associated with the file buffer can be placed on one physical page. This buffer that maps to one physical page containing the list of all page numbers associated with the file buffer is referred to as the *Driver Buffer*. Both of these buffers are initialized in the `DriverEntry` function of the client, and their use is discussed further in section 5.5.2. A higher level abstraction of the AV client is depicted in figure 5.6

5.5.1 System Service Table

The system service table is responsible for keeping track of where in memory any particular native Windows API function is, with respect to the dynamic library

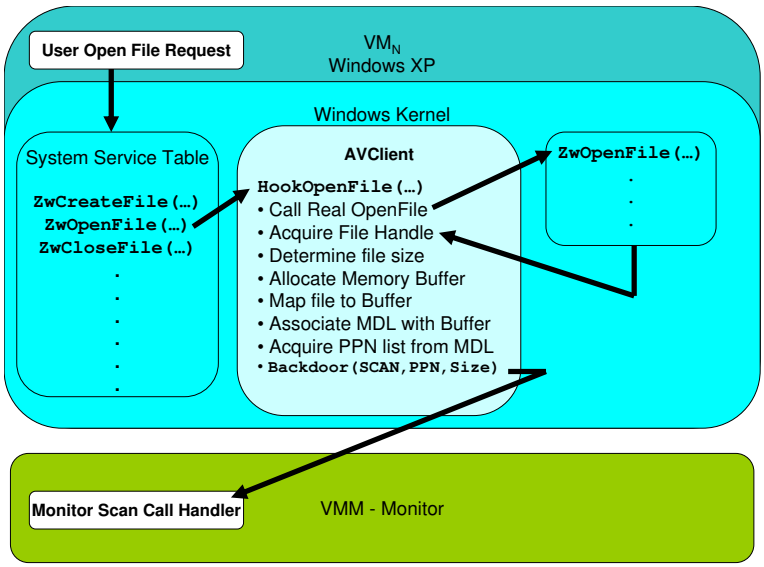

Figure 5.6: Abstraction of the AV Client summarizing step by step the process that the client goes through to invoke a scan up to the monitor component

`NTDLL.DLL`. In the design, the idea is to replace the pointers to the API functions with pointers to our own functions, while keeping track of the pointers to the native API functions (because we still want to be able to execute these functions).

One issue with Windows XP is that the system service table will be write protected. This is a security mechanism that Microsoft has implemented to prevent third party code executing in critical Windows kernel structures [16].

The way to get around the limitation of having a write-protected system service table is to effectively double map the service table in kernel memory. This is done by mapping the `KeServiceDescriptorTable` to alternate physical pages via use of an MDL describing the structure, acquiring and locking the physical pages associated

with the structure, and finally mapping this structure to an alternate system virtual address. When the AV Client system driver loads, this is one of the first operations it performs, allowing data to be written to the service table.

The next step is to have pointers to functions defined that will replace the real `ZwOpenFile(...)` and `ZwCreateFile(...)` functions on the service table [36]. Because we want to be able to run the real functions, we need to define function pointers that have the same parameters as the real corresponding functions, so we can pass these parameters to the real functions in order to preserve the normal execution of the Windows kernel.

Once these pointers have been established, the only thing left to do, aside from writing the values of the pointers on the double mapped system service table, is to acquire the placeholders in memory for the original service functions. This is done by analyzing the stubs in the `ntdll.dll` file. On X86, all the stubs start with the instruction `MOV EAX, ServiceIndex`, where ServiceIndex represents the pointer for the real function. This is a 5 byte instruction, where the last 4 bytes represent the pointer to the real function. Establishing this, acquiring the pointer for the real functions is a matter of reading this line of code offset by +1.

Once the real function pointer is acquired, the hook is simply a matter of exchanging the pointer for the real function with the pointer for the function containing the injected scan code. This is done using the Windows kernel function `InterlockedExchange(...)` [36]. This whole process is done for the kernel functions `ZwCreateFile` and `ZwOpenFile`, and, as a result, whenever either of those functions are called by the Windows kernel,

the functions with the injected scan code will be executed in their place.

The client implementation has its own mechanism for I/O blocking. When a system call is made in the kernel to open or create a file, a handle to that file will be passed into that function, where it is updated, and then output from the function. In the event that a negative response is generated from the server and file access must be denied, the hook function can refuse to return that file handle to the calling function, by setting it to null, effectively denying access to the user and/or calling functions.

5.5.2 Invoking the Scan

For each time a file is accessed, i.e. every time a call is made to `ZwCreateFile` or `ZwOpenFile`, a procedure is invoked upon the file for which is being called upon. This procedure, which is similar in both the cases of `ZwOpenFile` and `ZwCreateFile`, is outlined in figure 5.6.

The routine begins by running the real corresponding function. This is done to acquire a file handle on the file in order to gather information about the file. We first determine if the request is a file or a directory. This is done using the `ZwQueryInformationFile(...)` with the `FileStandardInformation` structure [36]. Running the query information on the file with the standard information structure allows us to get two pieces of information relevant to the scanning procedure. First, it gives us a boolean value as to whether or not the file request is actually a file and not a directory. Second, it allows us to determine the size of the file that is being accessed.

In the event that we know we are working with a file contained within the filesystem and we have successfully acquired the file size, we can begin the scanning process. We begin by mapping the file into the file buffer, which was allocated at the initialization phase of the driver. This mapping is done via the Windows Kernel function `ZwReadFile(...)`. Next, because the file position is modified upon execution of the `ZwReadFile(...)` function, it is necessary to to reset the position of the file back to 0. If this is not done, the Kernel will think that the position at the end of the file (where the file position pointer is left after execution of `ZwReadFile(...)` is the beginning of the file, and will throw an exception upon trying to read.

Now that we have mapped the necessary information to the file buffer, we can now acquire the necessary information to send a scanning request to the monitor . This requires that we determine the physical pages numbers (with respect to the guest) of the file buffer, and write this list to the driver buffer, which is pre-mapped in the monitor for reading upon initialization of the driver. In order to get this information about the file buffer, it is necessary to use a *Memory Descriptor List*(MDL). An MDL is a structure defined by the memory manager that is used to describe the pages that back a virtual memory range [36]. After proper allocation of the MDL, we acquire the array of physical pages associated with the file buffer via the function `MmGetMdlPfnArray(MDL)`, and write these physical page numbers into the driver buffer.

After successfully acquiring the physical page numbers associated with the mapped file, we can now make the call to the monitor to instruct the kernel to perform a scan on the file. As stated before, this is done using the backdoor interface to the monitor.

We write the command we want the monitor to perform to the virtual register `VECX`, a virtual register designed for backdoor, and we write our parameter, the size of the file, to the virtual register `VEBX`. A sequence of x86 instrutions are then executed to instruct the monitor that there is data ready for it to process. From the viewpoint of the guest machine, this operation, which makes the call to the monitor, will return instantly with the result of the scan. This is because when control is passed to the monitor, the virtual CPU will freeze. What really happens at this point is discussed in section 5.7.

When the monitor has passed data as output from the monitor call, the driver will read the value from this output, which is the result of the virus scan. If the result is positive (i.e. there was no virus contained in the file), the file handler will be passed back to the calling function in the Windows Kernel. In the event that the result is negative (i.e. a virus is contained in the file requested), the file handle will be set to null and returned to the calling function, effectively preventing the user from accessing (reading, writing, or executing) the file. The remainer of this process is depicted in figure 5.7.

5.6 AV Server

As stated before, the server side of the prototype is a POSIX style application that is run as a UserWorld application, compiled with UwGlibC-2.2.5. The main design aspect of the server is that it will run a single instance of the virus scanning engine

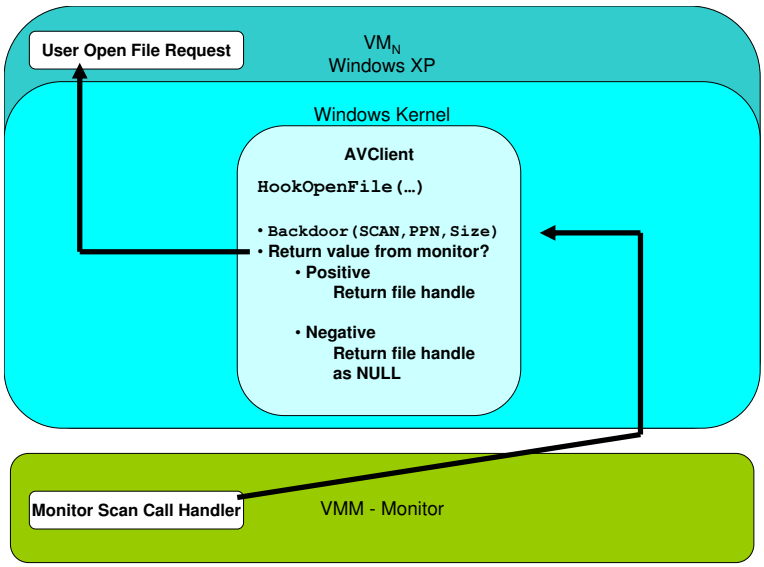

Figure 5.7: Abstraction of the AV Client summarizing step by step the process that the client goes through when scanning results are returned from the monitor.

and share the configuration of the scanning engine amongnst the worker threads affiliated with the server. There are two aspects to the server: the *initialization* phase and the *daemon* phase. In the initialization phase, the scanning engine is configured and the appropriate virus pattern database is loaded. In the daemon phase, worker threads wait to be called upon to perform the actual scanning of the input data. An overall abstration of the server is shown in figure 5.8

5.6.1 AV Server Initialization

The server begins by initializing the scanning engine. This is done in several steps. First, it will create an object that repsents the scanning engine configuration. This object is referred to as the *Virus Scanning Context*(VSC) object. After the VSC has

been created, it is instructed that the path containing the pattern virus (the database of virus signatures) is in the local directory (i.e. the same directory as the executable).

Next, after the VSC has been instructed to look in the local directory for the pattern file, it will read the pattern file and load it into memory. This is only done once, to ensure that the pattern file is loaded into the memory once, as it consumes roughly 30MB of memory. After the pattern configuration is complete, the VSC is instructed to consider all types of files (as it is not responsible for filtering which files to scan and which files not to), including archives.

After the VSC has been instructed to consider all file types, the action that the scanning engine will take on the file is defined to take no action. This is done because if the scanning engine attempts to write to the mapped memory from the guest, when control is returned to the guest kernel, it will detect a change in the physical memory and consider it corrupted, causing a kernel panic.

Finally, a callback function is defined in the VSC for the scanning engine to perform a custom action upon scanning the file. This is done so that a pointer to the callback function can be passed to the worker threads so that when a worker thread performs a scan, it knows what to do to properly send the appropriate value back to the guest that requested the scan. The details of this function are discussed further in section 5.6.2.

The remainder of the initialization process involves configuring the worker thread pool. We determine the number of physical CPUs present on the ESX server by making a call to the VMKernel. The number of worker threads spawned corresponds to the number of physical CPUs present on the system.

The server will then prepare a data structure containing all the necessary information

to pass to a worker thread. The structure passed to the worker threads contains all the data necessary to independently perform the scan. This structure has its own thread ID (discussed further in the Synchronization section) and the VSC for sharing the configuration of the scanning engine.

Finally, after all the thread data has been prepared, joined threads are created corresponding to the number of physical CPUs in the ESX server. The remainder of the work is done in the worker threads, discussed in the next section.

5.6.2 AV Server Daemon

Unlike most thread pool models, the worker threads in the prototype are responsible for collecting their own input data. The VMKernel is responsible for dispatching a worker thread, not the main process.

Each worked thread begins by reading the data passd in by the `pthread_create(...)` function; namely, its thread ID and the location in memory of the VSC created in the main process.

The worker thread then checks to see if the VSC is valid and properly configured. If there is a problem with the VSC, no scanning can take place. Consequently, the server process will halt, as well as all VMs equipped with the AV Client.

In the expected event that the VSC is valid, the worker thread will enter an infinite loop which is responsible for the remainder of the scanning process. Since the VMKernel is responsible for dispatching the worker threads, each worked thread will

make a call to the VMkernel, called `VMKernel_AVWaitForScan(...)`, that instructs the VMKernel that the calling worker thread is available for work.

This function passes the necessary data into the worker thread, while instructing the VMKernel what is the thread ID which is passed in as an input parameter to the `VMKernel_AVWaitForScan(...)` function (for synchronization purposes, discussed further in the synchronization section). When the VMKernel is ready to dispatch the worker thread, it will pass into the worker thread the size of the file to be scanned as well as a pointer to where the file has been mapped into the AVServer world virtual address space (Discussed further in the synchronization section).

After the VMKernel has given the worker thread the necessary information to perform the scan and dsispatched it, the worked thread will continue forward to create a *resource object*. A resource object is defined in the Trend Micro Anti-Virus API as the input to the function that invokes the scanning engine.

There are several ways that the scanning engine can be invoked. Because we are instructing the scanning engine to analyze a segment of memory, we must define the segment as a resource and consequently invoke the enging using `VSScanResource(...)`.

The resource created is defined to contain the segment of memory beginning with the base pointer passed from the VMKernel offset by the size of the file. This resource object created serves as a handle to the segment of memory that we wish to scan with respect to the scanning engine.

The scanning engine is then invoked with the VSC (from the main process) for the configuration of the scanning engine and the resource defining the point in memory to scan as inputs. As mentioned earlier in the inistialization section, there is a callback defined to extract the output of the scanning engine from the VSC.

The callback function will analyze the findings of the scanning engine and assess the findings as either positive (no malware found) or negative (malware found). This callback has a boolean value passed in by reference to the `VSScanResource(...)` function. Once the scanning is complete, this value determined within the callback function is available to the local scope of the worker thread.

Finally, the worker thread will instruct the VMKernel that the scan is complete by calling the system call `VMKernel_AVFinishScan(...)` while at the same time passing the boolean scan result into the VMKernel in order for the guest to be notified to take the appropriate action.

5.7 Synchronization Model

The synchronization model is the heart of the whole prototype. As stated earlier, communications between the AV Client and the AV Server is accomplished via two add on modules, one to the Monitor, and the second to the VMKernel. The monitor code serves as a bridge between the guest VM and the VMKernel, and the VMKernel is responsible for the bulk of the synchronization. A visual abstraction of the synchronization between the client and the server is shown in figure 5.9

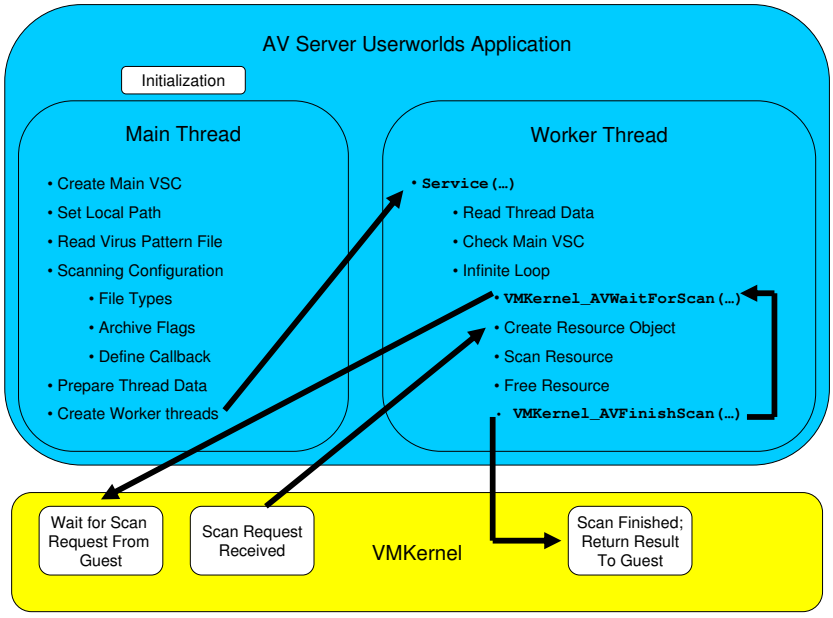

Figure 5.8: Abstraction of the AV Server illustrating the initialization process as well as the task of a worker thread

5.7.1 Monitor

Guest communications to the VMKernel are established through the monitor. Discussed earlier in the client section, when a guest VM requests a scan to be performed on a given file, the guest will place data in virtual registers and then invoke a sequence of x86 instructions to indicate to the monitor that data is ready for it to process. The receiving module in the monitor code is a virtual bus implementation called *busmemAV*. BusmemAV contains a mechanism for interpreting backdoor commands (denoted the *backdoor handler*) sent from the guest VM, as well as memory initialization, and a system call to the VMKernel to invoke a scan.

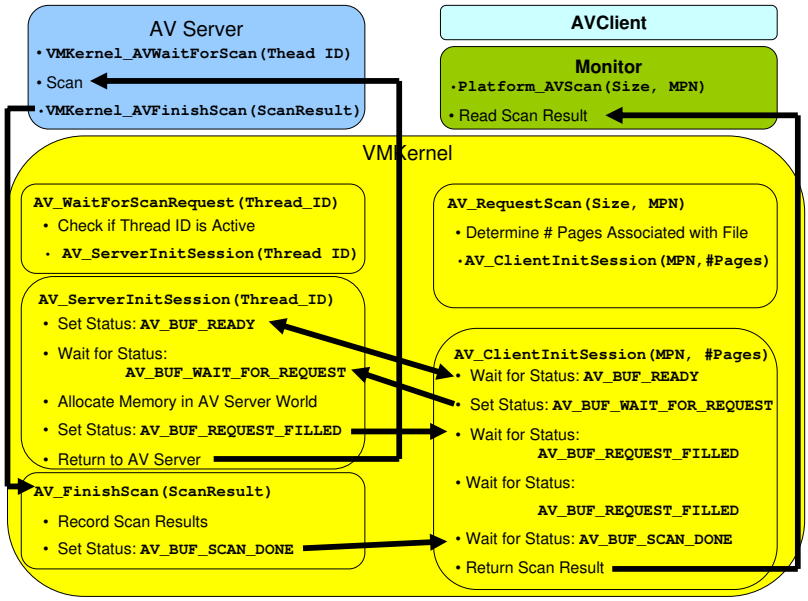

Figure 5.9: Abstraction of syncronization between the AV Client and AV Server via the VMKernel.

The backdoor handler interprets 3 commands from the guest virtual bus. The first command, `AV_BDOOR_CMD_INIT`, instructs the virtual bus to check with VMKernel that the driver buffer - the buffer that contains the list of physical page numbers associated with the file we are going to scan - is stored in VMKernel. This command is typically run when the AV Client is running its initialization code. A variation of this is also run to perform sanity checks on the driver buffer, to ensure that the MPN did not change due to a state change, such as reverting to a snapshot of the VM.

Because the guest passes in the physical page number associated with the driver buffer with respect to the guest, there is a need to convert this physical page number

to a *Machine Page Number* (MPN), which is the physical page number with respect to the VMKernel. This translation is performed in the virtual bus itself. When an initialization call is made to busmemav, it will begin by translating the physical page number of the given driver buffer to a *Bus Page Number* (BPN), which is a physical page number with respect to the monitor code, as the Virtual Machine Manager (which contains the monitor) is responsible for managing a range of physical pages for a given VM world.

Once we have the BPN of the driver buffer, we compare this value to the value stored in the global VMKernel structure (which will already contain both the MPN and the BPN). If the BPN of the given driver buffer matches the BPN of the driver buffer stored in VMKernel, then the driver buffer page number is up to date and no further action is taken; Our initialization is successful. In the case where the BPN does not match what is stored in the VMKernel, the initialization function must update the BPN and MPN of the driver buffer in the VMKernel. We also pin this MPN as active in the VMKernel, in order to reference it. This is done through the `BusMem_TranslateBusFrame(...)` function.

Upon successful execution of this function, our initialization is successful, and the VMKernel is ready to accept new scan requests. Initialization is done within the monitor because it will associate a unique driver buffer with the present world within VMKernel. This is because each VM has its own VMM, and consequently its own monitor. This is necessary to introduce the concept of sessions to the synchronization model, for supporting multiple VMs making calls to the AV Server.

Another command associated with the initialization command is the cleanup command, `AV_BDOOR_CMD_CLEANUP`, which is used to instruct the VMKernel that a guest has gone offline and that any data associated with it should be freed. Essentially the command will unpin the driver buffer MPN. The VMM takes care of the rest.

The third and most important command for the Virtual Bus is the actual scanning command, `AV_BDOOR_CMD_SCAN`, which is responsible for passing in the necessary data the VMKernel requires for scanning as well as indicating to the VMKernel that a scan request has been made from the guest, which passes in as arguments to the command the size of the file. This is the only necessary parameter as the VMKernel is already aware of where physically in memory the file segment to be scanned resides.

The scan command performs its own sanity check on the driver buffer, similar to the initialization function. Then it will pass the data given (the file size), the MPN of the buffer containing the page list, and a pointer to a Boolean value, into the VMKernel for the scanning transaction. The function responsible for passing this data is `Platform_AVScan(...)`.

At this point, the VMKernel has taken over the scanning process. The virtual bus will wait until the VMKernel returns with the scan result, which will then be placed into a virtual register that corresponds to the backdoor call output in the guest. The guest will then read that value and continue.

5.7.2 VMKernel

The module added to the VMKernel is the heart of the synchronization model. The most important aspect of the synchronization model is the introduction of the concept of session state. The VMKernel is responsible for managing scan requests from all guest VMs, unlike the monitor, which is only responsible for its corresponding VM world. For each scan request, there is a series of states that each session must go through in order to successfully complete a scan. This is kept track of via an array of global data structures from within the VMKernel. How this data structure is used, as well as the session state process, is outlined in this session.

From the AV Server point of view, when a worker thread is prepared to perform scanning, it will make a call to the VMKernel indicating such via the `AV_WaitForScanRequest(...)` call. When a worker thread makes this call, it passes in its own thread ID, assigning that thread to the corresponding session. While control is in this system call, a cleanup and initialization process takes place. The cleanup is done initially, where the function makes sure that any memory mapping from a previous instance of a scan is freed.

The process continues by initializing the thread for a new scan request. During the initialization process, the VMKernel first checks the data structure corresponding to the calling thread to see if a scanning session is active. If a session is active, the initialization process will abort. If not, the VMKernel will assign the thread a new session ID and set the state of this session to `AV_BUF_READY`. This process takes place in the `AVServerInitSession(...)` function. The worker thread is now prepared to accept a new scanning request.

On the guest side of a scan request, the function Platform_AVScan(...) from the monitor makes a call into the VMKernel. This function is actually a stub function that makes a call to the VMKernel function `AV_RequestScan(...)`. The monitor passes into the VMKernel as input to this function the mpn of the affiliated driver buffer, the size of the file to be scanned, and a place in memory to store the scanning result.

The `AV_RequestScan(...)` function will begin by determining the number of physical pages associated with the file. The scan request must now wait until a scanning thread is ready for a new session. This waiting occurs in the `AVClientInitSession(...)` function. Within this function, the request will continuously wait until a thread is in the `AV_BUF_READY` state. When a corresponding session has been found to be ready, the client request will be assigning a session ID number and placed in the state `AV_BUF_WAIT_FOR_REQUEST`.

Now the worker thread must wait while the file that is requested for scanning is mapped into the AV Server world virtual address space. Once the file is successfully mapped into memory, the session status is now set to `AV_BUF_REQUEST_FILLED`. The client scan request now simply waits for the scan to be finished by executing the `AVWaitForStatus(...)` function, which instructs the client request to wait until the session state is set to `AV_BUF_SCAN_DONE`.

Now that the session state has been set to `AV_BUF_REQUEST_FILLED`, the worker thread that was waiting for work can now take action (from the `AV_WaitForScanRequest(...)` function). The function will carry on, assigning a virtual address in the AV Server

world virtual address range, and returning that pointer to the Server application.

The scan will then take place in the AV Server world. As discussed in the AV Server daemon section, when a scan has been completed, a second VMKernel system call, `AV_FinishScan(...)`, is called, passing into the VMKernel the scanning result. In this function, we begin by setting the session state to `AV_BUF_SCAN_DONE`. This will allow the client scan request function `AV_RequestScan(...)` to read the value of the scanning result and pass it back to the monitor, which in turn will pass it to the guest kernel.

The `AV_FinishScan(...)` function will then wait for the `AV_RequestScan(...)` function to set the session state to `AV_BUF_ACK_SCAN_DONE`, and will then carry on to clean up all the memory mapping done for the scanning process, and, upon success, will finally set the session state to `AV_BUF_FREE`. This indicates that the worker thread is free to be assigned new work, and our entire scanning process is complete.

Chapter 6

Results and Analysis

6.1 Experimentation Setup

The prototype is designed to be built in the ESX version 3 model. The base configuration for prototyping is ESX server build 23447. For the design of the prototype, the main changes that were made to the server, aside from the addition of a UserWorlds application for scanning, is a module that sits in the monitor which is responsible for intercepting AV-related guest backdoor calls.

The VMKernel also has a module built onto the default configuration that is responsible for synchronizing the data passed into the vmkernel from the monitor with the Server Daemon. Aside from these modifications, all the experiments are performed on ESX build 23447, in development mode, using an OPT build directive, signifying that no debugging symbols were compiled into the VMKernel, as well as no kernel logging, to maximize performance.

The hardware used in this experiment is an HP wx9300 server. The wx9300 has two Opteron 252 processors (single core) clocked at 2.52 GHz. The server machine has 4.5GB main memory, and two hard drives (as is the conventional configuration for ESX running in development mode), an 160GB 7200RPM IDE drive containing the ESX COS and source, and a 300GB 15000RPM SCSI drive containing the VMs.

Two separate benchmarks were used. To model a realistic user workload on a VM, the Business Winstone 2004 benchmarking suite was used. This benchmark is a scripted benchmark which essentially mimics the behavior of a user interacting with various office-related applications such as Microsoft Word, Excel, and Access. Measurements are taken based on time intervals, and an overall score in "winstone units" is derived from those timing intervals.

Since scalability is one of the criterion of the prototype, it is important to run testing on multiple machines. Unfortunately, a scripted time-based benchmark such as Winstone will not function correctly on multiple virtual machines running concurrently within the ESX server. This is due to the fact that running concurrent VMs will result in stressing the physical hardware which in turn results in greater clock drift and less consistent interrupt rates delivered by the VMKernel [27]. This causes inconsistent results and test failures as certain operations in this timed environment may expect certain operations to perform faster (or slower) than they experiences in a concurrent VM environment. Because the concept of time is experienced differently inside a VM, the results of the winstone benchmark are only shown in the case of 1 VM running as a demonstration of how well the configurations work with a realistic workload.

As a result of this timing issue and the inability for the benchmark to measure the performance of concurrent VMs running, a second synthetic benchmark was designed to stress the I/O system of a guest VM. A C routine was prepared to mimic the behavior of the unix command *grep* in the sense that it will search a file for a given byte sequence. Unlike grep, this routine will not do anything if this byte sequence has been found, it is merely a method for stressing the I/O system within the guest and triggering the real-time anti-virus scanning mechanism.

A supplemental workload was prepared with the synthetic I/O stressing benchmark ; a collection of 1000 randomly sampled windows-based binary files with the only criterion being that the file is in fact a binary file (executable, dynamically linked library, or kernel code) and that it is less than 4MB in size. The benchmark will open all the files in the workload and do a bytewise traversal for a given pattern as described in the source code, shown in the appendix.

Each virtual machine is configures as a uniprocessor system with 256MB main memory. The software on each VM is the Microsoft Windows XP operating system, with service pack 2 installed. Each instance has every update package installed as of 10.23.2006. The only other software installed on each system outside of the operating system and the updates varies based on the testing case.

There are two main testing setups. The first setup, winstone, contains as additional software the winstone benchmark and any additional software that conforms to the specifications of each configuration, discussed below.

The second setup, the synthetic I/O stressing benchmark, contains as additional software the binary of the compiled c routine (shown in the appendix) as well as the workload, which is 1,000 binary files, each of which is less than (or equal to) 4MB. Thetotal size of the workload is approximately 670 MB.

For each setup, there is 5 configurations corresponding to different cases of anti-virus scanning mechanism. The first, denoted *Baseline* in all data presentation, is the baseline configuration in which no additional anti-virus software is installed. This configuration is considered as a point of reference for all other configurations and consequently should outperform all other configurations in all cases.

The second configuration, denoted *Trend Micro* in all data presentation, represents the case where the Trend Micro PC-Cillin version 8 is installed with the most recent patterns file (virus signature database) as of 10.23.2006.

The third configuration, referred to as *McAfee* in all data presentation, is the case where the McAfee Enterprise 8.01 virus scanning engine is installed. Because this package contains more features than a signature based scanning engine (such as a firewall, buffer overflow detection, etc), all of the features of the security suite, with the exception of the real-time virus scanning, have been disabled for a fair comparison.

The fifth and final configuration is referred to as *AV-Prototype* in all data presentation. This configuration is the case where the proposed real-time anti-virus prototype catered to the ESX architecture is configured. This entails, as stated before, the installation of a lightweight system driver that maps files to memory and makes the

appropriate call to the monitor. It also involves the execution of the userworld application avserver, which is the Server Daemon. The only variance in the configuration of the server daemon execution is the number of threads allotted to the avserver world. These threads are the worker threads which are responsible for scanning a representation of a file (a segment of memory). For all cases the number of worker threads associated with the avserver world is defined to be the number of physical CPUs, which in the case of the testing hardware used, is 2. Since the avserver world is based on a trend micro scanning engine, it's signature database is also trend micro based. The signature database used by the server daemon is the same as that of the Trend Micro pattern file.

6.2 Results

Performance measurements were taken using two separate benchmarks. The first benchmark, Business Winstone 2004, provides its own mechanism for scoring based off of timing checkpoints. Winstone will assign an overall score based off of the highest score through the number of runs as specified by the user. For example, all the testing was done using 5 trials. Winstone will say the overall score is the highest score in the 5 trials. The data presented in 6.1 does not use the method, instead, will take the average value across the five trials, given the variance across the trials is less than 5%. If the variance is higher than 5%, the results are discarded and the benchmark is executed again. The results for the benchmark are shown in figure 6.1.

We see from the data presented in figure 6.1 that our baseline configuration, as suspected, will perform the best as it is not performing any anti-virus scanning duties. The AV-Prototype performs second best, Trend Micro PC-Cillin next, and McAfee

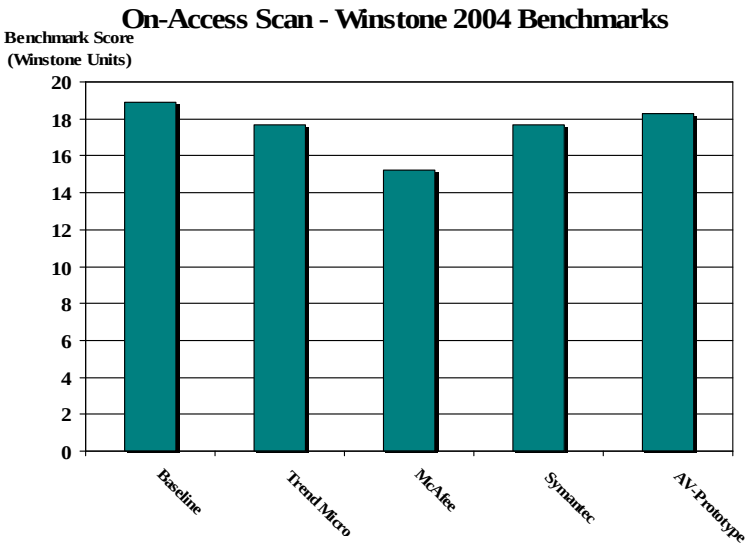

Figure 6.1: Real time scanning performance measurement of Trend Micro PC-Cillin, McAfee, and AV-Prototype using the Business Winstone benchmarking suite - Higher is better

performing the worst. This data suggests that serially (for one VM) that the AV-Prototype only introduced 3.38% of overhead from the baseline configuration, saving 3.28% of overhead with respect Trend Micro PC-Cillin, 16.28% of overhead with respect to McAfee, and 3.07% of overhead with respect to Symantec. Although this benchmarking methodology might be a decent representation of typical use of a machine, it has a few downfalls. First, because it is time based and uses guest VM timing mechanisms, its measurements are not the most accurate. Second, the benchmark does not put a lot of stress on the real time scanning processes, as most of the files that are involved with the benchmark are loaded before the benchmark begins and are not part of the scoring measurement. Finally, the winstone benchmark does not allow us to test the scalibility of the prototype.

As a result of these shortcomings, measurments were taken using the IO-stressing benchmark. As shown in the source code for this synthetic benchmark, a timing mechanism is included in the code. However, because of the timing issues demonstrated from the last benchmark, all of the scores (times in this case) have been taken using a wall clock, sampled using real time and not time perceived by the virtual machine. The results of this second benchmark are shown in figure 6.2.

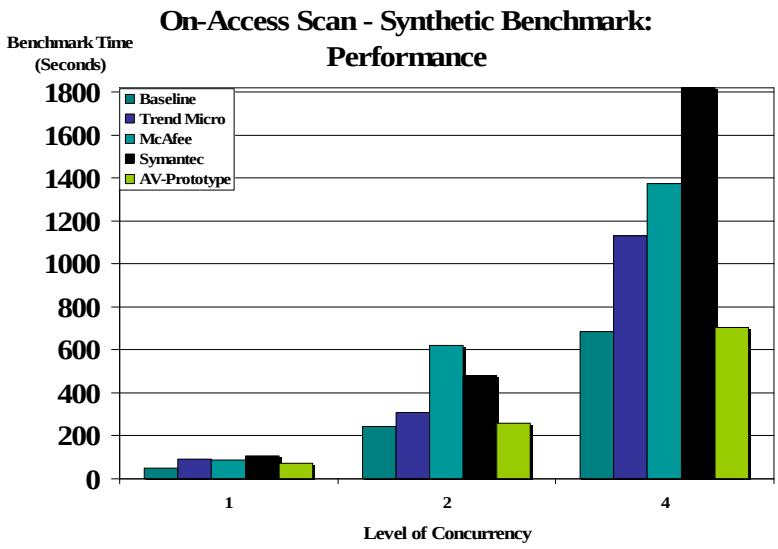

Figure 6.2: Real time scanning performance measurement of Trend Micro PC-Cillin, McAfee, and AV-Prototype using the IO-stressing synthetic benchmark for different levels of concurrency. All scores are based on real time (wall clock) - lower is better.

The times shown in figure 6.2 demonstrate a trend as the level of concurrency increases. For all cases, the AV-Prototype outperforms its commercial counterpart, PC-Cillin, which in turn outperforms the McAfee scanner. The AV-Prototype, for the serial case, will introduce approximately 45% overhead to the IO stressing benchmark, where as Trend Micro , McAfee, and Symantec will introduce 84%, 76%, and 119%,

respectively. In the serial case, the AV-Prototype is saving roughly up to 74% of the overhead introduced by scanning. This gap only increases as you increase the level of concurrency. For the case of 2 VMs running concurrently, almost 68% of the overhead is saved compared to its commercial counterpart, and almost 76% in the case of 4 VMs.

Memory consumption is another concern when executing anti-virus software. The prototype model provides a good opportunity to significantly reduce the amount of memory consumed by the execution of anti-virus software through the consolidation of resources. One of the major memory consumers in the anti-virus model is the signature database, consuming anywhere from 20 to 40 MB, depending on the implementation. In the ESX server, running multiple machines with anti-virus software means multiple instances of signature databases, which consumes significant amounts of memory. The prototype consolidates this memory by sharing the signature database, among other configuration parameters, amongst the worker threads associated with the server daemon.

Measurements of memory consumed by each model were approximated using measurements taken from the windows resource monitor tool to determine how much memory is being saved by implementing the shared memory model. Because of the difference in the structure of the implementations, it is difficult to take exact measurements of memory consumed by each implementation, as even the two commercial implementations differ slightly.

All the measurements are meant to isolate the memory introduced by the anti-virus programs, not the benchmark. Consequently, the results do not include the baseline

configuration, as the amount of memory consumed by anti-virus in that case is 0. For each level of concurrency, the same IO-stressing benchmark was used as for the results shown in figure 6.2. The data collected in 6.3 attempts to capture the *maximum* memory consumed for each case.

For Trend Micro PC-Cillin, the amount of memory consumed was determined by measuring the maximum memory consumed by the associated processes during the execution of the benchmark (and in the case of multiple vms, the sum of these measured values), as well as the spikes in the total kernel memory. It has been observed that there is up to 47MB extra allocated from the Windows kernels paged pool, and up to 37MB memory allocated to the associated processes. All the numbers were determined using the windows tool perfmon.

For McAfee, once again the amount of memory consumed was determined by taking measurements during the execution of the benchmarks. It was determined that McAfee will consume up to 14MB allocated from the kernels paged pool, and that the associated scanning process will consume up to almost 34MB. Again, these measurements were taken using the perfmon tool.

For the AV-Prototype, measurements were more difficult to acquire. In the guest, using windows perfmon once again, it is determined that the system driver consumes at most 4600KB (the majority of which is used for mapping the active file to guest memory) from the kernels non-paged pool. On the server side, measurements are taken using the esxtop tool, which gives active memory use for each world. Measurements show that the main process in the AVServer, the one containing the mapping of the signature database, consumes roughly 41MB , and each worker thread consumes roughly 1MB. Thus, the only change in the amount of memory consumed by

the avserver world is contained in the case where the level of concurrency is increased from 1 to 2, where an extra megabyte is consumed during the execution of the second worker thread. This is because in the case of a single VM, only 1 worker thread can be used at any given time because the guest vCPU is frozen. The only increase in memory for the cases of 2 to 4 are due to the extra memory consumed by the guest driver mappings. The results of all the memory consumption are shown in figure 6.3.

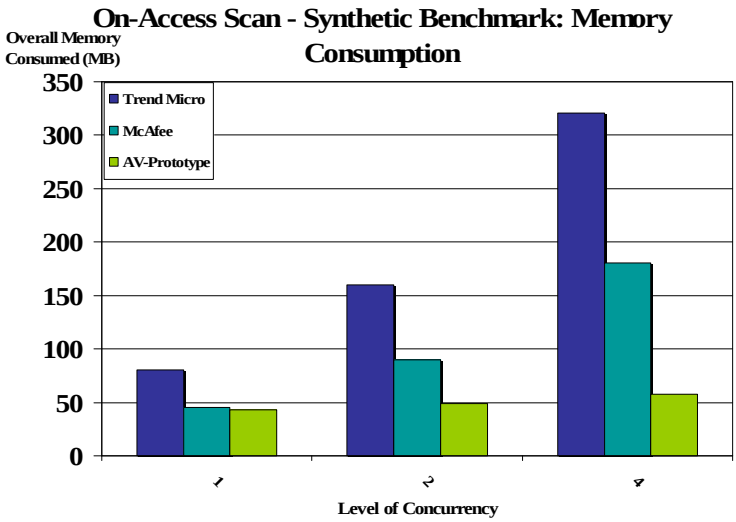

Figure 6.3: Real time scanning memory consumption measurement of Trend Micro PC-Cillin, McAfee, and AV-Prototype using the IO-stressing synthetic benchmark for different levels of concurrency. All measurements are presented in MB - lower is better.

We see from the figure that for the case of a single VM, the amount of memory consumed by the AV-Prototype is similar to that of McAfee. In the case of Trend Micro, there is a lot of extra memory allocated in the kernel space, which makes the Trend Micro implementation consume almost double the amount of memory consumed by the AV-Prototype and McAfee.

Where McAfee and the AV-Prototype consume similar amounts of memory for 1 VM, if the level of concurrency is increased, the amount of memory consumed by the AV-Prototype hardly increases. In the case from 1 to 2, the amount of memory consumed increases by roughly 12% and in the case of 2 to 4, approximately 17%. In the case of the commercial scanners, the amount of memory is being increased roughly around 100% (effectively doubling the memory consumption!). In the case of 4 VMs, the AV-Prototype is saving the physical system over 260MB of physical memory consumed.

As discussed in section 5.3, the amount of pressure placed on the guest kernel increases the amount of activity in the translation cache, which introduces significant amounts of overhead. The amount of pressure placed on the translation cache is decreased in the prototype model because the system driver installed on the guest kernel is as lightweight as possible, and because all scanning takes place outside the guest VM. The same benchmark that was used to collect the data shown in figure 5.4 is shown in figure 6.4 with the AV-Prototype considered.
Observed from the data presented that the prototype introduces roughly 54% of an increase in time spent in the translation cache. The next closest case is McAfee, which introduces roughly an 89% increase in time spent in the translation cache. The only reason this value is so low with respect to the other anti-virus packages (which introduce on the order of 488% increase in time spent in the translation cache) is because McAfee will try to offload some of the scanning duties into a user process, introducing the extra overhead of switching between kernel and user space within the guest VM (an operation that requires time within the translation cache).

The data presented in figure 6.4 also shows where most of the execution is going within

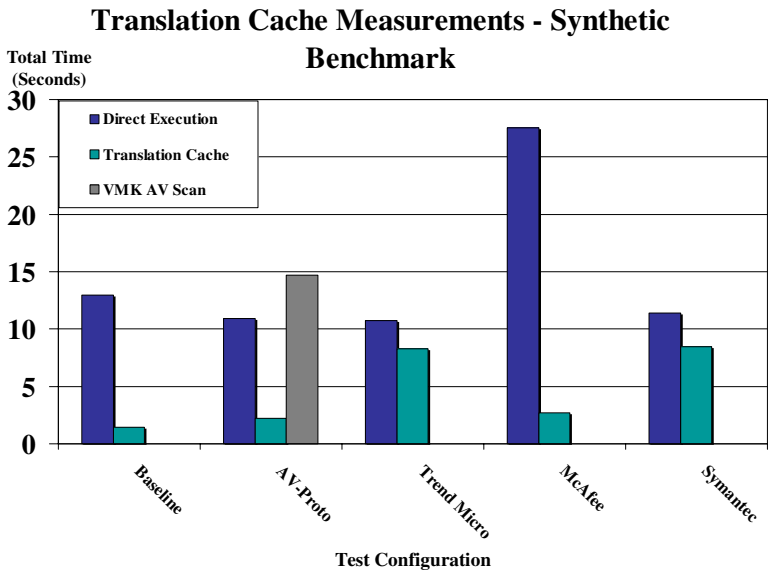

Figure 6.4: Representation of the amount of increased pressure exhibited on the translation cache by the execution of various Anti-Virus packages due to increased guest kernel level activity.

the workload; Namely, the userworld call to perform a system scan. By placing that piece of code that is frequently used in VMKernel, up to 434% of the time spent in translation cache using commercial scanners can be alleviated.

Chapter 7

Summary

In this thesis we have analyzed from an architectural point of view the impact that executing Anti-Virus software on the windows platform has on the overall performance of a system. We have analyzed the innards of the windows kernel to establish various methods for mimicing the behavior of a real-time Anti-Virus in order to adapt a real time scanner to an environment containin a hypervisor. We have taken that motivation to propose a new model for distributed Anti-Virus execution in a distributed environment. The VMWare ESX architecture was discussed at a higher level to demonstrate the possible opportunities to leverage the performance of such applications in a virtualized environment.

Using tools provided by Trend Micro, we have isloated the most significant part of an Anti-Virus package, namely the scanning engine, to create a model for accepting multiple request scans concurrently from various clients, all contained within the hypervisor model. The mechanism for communicating with the scanning engine from within the windows kernel was developed as the scanning client. The synchronization

model for passing data from a windows guests kernel, through the virtual machine monitor, and into VMKernel via virtual registers and memory mapping within the monitor was developed.

7.1 Conclusions

We have shown through experimentation and measurements that by taking advantage of the aspects of the ESX architecture, we can drastically increase the performance of scanning. We have shown that we can reduce up to 76% of the overhead introduced by the execution of anti-virus software in the ESX environment. This is due in mainly to the standard model of the commercial anti-virus, which is heavily dependent on the Windows kernel. In a virtual machine, kernel operations, or system level operations, must undergo binary translation to be executed in a contained environment, which involves putting more pressure on the monitor. Because the AV-Prototype puts as little pressure on the windows kernel, we are experiencing immense performance gains. We are also experiencing less context switching inside the vCPUs and only slightly increasing the world context switching inside the VMKernel.

By exploiting the parallelism introduced by the ESX architecture, we can consolidate the resources of the system. This is shown by both the performance gains and the memory consumption, in particular, as the level of concurrency is increased. We have shown that we can consolidate up to 260MB of memory consumed by anti-virus processes using a shared memory model. This is due to the fact that the scanning daemon shares the configuration, and most importantly , the signature database amongst the worker threads and, as a result, the guest VMs.

7.2 Future Work

The development of the prototype is only the beginning of the possibilities for the design model, which can be applied to almost any service that all VMs require.

One potential area of development with respect to the prototype is the design of the client. The present client model makes use of the system service dispatch table. Hooking these system level functions introduce overhead in the kernel that could be alleviated if the design is changed to a filter driver model, intercepting all relevant I/O interrupts as opposed to related system calls.

Another potential area of development is the design of the server daemon itself. Currently, the prototype executes a userworld application which communicates directly with the VMKernel. This provides opportunities to consolidate management resources through another application that could potentially be contained in the host management software. Another approach to the design of the server daemon is to apply the *Appliance Model* to the execution.

In the appliance model, the server daemon would be executed inside a specialized virtual machine, designed specifically for running the appliance. This offers the added flexibility of being able to contain specific guest VMs and associate them with specific AV servers. The appliance model would also provide user frontends for configuring each scanning engine from within the VM itself.

One of the obvious benefits of development in the appliance model is its portability. Because the scanning routines are linked against a specific version of glibc, there

could be potential recompilation issues down the road, where certain necessary user world glibc modules will not be ported and consequently make compilation as a user world application more difficult. Since the appliance model would be making use of standard glibc libraries as opposed to user world glibc, the compatibility of the design would be increased.

Finally, a set of standardized API could be provided to the development environment of the prototype model. This API could include standard system calls in the VMKernel as well as places in the client, server, and monitor. One of the major benefits of introducing a standard API, aside from ease of additional development, would be to enable the ESX server administrator to choose which anti-virus scanning engine to execute in the AV Server. In other words, it could make the design of the prototype more modular and exhibit more control over the options available to the design itself.

Plenty of other opportunities can be implemented in this model to give anti-virus programs leverage in the fight against malware. For instance, sandboxing, the concept of executing a binary in a contained environment and applying a set of heuristics to determine particular malicious behavior, could be implemented as a specialized virtual machine, managed (and isolated) by the VMKernel itself.

Another higher-level design change that could take place would be to introduce a system of system cleanup. Upon the detection of a particular piece of malware contained within a virtual machine, the guest could effectively quarantine the file in the filesystem and alert the VMKernel that this file needs to be removed. The virtual machine could then be safely powered down and the infected file(s) could be removed

from the filesystem offline.

Bibliography

[1] Irp mj close. http://msdn2.microsoft.com/en-us/library/ms795859.aspx.

[2] Irp mj create. http://msdn2.microsoft.com/en-us/library/ms795806.aspx.

[3] S. Junaidu A. Badhusha, S. Buhari and M. Saleem. Automatic signature files update in antivirus software using active packets. Proc. of the ACS/IEEE Internation Conference on Computer Systems and Applications, Jun 2001.

[4] Clam Anti-Virus. Clam AntiVirus, http://www.clamav.net/.

[5] F-Prot Anti-Virus. http://www.f-prot.com/.

[6] Open Anti-Virus. http://www.openantivirus.org/.

[7] Enterprise Management Associates. http://www.enterprisemanagement.com/.

[8] AV-Comparatives. http://www.av-comparatives.org/.

[9] AV-Test. http://www.av-test.org/.

[10] A. Baratloo, N. Singh, and T. Tsai. Transparent run time defense against stack smashing attacks. Proc. of USENIX Annual Technical Conference, Jun 2000.

[11] Bochs. http://bochs.sourceforge.net/.

[12] Virus Bulletin. http://www.virusbtn.com/.

[13] C. Luo C. Chen. Anti-virus agent for use with databases and mail. Patent, 5,832,208, Nov 1998.

[14] CERT. http://www.cert.org/.

[15] Microsoft Corporation. http://www.microsoft.com/.

[16] Microsoft Corporation. Kernel patch protection: Frequently asked questions. Technical report, 2006. Windows Hardware Developer Central.

[17] Symantec Corporation. http://www.symantec.com.

[18] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Z. Qian. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. pages 63–78. Proc. 7th USENIX Security Conference, Jan 1998.

[19] R.J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal for Resource Development*, 25(5):8, 1981.

[20] M. Moffie D. Uluski and D. Kaeli. Characterizing antivirus workload execution. In *Workshop on Architectural Support for Anti-virus and Security*, Oct 2004.

[21] Berkeley Software Distributions. http://www.freebsd.org.

[22] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. Proc. of the 10th USENIX Security Symposium, Aug 2001.

[23] G. Sorkin G. Tesauro, J. Kephart. Neural networks for computer virus recognition. IBM Thomas J. Watson Research Center, Aug 1996.

[24] IEEE. http://standards.ieee.org/regauth/posix/.

[25] VMWare INC. http://www.vmware.com/.

[26] VMWare Inc. Esx server performance and resource management for cpu-intensive workloads. Technical report, 2006.

[27] VMWare Inc. Timekeeping in vmware virtual machines. Technical report, 2006.

[28] VMWare Inc. Virtualization conisderations. Technical report, 2006.

[29] VMWare Inc. Virtualization overview. Technical report, 2006.

[30] M. Swimmer S. White J. Kephart, G. Sorkin. Blueprint for a computer immune system. IBM Thomas J. Watson Research Center, Oct 1997.

[31] F. Jahanian J. Oberheide, E. Cooke. Rethinking antivirus: Executable analysis in the network cloud. EE and CS Department, University of Michigan.

[32] J. Lettvin. Software anti-virus facility. Patent, 5,559,960, Sep 1996.

[33] M. Corio M. Wambach. Computer memory with anti-virus and anti-overwrite. Patent, 6,330,648, Dec 2001.

[34] McAfee. http://www.mcafee.com.

[35] Trend Micro. http://www.trendmicro.com/.

[36] Microsoft Corporation. *Windows Driver Development Kit Manual.*

[37] Sun Microsystems. http://java.sun.com/.

[38] Xen Virtual Machine Monitor. http://www.cl.cam.ac.uk/research/srg/netos/xen/.

[39] C. Nachenberg. State-based cache for antivirus software. Symantec Patent, 5,999,723, Dec 1999.

[40] C. Nachenberg. Antivirus accelerator. Symantec Patent, 6,021,510, Feb 2000.

[41] P. Barham B. Dragovic K. Fraser S. Hand T. Harris A. Ho R. Neugebauer I. Pratt and A. Warfield. Xen and the art of virtualization. http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf.

[42] C. Nachenberg R. Waldin. Antivirus accelerator for computer networks. Symantec Patent, 6,094,731, Jul 2000.

[43] G. Smith S. Basu, J. Dunagan. Why did my pc suddenly slow down. Microsoft Research.

[44] M. Silberstein. Designing a cam-based coprocessor for boosting performance of antivirus software. http://tx.technion.ac.il/m̃arks/.

[45] Symantec. In transit detection of computer virus with safeguard. Symantec Patent, 5,319,776, Jun 1994.

[46] Symantec. Understanding heuristics: Symantec's bloodhound technology. Technical report, 1997. Symantec White Paper Series, Volume XXXIV.

[47] Tarari. Regex content processor. http://www.tarari.com/regex/.

[48] Simics Virtutech. http://www.simics.com/.

[49] Wikipedia. I/o request packet. http://en.wikipedia.org/wiki/I/O_request_packets.

[50] Wikipedia. Neural network. http://en.wikipedia.org/wiki/Neural_network.

[51] Wikipedia. Virtualization. http://en.wikipedia.org/wiki/Virtualization.

[52] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Proc. of the 18th Annual Computer Security Applications Conference, Dec 2002.

[53] Y. Miretskiy A. Das C. Wright and E. Zadok. http://www.inf.bme.hu/ mszeredi/avfs/.

[54] D. Ye and D. Kaeli. A reliable return address stack: Microarchitectural features to defeat stack smashing. In *Workshop on Architectural Support for Anti-virus and Security*, Oct 2004.

Appendix A

IO-Stressing Benchmark

Shown below is the source code for the synthetic IO-stressing benchmark used for scalability testing.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main(int argc, char * argv[]) {
   char fileName[3], buf;
   int i;
   FILE *fp;
   time_t t1, t2;

   /* Begin timer */
   t1 = time(NULL);
```

```
for (i = 0; i < 1000; i++) {
   /* Set file name based off loop iteration */
   sprintf(fileName, "%d", i+1);

   /* Open the file */
   fp = fopen(fileName, "rb");
   if (fp == NULL) {
      printf("Error: fp = NULL\n");
      return -1;
   }
   printf("%s\n", fileName);

   /*
    * Traverse the file until we reach the end.  Force the file
    * to read byte-wise to trigger OS read operation, which in turn will
    * trigger anti-virus scanner
    */
   while (!feof(fp)) {
      buf = fgetc (fp);
      if (buf == 'a')
         /* So the compiler doesn't optimize this code */
         (1 == 1);
   }

   /* Close the file */
```

```
        fclose(fp);
    }

    /* Stop our timer and report how long the operation took */
    t2 = time(NULL);
    printf("Time: %d seconds\n", t2 - t1);
    return 0;
}
```