Characterizing Antivirus Workload Execution

Derek Uluski, Micha Moffie and David Kaeli
Computer Architecture Research Laboratory
Northeastern University
Boston, MA
{duluski,mmoffie,kaeli}@ece.neu.edu

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 paper we present 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 [3]. [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.

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 [11].

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 paper we will focus on execution overhead associated with an *on-access* model.

The rest of this paper is organized as follows. First, we present data showing the per-

[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.

formance penalty due to anti-virus execution in section 3. In section 4 we discuss our Simics environment and in section 5 we present some results from our workload characterization. We conclude the paper in section 6.

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 [11]. 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 [14]. Mechanisms that execute untrusted software in a sandbox, while monitoring behavior, are described in [11].

An important class of software-based intrusions include stack smashing attacks [6, 15]. 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 [7] 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 [4] 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 [8] provides a hardware-based stack protection; the hardware is responsible for encrypting and decrypting return addresses. Another approach described in [15] enhances the return stack address to detect buffer overflow attacks.

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 [3]. They compared the impact of running a range of anti-virus scenarios. Another comparison of different anti-virus software can be found in [5].

There have been a few studies that have proposed solutions to overcome anti-virus execution overhead. In [9], the authors analyze the underlying algorithms of open source anti-virus projects [1, 2] and propose a *CAM-based* co-processor for boosting anti-virus software execution performance. In [10], 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 [12] describes the implementation of a co-processor that is capable of simultaneously matching complex regular expressions.

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

3 Anti-virus performance degradation

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 4. Figure 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

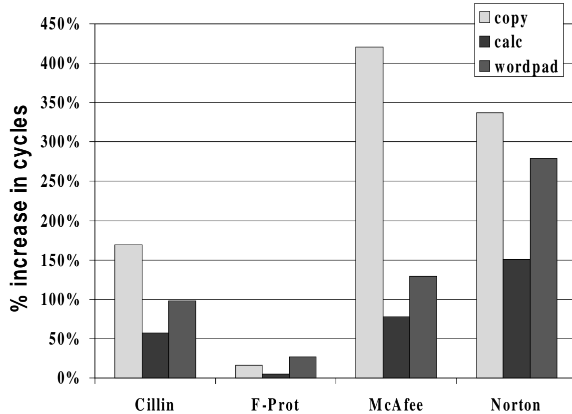

Figure 1: Anti-virus performance degradation.

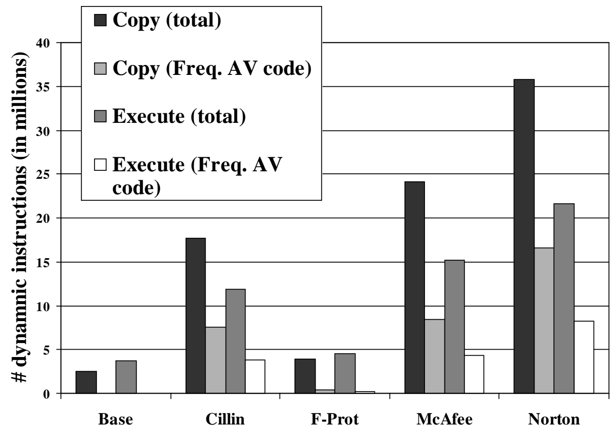

Figure 2: Anti-virus overhead.

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

4 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 [13], 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

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 1: Structure of the P4 microarchitecture used in this work.

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.

Figure 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 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).

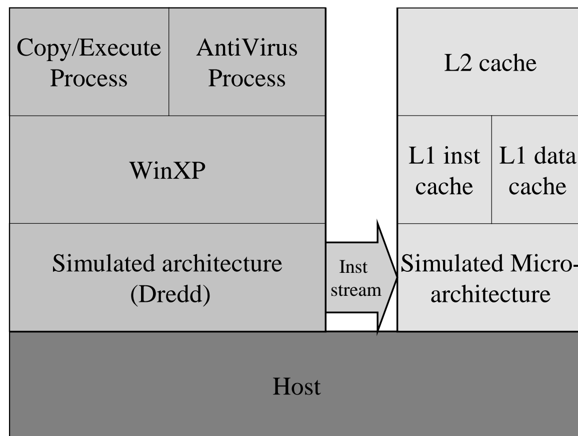

Figure 3: Multi-level architectural & micro-architectural simulation environment.

Table 2 summarizes the 5 different configurations:

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

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

Configuration	Anti-Virus edition	version
Base	-	-
NAV	Norton Anti-Virus Professional 2004	10.0.0.109
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 2: Five environments evaluated: Base has no anti-virus software running.

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.

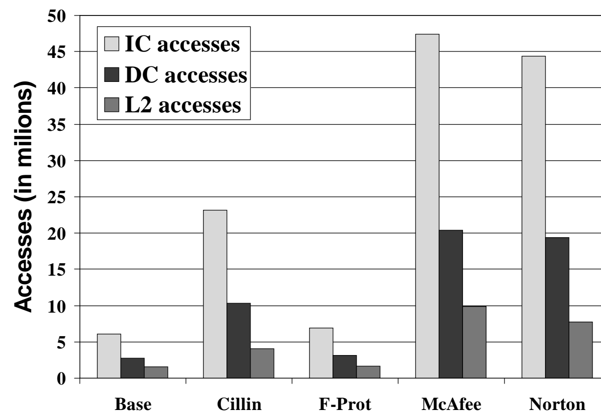

Figure 4: Cumulative memory accesses during execution of the copy test.

5 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.

5.1 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 4, 5, and 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 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 7, 8, and 9. We break down read 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

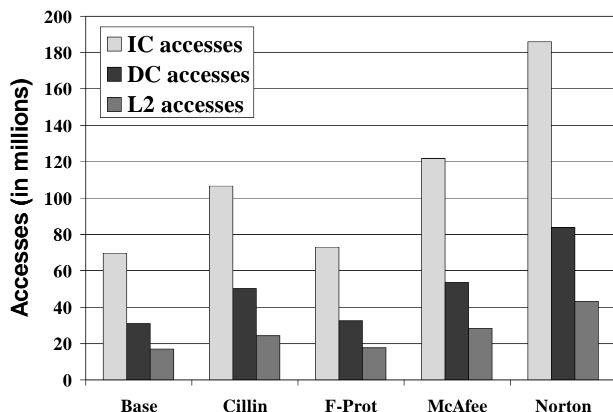

Figure 5: Cumulative memory accesses during execution of calc.

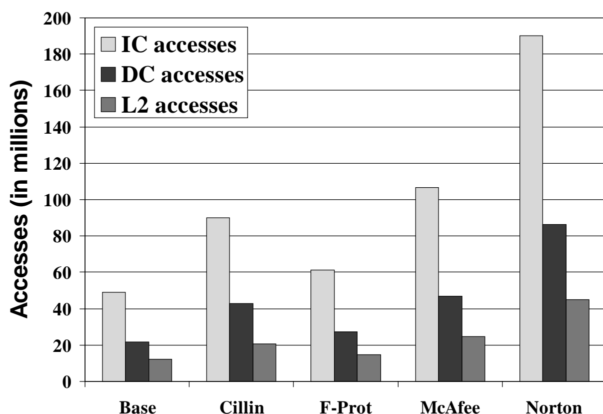

Figure 6: Cumulative memory accesses during execution of wordpad.

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.

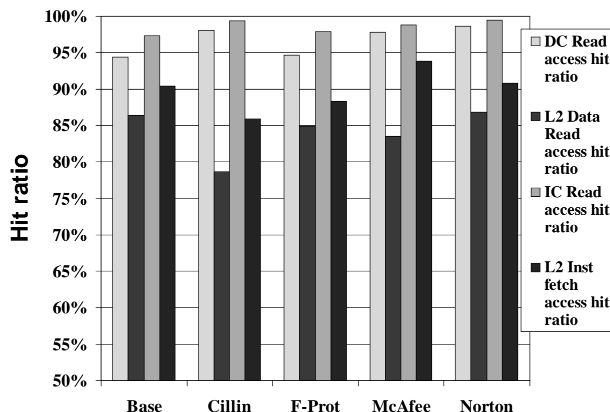

Figure 7: Cache hit ratio for the copy test.

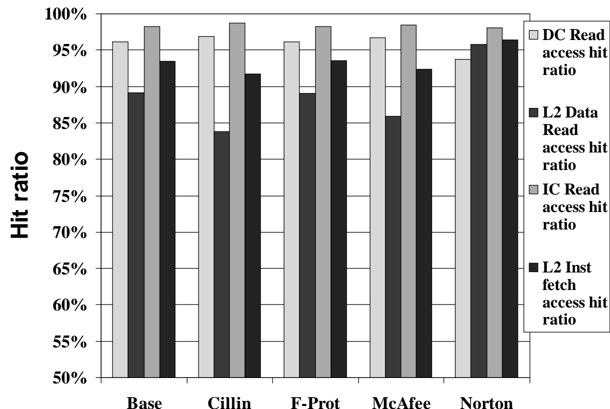

Figure 8: Cache hit ratio for calc.

5.2 Instruction memory footprint

In figure 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

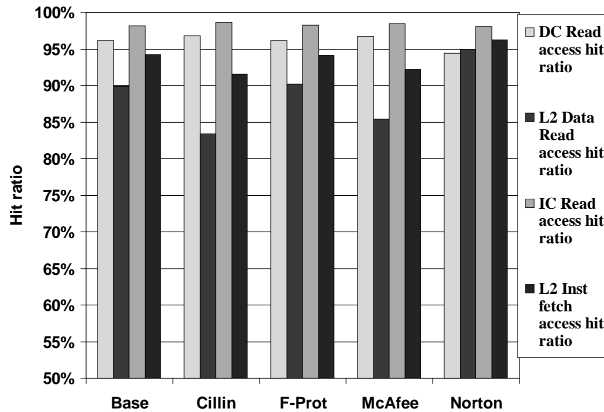

Figure 9: Cache hit ratio for wordpad.

case. The additional addresses that need to be fetched impact cache performance.

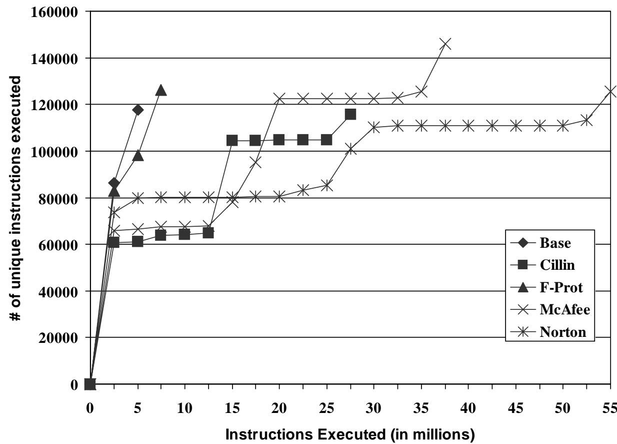

Figure 10: Anti-virus instruction memory footprint while executing Helloworld.

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 footprint of the copy process.)

5.3 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 11.

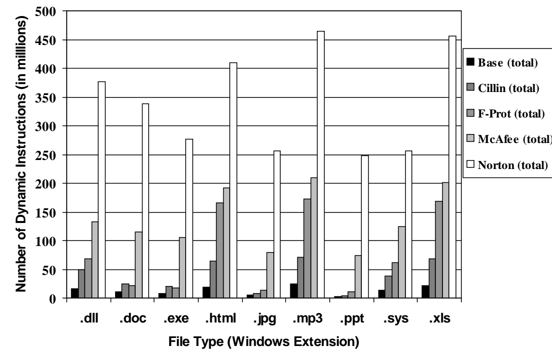

Figure 11: Overhead for different file types.

5.4 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 to reduce this overhead.

6 Conclusions

Viruses continue to plague computer users. Currently anti-virus software execution can impose significant overhead. In this work we presented a first look at the characteristics of the overhead introduced by four popular anti-virus packages. We characterized performance and memory behavior while running different binaries. We presented data showing the impact on the memory hierarchy when running anti-virus programs.

We plan to continue our research as we try to better understand anti-virus execution behavior. Our long-term goal is to develop novel hardware support that will alleviate much of the overhead introduced by the AV programs.

This work was supported by National Science Foundation Award Number 0310891 under the Computer Systems Architecture Program, and by the Institute of Complex Scientific Software at Northeastern University.

References

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

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

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

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

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

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

[7] 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.

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

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

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

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

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

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

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

[15] 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.