

Detection and Subversion of Virtual Machines

Dan Upton
University of Virginia
CS 851 - Virtual Machines

Abstract

Recent virtual machines have been designed to take advantage of run-time information to provide various services including dynamic optimization, instrumentation, and enforcement of security policies. While these systems must run in the same user space as the program running under their control, they must remain as transparent as possible so as to prevent affecting the correctness of the guest program. However, the virtual machine must store its own code and program state as well as information about the guest program. This data, stored in the program's user space, may lead to gaps in transparency that can be used to detect their presence. Additionally, while many virtual machines have a smaller code base than operating systems, they may still contain their own unique errors and security holes.

This research shows that it is possible to use different run-time clues to detect the existence of several common virtual machines. Further, information about the existence of these virtual machines can be used to attack the system. As a result, this paper presents countermeasures that should be taken by designers of these systems to prevent detection and attacks.

1 Introduction

The modern software development environment, using many source files and modules which may be dynamically loaded and may be compiled at different times and locations, has limited the effectiveness of traditional static analysis techniques. In addition, it has been suggested that due to the difficulty of debugging optimized code, release-version binaries are compiled with little or no optimization. To this end, dynamic optimization systems such as Dynamo [2] have been developed that allow for the run-time optimization across program boundaries. Microsoft's Mojo [6] performed similar run-time optimization of x86 binaries running under Windows NT. Both of these sys-

tems were designed specifically for optimization, and thus were designed to allow the optimization system to "bail out" and let the application run natively if the virtual machine overhead was too high.

Modern large software systems have the additional issue of being prone to software errors, as evidenced by the frequent security updates required for many Windows applications. Along with platforms designed for code portability, monitoring run-time behavior of software has become a concern. Systems such as Strata [25], DynamoRIO [4] [5], DELI [8], HD-Trans [26], and Pin [19] run the entire guest program under the control of the virtual machine. Many of these systems provide an application programming interface (API) that can be used to interface with the system to inspect and modify the instruction stream immediately before execution. Since it is guaranteed that all user code will be executed under control of the virtual machine, this opens the system to other applications including instrumentation [19] and security [25] [17].

For all of these virtual machines, and particularly those that translate binaries from one instruction set architecture (ISA) to another such as FX!32 [15] and IA-32 Execution Layer [3], transparency is a concern. More specifically, to a guest application, it should appear as if the application is running natively on its intended ISA. This involves being able to reconstruct proper exception state with respect to unmodified execution of the binary, as well as providing a correct view of the program's memory space. Due to memory protection requirements, the virtual machine itself must generally be running in the same memory space as the guest program and thus leads to additional data in the guest program's memory space. For instance, HD-Trans [26] stores register state on the guest program's stack, and FX!32 [15] stores return address information on a "shadow stack" maintained at the top of the native stack.

This information provides gaps in the transparency of the system which may reduce the usefulness of virtual machines for security purposes. Consider the case

where a user always runs untrusted code under control of a security-monitoring virtual machine for some number of runs until he feels reasonably sure that the program is safe. A program able to detect that it is not running natively can modify its behavior and only behave maliciously when it determines that it is running natively. Recent work concerning virtual machine honeypots [1] cites a conversation with a security researcher that suggests existence of automated tools for detecting virtual environments and modifying behavior, so this is a legitimate concern.

The work presented in this paper shows that it is possible to exploit information to detect the presence of a virtual machine, and in many cases specifically which one is being used, for several current virtual machines. Further, once a virtual machine is detected, that information can often be used to either subvert or circumvent the system. The primary contribution of this work is as a survey of these issues which then serve as suggestions for development of these virtual machines to lessen the likelihood of a guest program being able to detect or attack them.

The rest of the paper proceeds as follows. Section 2 provides background. Section 3 discusses methods for detecting the presence of a virtual machine. Section 4 follows with a discussion of ways to attack or circumvent a virtual machine, and Section 5 concludes the paper.

2 Background

Virtual machines come in two basic flavors, process virtual machines and system virtual machines. This work focuses primarily on detection and subversion of process virtual machines; however, a discussion of system virtual machines relative to the subject of security is worthwhile as a point of reference. In addition, Asrigo *et al* [1] reference a conversation suggesting detection techniques are available for use by malicious programs running within a system virtual machine.

Figure 1 shows the general layout of processes in the case of a process virtual machine. Here, the user may opt for each application whether to run it natively on the operating system/hardware or under the control of a virtual machine. Since running under control of a virtual machine generally introduces overhead, the option to allow a program to run natively may be appealing to a user for trusted or high-performance applications. If a virtual machine is being used to enforce security policy, each application having its own dedicated virtual machine has the added benefit of being able to customize security policy by application. Examples of process virtual machines include systems

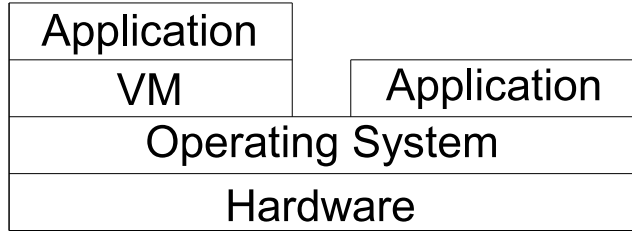


Figure 1. Layout of hardware and software for process virtual machines. On the left is an untrusted or non-native binary running under control of a virtual machine; on the right, an application runs natively.

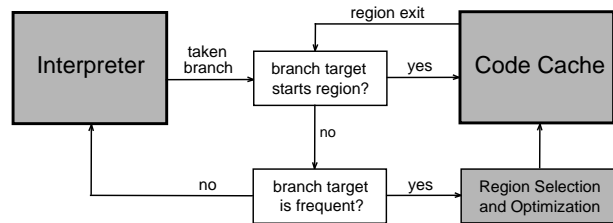


Figure 2. Overview of a generic dynamic optimization system. Shaded boxes correspond to parts of the system that execute the guest program.

like DynamoRIO [4][5], Strata [25], HDTrans [26], Valgrind [22], and Pin [19]. Bytecode compiled platforms such as the Java Virtual Machine [18] and the Microsoft Common Language Infrastructure [20] are special instances of process virtual machines in that the option to trust code and run it natively does not exist. Figure 2 shows an overview of the layout of many process virtual machines; security policy enforcement overhead is reduced by the presence of the code cache, because the code must only be checked for safety once and then can be run as trusted from the code cache.

In the case of a system level virtual machine, the operating system and all applications tend to run under the control of the virtual machine. System level virtual machines may either run directly on the hardware, as in systems like Xen [10] or Transmeta’s Code Morphing Software [7], or as a user-level application itself, as in User-Mode Linux [9] or systems like the VMware Virtual Desktop Infrastructure [27] or Microsoft’s Virtual PC [21]. As with process virtual machines, it is possible in many cases to create separate security policies for different invocations; in addition, some system virtual machines allow virtualization of hardware re-

sources such as by creating a file on the host computer’s hard disk and only accessing that instead of accessing the full native disk. Figure 3 shows several potential layouts of hardware and applications with respect to system virtual machines.

Most previous work in terms of virtual machine security has been in either securing machines or allowing secure monitoring and logging of machine activities. Strata [25] provides the ability to intercept system calls which can be used to catch and prevent reading or writing sensitive files such as `/etc/passwd/`. Kiriansky *et al* present program shepherding [17] which can enforce policies about code origins and control transfers to prevent execution of arbitrary code. More recently, Hu *et al* use virtual machines to perform instruction set randomization to prevent code injection attacks [16]. Dunlap *et al* designed ReVirt [11], a system like Figure 3 (i) that performs logging in the virtual machine; the log files are thus protected from modification after a system becomes compromised, allowing an administrator to replay events both leading up to and following the attack. Garfinkel *et al* extend this model for Terra [12], which allows applications or groups of applications to have their own dedicated virtual machines as in Figure 3 (iii).

Some examples of work relating to the security of or weaknesses in virtual machines do exist. Govindavajhala and Appel [13] present work using soft memory errors to break Java’s strong type system and as a result circumvent the security manager and execute arbitrary code; however, the attack presented in their work was very space- and computation-intensive, and required physical access to the device to be able to be performed with any reasonable degree of success. Robin and Irvine [24] presented a survey of the issues with implementing a secure virtual machine monitor on Intel hardware, noting that there are a number of sensitive instructions (i.e., instructions that could modify virtual or host machine state, change sensitive registers, or access the memory system) that can be executed in an unprivileged mode. Finally, Holz and Raynal [14] present methods for detecting system virtual machines.

3 Detecting Virtual Machines

3.1 Environment Variables

Any process virtual machine must have a way to initially take control of the guest application. One of the common ways, used in HDTrans [26], Valgrind [22], and the Linux version of DynamoRIO [5], is to use Linux’s `LD_PRELOAD` environment variable. This de-

```

; dynamorio envtest
HDTrans present on system
Executing under DynamoRIO

; HDTrans envtest
Executing under HDTrans
DynamoRIO present on system

; valgrind --tool=none envtest
==6637==
HDTrans present on system
DynamoRIO present on system
LD_PRELOAD: /usr/lib/valgrind/vg_inject.so
==6637==

```

Figure 4. Shell session illustrating a program using environment variable-based detection.

fines additional libraries to be dynamically loaded prior to execution of the guest application. DynamoRIO and HDTrans set this variable by executing a shell script prior to calling `exec` on the guest program. Additionally, they each require an extra environment variable, `DYNAMORIO_HOME` and `HDTRANS_SODIR` respectively, to know where to find the preload libraries.

If an attacker is familiar with the different process virtual machines, it is straightforward to inspect these variables using the `getenv()` call and then use string comparisons to match against known libraries. Further, these variables offer at least two opportunities for an attacker to fingerprint virtual environments. First, while `LD_PRELOAD` is only available when a program is running under control of the virtual machine, the directory environment variables will be viewable by any process. A curious attacker could have a program phone home in one way or another to alert that a particular host has these programs installed and that subsequent executions may take place under their control. Second, if `LD_PRELOAD` is set to an unrecognized value, the program could again phone home with the path and filename, giving the attacker information to use to research the unknown virtual machine.

Figure 4 shows the output of a program configured to recognize DynamoRIO and HDTrans as executed on a machine with both directory environment variables set. It successfully recognizes execution under those two virtual machines; in addition, it is able to detect the presence of an unknown virtual machine when executed under Valgrind.

3.2 Stack Inspection

Other systems, such as Pin [19] or the Windows version of DynamoRIO [4], use methods other than

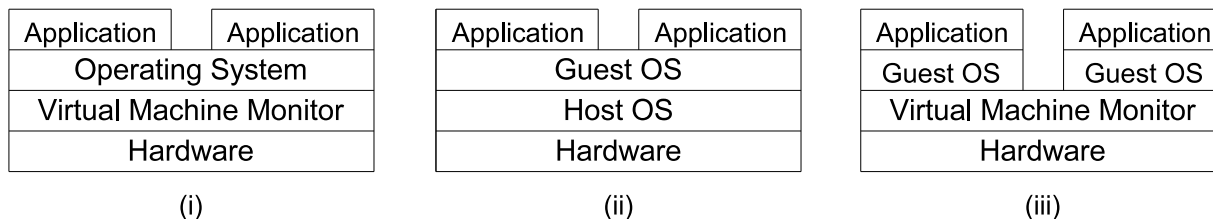


Figure 3. Overview of system virtual machines. (i) System with a virtual machine monitor between the OS and hardware, such as Xen. (ii) System running a single guest operating system on a single host operating system, such as User Mode Linux. (iii) Multiple guest operating systems on a virtual machine monitor.

LD_PRELOAD to take control of the guest program. Thus, detecting these virtual machines requires use of other techniques. In addition, non-stock installations of the aforementioned systems (such as renaming the library files or modifying the shell scripts) can break simple environment variable-based detection, so additional methods are useful as a backup.

Since all of these programs have their own program state, including the code cache, as well as helpful information about the guest program such as branch targets, it is necessary to store information on the program stack. Due to the memory protection system, the virtual machine must run in the same space as the guest program; virtual machine state is usually loaded much higher on the stack in an attempt to prevent overlapping the space the guest program will use. For instance, the first variable in the `envtest` program used above is stored at address `0xBFFFF874`, while HDTrans’s [26] code cache is initialized at `0xB789ECFA`.

A direct way then to attempt to detect a virtual machine is to directly reference its code cache or some other part of its state. Since this will lead to a segmentation fault if no additional state resides at that location, it is necessary to register a SIGSEGV handler to recover and continue execution. However, there are a few issues with this method:

- It is necessary to have access to at least the virtual machine executable, and possibly the source code, to be able to reliably determine the code cache starting address.
- Non-stock builds of any given virtual machine, such as adding additional profiling state to HDTrans, will result in the code cache residing further up the stack. This is only a minor problem, as it is likely that the pointer to where the code cache was expected will still land on some program state.
- Address-space randomization [23] moves things around and essentially foils this detection method.

In particular, the difference between the first variable and the address of the code cache varies, so it is unreliable to even try to detect using a relative offset.

A related method is to explore the stack by starting at the first variable in an activation record and then move up the stack. This again requires a handler for SIGSEGV because segmentation faults will be generated on any attempt to reference memory between the top of the guest program’s stack and the beginning of the virtual machine’s state. By waiting for the first SIGSEGV, indicating that the program’s stack has been passed, and then checking to see if a memory access further up the stack ever succeeds, the program can with some degree of accuracy whether something else is running in its memory space. This method has some issues as well:

- Some of the virtual machines do not appear to support the SIGSEGV handler using `sigsetjmp()` and `sigsetlongjmp()`. Valgrind exits immediately when the guest program executes `sigsetjmp()`; DynamoRIO fails with a segmentation fault in its own code when the guest program would have had one.
- There are some instances of executing the stack inspection program natively that give off false positives. It appears that this is related to address-space randomization as well, because running the same program on a Linux machine with randomization disabled did not lead to any false positives.
- While it may be possible to attempt to walk the stack all the way from the first activation record to the bottom of memory (i.e., address `0x00000000`), this can be a time-consuming process. Figure 5 shows a comparison of walking the stack from the first activation record to the bottom of memory natively to the performance under HDTrans and

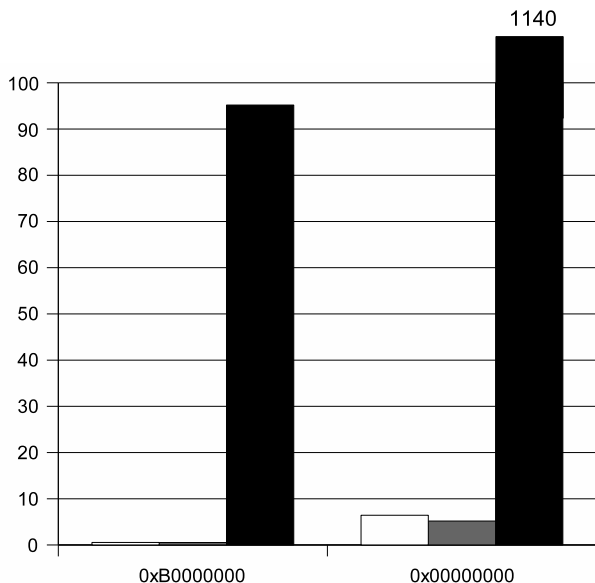


Figure 5. Comparison of times in seconds to explore a small part of the process stack and the full stack. Times reported are the best of 3 runs, with the exception of Pin walking to the bottom of memory which is an approximation based on one run.

Pin. Depending on what else the malicious application does, long periods of time spent exploring the stack may be noticed, and so it may be in the best interest of the application to search no further than necessary. This in turn leads to possibly ending the search too quickly.

A final stack inspection-related method could be to simply check the address of local variables. While these virtual machines are designed with transparency in mind, some appear to modify the layout of memory. For instance, recall that natively, `envtest` has its first variable at address `0xBFFFF854`. Under Pin, this is moved to `0xBFFFF824`, and under Valgrind, it is moved to `0xBEFFF804`. Valgrind skins may further alter the memory stack; Memcheck moves the same variable to `0xBEFFF7D4`. While this method is again prevented by address-space randomization, the fact that memory locations are modified is still of interest.

4 Attacking Virtual Machines

4.1 Library file-based methods

Using environment variables to specify library paths and libraries to preload presents a gap in the transparency of the system, as shown above. However, it has a second and perhaps worse consequence: it may directly open the virtual machine to attack. In particular, overwriting the libraries will prevent later attempted executions under control of the virtual machine from succeeding unless the libraries are rebuilt. On a basic level, this is would be preventable by setting those files from being unwritable. However, the common case is that the program running under the virtual machine’s control is being run by the person who installed the virtual machine itself; thus, the malicious program simply has to add a `system()` call to first `chmod` the file before writing it.

Recall that from Figure 4, the program was able to read the full path to Valgrind’s library file, even though it did not necessarily recognize what virtual machine the file belonged to. This means that even unknown virtual machines are vulnerable to attack when using `LD_PRELOAD` to take control of the guest program.

While simply overwriting the library file can potentially cause enough trouble, a worse attack is to either modify or replace the binary with one which is preloadable. The latter case is straightforward to accomplish in a rather blunt manner: simply include the replaced library as an otherwise apparently innocuous file in the malicious application’s distribution, and then use `system()` to execute a `cp` command to copy the malicious library file over the second file. Note that in this direct case, writing the file while it is still in memory causes a bus error, as does trying to copy over it. Thus, successfully replacing the binary would require either two executions of the malicious program: the first will effectively destroy the original but then terminate with a bus error, and the second will make the copy successfully. Alternately, executing `mv` will successfully overwrite the library file in a single step. At this point, anytime the user executes a program under control of the virtual machine, the code in the malicious user’s library’s `_init()` and `_fini()` functions will be executed. Figure 6 shows an example execution of a program successfully replacing HDTrans’s [26] library.

4.2 Code cache-based methods

Many traditional exploits are based on buffer overflows that lead to the execution of new code, frequently involving returning to `libc` to attack using preexisting

```

; HDTrans breakHD
Executing under HDTrans
DynamoRIO not found

; HDTrans breakHD
Causing trouble
Executing under HDTrans
Causing trouble
...or not
DynamoRIO not found
...or not

```

Figure 6. Example of replacing a preload library file. In the second execution, the phrase “Causing trouble” is printed when the malicious library is dynamically loaded.

code, thus circumvent restrictions on executing code from data pages such as the program stack. In turn, several methods have been developed to prevent these attacks. However, the code cache used by many virtual machines provides a large region of memory where a malicious program may be able to modify control flow. For instance, while optimized traces are linked to keep execution in the code cache as long as possible, exit stubs are created to return control to the virtual machine when a new code region must be translated. Overwriting the address to jump back to the virtual machine’s control could be equivalent to overwriting a return address in a typical “stack smashing” exploit.

Unfortunately, I was not able to successfully modify anything in the code cache. HDTrans was the most likely target, since it is easy to determine its initial code cache location. My experimental intent was to write a small program and then use `gdb` to find an address near the end of `main`, then have a function perform an infinite loop of overwriting the code cache with an instruction to jump to it. The program itself was small and simple enough that I would expect the code cache to be small and so it should not take long to completely overwrite it; however, writing one direction along the cache resulted in remaining in the infinite loop, and the other direction resulted in an immediate segmentation fault. It has also been suggested that some virtual machines use `mprotect()` to prevent the guest program from being able to modify the cache [28],

so this may not even be possible.

5 Conclusions and Future Work

This work has shown that it is possible to detect the presence of virtual machines during program execution, both using environment variables and stack inspection. Additionally, it has shown that there exist flaws in the transparency of the system, such as memory locations of variables, which may lead to further detection methods on some hosts. Finally, it has presented an example of how to attack both the virtual machine and the host system itself, by removing or replacing certain files.

Kiriansky’s program shepherding paper [17] presents runtime defenses of the code cache and data structures by keeping track of whether the system is currently executing in application mode or RIO mode and only allowing modification of data structures while in RIO mode. In order to enforce this, it must monitor system calls to prevent the application from changing memory protections. This does not directly address modifying or overwriting library files; however, since system calls must be monitored, it would be straightforward enough to watch for system calls that would do so. Since some systems are started using a shell script, an additional degree of protection may be afforded by first having the script verify a checksum on all involved libraries; however, this would still require protecting shell script from modification to have a new checksum matching the modified library.

For transparency from detection, systems should monitor whether the application is trying to read outside of its own stack and perhaps raise a `SIGSEGV`. It may appear that this would affect the correctness of the guest program; however, if the program ran natively, the operating system would raise this same error. Additionally, intercepting requests for environment variables associated with the virtual machine would provide an extra measure of detection prevention.

Future work could involve looking for more substantial information from stack inspection to be able to distinguish between systems by the makeup of the code cache and system state section. In addition, some further attempts to modify the code cache to either execute arbitrary code or to jump back to execute natively would be interesting, although based on the idea that many systems may protect their code cache, it may require a modified system and be purely academic.

References

- [1] K. Asrigo, L. Litty, and D. Lie. Using VMM-based sensors to monitor honeypots. In *2nd International Con-*

- ference on Virtual Execution Environments (VEE06), June 2006.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
 - [3] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems, 2003.
 - [4] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.
 - [5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization, 2003.
 - [6] W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.
 - [7] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges, 2003.
 - [8] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher. Deli: A new run-time control point, 2002.
 - [9] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.
 - [10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
 - [11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.
 - [12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing, 2003.
 - [13] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003. IEEE Computer Society.
 - [14] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *6th IEEE Information Assurance Workshop*, June 2005.
 - [15] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, pages 3–12, February 1997.
 - [16] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *2nd International Conference on Virtual Execution Environments (VEE06)*, June 2006.
 - [17] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding, 2002.
 - [18] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, Second Edition. <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>.
 - [19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.
 - [20] E. Meijer and J. Gough. Technical overview of the Common Language Runtime, 2000.
 - [21] Microsoft. Microsoft Virtual PC 2004. <http://www.microsoft.com/virtualpc>.
 - [22] N. Nethercote and J. Seward. Valgrind: a program supervision framework, 2003.
 - [23] PaX Team. PaX address space layout randomization (ASLR). <http://pax.grsecurity.net/docs/aslr.txt>.
 - [24] J. S. Robin and C. E. Irvine. Analysis of the intel pentium’s ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, August 2000.
 - [25] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and re-targetable software dynamic translation. In *1st International Symposium on Code Generation and Optimization*, pages 36–47, March 2003.
 - [26] S. Sridhar, J. S. Shapiro, and P. P. Bungale. HDTrans: A low-overhead dynamic translator. In *2nd International Conference on Virtual Execution Environments (VEE06)*, June 2006.
 - [27] VMWare Inc. VMware Virtual Desktop Infrastructure. <http://www.vmware.com>.
 - [28] D. Williams. Private conversation, April 2006.