Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection

Brendan Dolan-Gavitt*, Tim Leek†, Michael Zhivich†, Jonathon Giffin*, and Wenke Lee*

*Georgia Institute of Technology
School of Computer Science
{brendan,giffin,wenke}@cc.gatech.edu
†MIT Lincoln Laboratory
{tleek,mzhivich}@ll.mit.edu

***Abstract*—Introspection has featured prominently in many recent security solutions, such as virtual machine-based intrusion detection, forensic memory analysis, and low-artifact malware analysis. Widespread adoption of these approaches, however, has been hampered by the *semantic gap*: in order to extract meaningful information about the current state of a virtual machine, detailed knowledge of the guest operating system's inner workings is required. In this paper, we present a novel approach for automatically creating introspection tools for security applications with minimal human effort. By analyzing dynamic traces of small, in-guest programs that compute the desired introspection information, we can produce new programs that retrieve the same information from outside the guest virtual machine. We demonstrate the efficacy of our techniques by automatically generating 17 programs that retrieve security information across 3 different operating systems, and show that their functionality is unaffected by the compromise of the guest system. Our technique allows introspection tools to be effortlessly generated for multiple platforms, and enables the development of rich introspection-based security applications.** [1]

I. Introduction

As kernel-level malware has become both more common and more sophisticated, some researchers have advocated protecting insecure commodity operating systems through the use of virtualization-based security [11], [14], [19], [27]. By moving protection below the level of the OS, these solutions aim to provide many traditional security services such as intrusion detection and policy enforcement without fear of subversion by malicious code running on the system. At the same time, because the code base of the hypervisor is small, there is some hope that it can be verified and provide secure isolation for the virtual machines it supervises.

This enhanced security and isolation comes at a cost, however. Because the guest operating system is untrusted, information about its state can not be reliably obtained by calling standard APIs within the guest. Instead, security tools must use *introspection* to retrieve information about the current state of the guest OS by examining the physical memory of the running virtual machine and using detailed knowledge of the operating system's algorithms and data structures to rebuild higher level information such as lists of running processes, open files, and active network connections. Even if the guest OS is compromised, the tools using introspection will continue to report accurate results, so long as the underlying internal data structures used by the operating system are intact.

The creation and maintenance of introspection tools, therefore, is dependent on detailed, up-to-date information about the internal workings of commodity operating systems. Even for systems where the source code to the kernel is available, acquiring this knowledge can be a daunting task; when source is not available, prolonged effort by a skilled reverse engineer may be required. Moreover, the time and effort spent divining the internals of a particular version of an operating system may not be applicable to future versions. This problem of extracting high-level semantic information from low-level data sources, known as the *semantic gap*, presents a significant barrier to the development and widespread deployment of virtualization security products.

The fact that creating introspection tools by hand is difficult has multiple security implications. Security professionals and system administrators rely upon the output of hand-built introspection tools; at the same time, these tools are fragile and often cease to function upon application of operating system patches. Currently, therefore, one must often choose between keeping an up-to-date system and maintaining a working set of introspections. Moreover, hand-built introspection tools may introduce opportunities for attackers to evade security tools: to the extent that introspection must replicate the behavior of in-guest functionality, the developer must have an accurate model of the OS's inner workings. Garfinkel [10] has demonstrated, with numerous examples drawn from his experience with system call monitors, that any divergence between reality and the model assumed by a security program can provide opportunities for evasion.

In this paper, we both ease the burden of maintaining introspection-based security tools and make them more secure by providing techniques to *automatically* create programs that can extract security-relevant information from outside the guest virtual machine. Our fundamental insight is that it is typically trivial to write programs inside the guest OS that compute the desired information by querying the built-in APIs. For example, to get the ID of the currently running process in Linux, one need only call `getpid()`; by

[1]This work was performed under U.S. Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are not necessarily endorsed by the U.S. Government.

contrast, implementing the same functionality from outside the virtual machine depends on intimate knowledge of the layout and location of kernel structures such as the `task_struct`, as well as information about the layout of kernel memory and global variables. Our solution, named Virtuoso, takes advantage of the OS's own knowledge (as encoded by the instructions it executes in response to a given query) to learn how to perform OS introspections. For example, using Virtuoso, a programmer can write a program that calls `getpid()` and let Virtuoso trace the execution of this program, automatically identify the instructions necessary for finding the currently running process, and finally generate the corresponding introspection code.

Virtuoso takes into account complexities that arise from translating a program into a form that can run outside of its native environment. It must be able to extract the program code, and its dependencies; to do this, we use a dynamic analysis that captures *all* the code executed while the program is running. At the same time, Virtuoso must be able to distinguish between the introspection code and unrelated system activity, a challenge that we overcome by making use of dynamic slicing [17] to identify the exact set of instructions required to compute the introspection. Finally, the use of dynamic analysis means that a single execution of the program may not cover all paths required for reliable re-execution. To ensure that our generated programs are reliable, we introduce a method for merging multiple dynamic traces into a single coherent program and show that its use allows our generated programs to achieve high reliability.

This paper makes the following contributions:

1) A new method for automatically generating introspection programs for security and other purposes. This method requires only a programmer's knowledge of operating system APIs and a few minutes of computation, rather than detailed knowledge of kernel data structure layouts or weeks of reverse engineering.
2) A new technique for extracting *whole-system* executable dynamic slices of binary "programs". Previous work has focused on extracting simple functions from user-level code only.
3) An algorithm for combining the results of multiple dynamic traces into a single program, which overcomes some of the limitations of dynamic analysis present in previous work [16], and a proof of our algorithm's correctness.
4) Creation of memory analysis tools for the Haiku operating system [12], generated without knowledge of the Haiku kernel's internal design or data structures.

The remainder of this paper is organized as follows. Section II discusses related work in binary code extraction and virtual machine introspection. Next, Section III describes the scenarios in which Virtuoso can be used to speed up the development of security-focused introspection tools. Sections IV and V discuss the design and implementation of our system. In Section VI, we evaluate Virtuoso's generality, security, reliability, and performance by generating 17 introspection programs for Windows, Linux, and Haiku. Finally, we conclude by outlining future research directions (Section VII) and summarizing our conclusions (Section VIII).

II. Related Work

Most closely related to our own techniques are two recent works which perform extraction of binary code for security purposes. Binary Code Reutilization (BCR) [5] and Inspector Gadget [16] focus on the problem of extracting subsets of binary code, particularly in order to reuse pieces of malware functionality (such as domain name generation or encryption functions). While the former primarily uses static analysis and recursively descends into function calls, the latter adopts a dynamic approach similar to our own. Inspector Gadget performs a dynamic slice on a log collected from a single run of the malicious code and changes half-covered conditional branches into unconditional jumps (the authors do not discuss the possible correctness issues that may arise from this strategy, nor do they say whether their system can combine multiple dynamic traces into a single program). Both systems are more limited in scope than Virtuoso and do not consider kernel code. They make extensive use of domain knowledge about the target operating system, and would therefore be difficult to adapt to new platforms. By contrast, Virtuoso uses no domain knowledge aside from understanding the x86 platform, and performs *whole-system* binary code extraction.

More generally, virtual machine introspection has played an integral role in a number of recent security designs. Garfinkel and Rosenblum [11] first proposed the idea of performing intrusion detection from outside the virtual machine. Their system introspects on Linux guests to detect certain kinds of attacks; higher level information is reconstructed through the use of the `crash` utility [25]. Since then, other researchers have proposed using virtualized environments for tasks such as malware analysis [7], [13] and secure application-layer firewalls [31]; these systems all make heavy use of virtual machine introspection and therefore must maintain detailed and accurate information on the internals of the operating systems on which they introspect.

Introspection can also be useful in contexts outside of traditional virtualization security. Petroni et al. [28] presented a system called Copilot that polls physical memory from a PCI card to detect intrusions; such detection needs introspection to be useful. Malware analysis platforms that run samples in a sandboxed environment, such as CWSandbox [34] and Anubis [3], can also benefit from introspection: by extracting high-level information about the state of the system as the malware runs, more meaningful descriptions of its behavior can be generated. At the same time, this introspection must be secure and unobtrusive, as the guest OS is guaranteed to be compromised. Our work can help provide higher-level semantic information to such systems.

Originally proposed by Korel and Laski [17], the dynamic slicing at the heart of our algorithm has been used in several other recent security tools, such as HookMap [33] and K-Tracer [18]. The former makes use of this technique to identify memory locations in the kernel that could be used by a rootkit to divert control flow, while the latter uses a combination of dynamic backward and forward slicing (also known as "chopping") to analyze the behavior of kernel malware with respect to sensitive data such as process listings. Additionally, Kolbitsch et al. [15] described a malware detection system that uses dynamic slicing to extract the code necessary to relate return values and arguments between system calls.

III. Introspection for Security

To motivate the design of Virtuoso, we consider the common scenario where *secure introspection* into the state of a running virtual machine (known as the *guest*) from outside that VM (i.e., from the *host*) is needed. This need may arise when performing out-of-the-box malware analysis or debugging [13], [14], or when attempting to secure a commodity operating system by placing security software in an isolated virtual machine for intrusion detection [11] or active monitoring [27]. In cases such as these it is easy to see why in-guest monitoring is undesirable: if the guest OS can be compromised, then the in-guest tools themselves can become corrupted or the system APIs can be altered to return false information (such as by omitting a malicious process from a task list).

Developing tools to perform such introspections, however, is no easy task: extracting even simple information such as the current process ID or a list of active network connections requires deep understanding of the operating system running in the VM. In particular, the *location* of the data, its *layout*, and what *algorithms* should be used to read it must all be known. While some recent work has addressed the problem of automatically reverse engineering data structure layouts [20], [21], [30], the overall task of creating introspections is still largely a manual effort. With Virtuoso, however, another option is available: by simply writing small programs (which we refer to as *training programs*) for the guest OS that extract the desired information, host-level introspection tools can be *automatically* created that later retrieve the same information at runtime from outside the running VM.

To illustrate how Virtuoso can be used to quickly create secure introspection programs, consider the running example shown in Figure 1. Here, a developer wishes to create an introspection that obtains the name of a process by its PID. To do so, he first writes an in-guest program that queries the OS's APIs for the information. Such programs are generally trivial to create for even modestly skilled programmers. The program is then run inside Virtuoso, which records multiple executions, analyzes the resulting instruction traces, and creates an out-of-guest introspection program that can retrieve the name of a process running inside a virtual machine. This program can then be deployed to a secure virtual machine and used to obtain this security-relevant information at runtime. The generated program does not call any in-guest APIs, but rather contains all the low-level instructions necessary to implement that API outside of the virtual machine. This entire procedure involves only minimal effort on the part of the developer, and requires no reverse engineering or specialized knowledge of the OS's internals.

Tools generated in this way are specific to the operating system for which the corresponding training programs were developed; that is, if the developer writes a program that gets the current process ID in Windows XP, the generated out-of-the-box tool will only work for Windows XP guests. However, supporting a new platform only requires adapting the training program to a new OS and its API, rather than costly reverse engineering or source code analysis by a specialist. Moreover, different releases of various operating systems are often backwards compatible at either the source or binary level. This is the case, for example, with Microsoft Windows: programs written for Windows NT 3.1 (released over 15 years ago) can run without modification on Windows 7. In this case Virtuoso could analyze the behavior of the same program under each release of Windows, generating a suite of host-level introspection tools to cover any version desired with no additional programmer effort. Changes in the underlying implementations of the OS APIs would be automatically reflected in the generated host introspection tools.

Once the introspection tools have been generated for a particular OS release, they can be deployed to aid in the secure monitoring of any number of virtual machines running that OS. The generated introspection tools will provide accurate data about the current state of the guest VM, even when the code of that VM is compromised.[2] Thus, Virtuoso produces tools that enable secure monitoring of insecure commodity operating systems, and accomplishes this task with only minimal human intervention.

A. Scope and Assumptions

Although we believe Virtuoso represents a large step forward in narrowing the semantic gap, there are some fundamental limitations to our techniques and constructs that Virtuoso is not currently equipped to handle. First, Virtuoso is, by design, only able to produce introspection code for data that is already accessible via some system API in the guest. These introspections are a subset of those that could be programmed manually, and it is possible that some security-relevant data are not exposed through system APIs. For those functions that are exposed through system APIs, Virtuoso greatly reduces the cost of producing introspections; we describe six such useful introspections in Section VI.

Second, there are certain code constructs that are difficult to reproduce, such as synchronization primitives (which may require other threads to act before an introspection can proceed) and interprocess communication (IPC). These

[2]As with all introspection, malicious alterations to the guest's kernel data structure layouts, locations, or algorithms may still cause incorrect results to be reported.

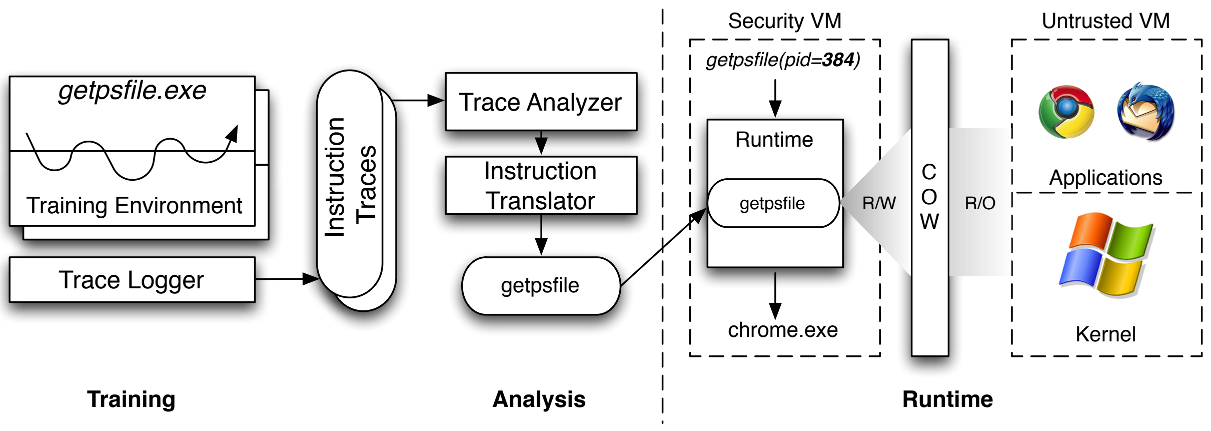


Figure 1. An example of Virtuoso's usage. A small in-guest *training program* that retrieves the name of a process given its PID is executed in the training environment a number of times, producing several instruction traces. These traces are then analyzed and merged to create an introspection program that can be used to securely introspect on the state of a guest virtual machine from outside the VM. We explain further details of this diagram in Section IV.

limitations are not neccessarily fundamental, but they require additional research to overcome. We discuss these limitations further in Section VII.

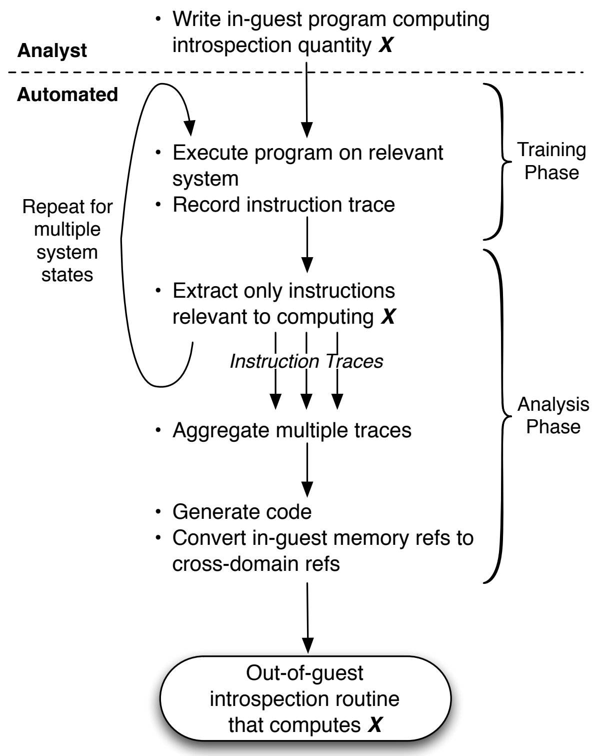
Finally, if an introspection requires interaction with a hardware device, such as the disk or a network interface, the introspection cannot currently be generated by Virtuoso. In Section VII we discuss possible extensions to Virtuoso that could enable automatic generation of some such introspections, however, for some devices any interaction might irreversibly perturb the guest state (for example, if an introspection must access the network, there is no way to reverse its effects—the packets cannot be "unsent"). This would leave the guest in an inconsistent state and may cause the system to become unstable; it also violates the principle that introspection should not alter the guest.

IV. Virtuoso Design

At a high level, Virtuoso creates introspection tools for an operating system by converting in-guest programs that query public APIs into out-of-guest programs that reproduce the behavior of the in-guest utilities. This is done in three phases (Figure 2): a *training phase*, in which an in-guest program that computes the desired information is run a number of times, and a record of each execution is saved; an *analysis phase*, which extracts just the instructions involved in computing the data required for the introspection, merges the traces into a single program, and translates that program into one that can be executed from outside the virtual machine; and finally a *runtime phase*, in which the generated program is used to introspect on a running virtual machine. These steps are described in more detail below.

A. Training

The training phase is shown in Figure 1, our running example: the training program, `getpsfile.exe`, is run several times in the training environment to capture a variety



Analyst
Write in-guest program computing introspection quantity X
Automated
Execute program on relevant system
Record instruction trace
Training Phase
Repeat for multiple system states
Extract only instructions relevant to computing X
Instruction Traces
Aggregate multiple traces
Analysis Phase
Generate code
Convert in-guest memory refs to cross-domain refs
Out-of-guest introspection routine that computes X

Figure 2. A high-level conceptual view of our system for generating introspection tools. This view corresponds to the training and analysis phases shown in Figure 1. The programmer creates an in-guest program that computes the desired introspection quantity by calling standard OS APIs. This in-guest program is then run repeatedly under various system states, and the instructions executed are logged. These instruction traces are then analyzed to isolate just the instructions that compute the introspection quantity, merged into a unified program, and then translated into an out-of-guest introspection tool.

of program paths, producing a number of instruction traces. For this stage, we assume the developer can create such

```
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include "vmnotify.h"

int main(int argc, char **argv) {
  char *name;
  HANDLE h;
  DWORD pid = atoi(argv[1]);
  name = (char *) malloc(1024);
  vm_mark_buf_in(&pid, 4);
  vm_mark_buf_in(&name, 4);
  h = OpenProcess(PROCESS_QUERY_INFORMATION,0,pid);
  GetProcessImageFileName(h,name,1024);
  vm_mark_buf_out(name, 1024);
  return 0;
}
```

Figure 3. A complete sample program that gets the name of a process given its PID under Windows. Programs such as these are easy to write from inside the operating system using documented APIs. `GetProcessImageFileName`, `OpenProcess`, and `malloc` are standard API calls known to Windows programmers. `vm_mark_buf_in` and `out` notify Virtuoso of input and output.

small, in-guest programs that compute the data needed for the introspection tool. This places a very minimal burden on the programmer: because ordinary programs running inside the target operating system need to query many of the same kinds of information desired by introspection tools, OS designers typically make rich APIs available for finding out this information. Additionally, such APIs are generally easy to use, stable, and well-documented, as they are the primary interface through which programs talk to the OS.

A sample program that retrieves the name of a process under Windows is shown in Figure 3. The program, annotated with markers that indicate where to begin and end tracing (`vm_mark_buf_in` and `out`), invokes the Windows API functions `OpenProcess` and `GetProcessImageFileName` to get the process name. The annotations also record the addresses of input and output buffers for the program; these will ground the data flow analysis performed to help identify what portions of the system's execution are necessary to compute the introspection quantity (in this case, the process name).

Even for simple programs like the one described above, there may be a number of different execution paths taken through the program depending on the state of the OS when the program is executed. To produce a program that works correctly in all situations, we run the program repeatedly in an instrumented environment and collect traces that record all instructions executed by the program and operating system. These execution traces are then analyzed and merged together to produce an out-of-guest introspection tool that can be used to reliably provide security-relevant information for monitoring applications.

B. Analysis

Although the execution traces gathered in the training phase contain all the instructions needed to compute the introspection quantity, they also contain a large amount of extraneous computation. This extra "noise" is present because our traces record the execution of the whole system: in addition to computing the introspection quantity, the traces also contain unrelated events such as interrupt handling and unrelated housekeeping tasks performed by the kernel. Because we are only interested in the code that is necessary for producing a working introspection program, we must excise these extraneous parts of the traces. We accomplish this by first throwing away or replacing parts of each trace that we know *a priori* to be unrelated to the introspection, such as hardware interrupts and memory management. Next, we identify the instructions that actually compute the output by performing a dynamic data slice [17] on each trace. Finally, we merge the slice results across basic blocks and traces, producing a unified program that can be translated into an out-of-guest introspection routine.

In order to produce a working routine, the inputs and outputs to the training program must be identified. To do so, we look for places in the trace where buffers marked as inputs are used. These instructions are marked as places where input data is used so that, at translation time, we can generate code that splices in the program inputs at these points. Similarly, as the location of the output buffer may change depending on the current environment (for example, if the output buffer is on the stack, its location will change depending on the value of `ESP` when the introspection tool is run), the analysis determines which instructions are immediately responsible for writing the data to the output buffer. These "output-producing instructions" are marked; the translator will handle such instructions specially, so that the output data can be found independent of the runtime CPU state.

Once the instruction trace has undergone slicing, merging, input splicing and output instruction marking, the merged traces are translated into an out-of-guest introspection program. The programs generated by Virtuoso consist of a set of basic blocks together with successor information. Within a basic block, each individual instruction is implemented by a small snippet of code in a high-level language. The use of a high-level language allows our generated programs to be usable in many different contexts: forensic memory analysis, virtual machine introspection, and low-artifact malware analysis.

In our example (Figure 1), we can see that the Trace Analyzer and Instruction Translator together comprise the analysis phase, and create the final introspection program `getpsfile` from the instruction traces generated during training.

C. Runtime Environment

The translated code, however, cannot be executed directly. Instead, it must be provided with an appropriate *runtime environment* in which to execute. The runtime environment (shown on the right in Figure 1), is installed on the host and runs the translated instructions in a context that gives them access to the resources (such as CPU registers and memory) of the guest virtual machine without perturbing its state.

In particular, low-level operations that affect registers or memory are wrapped so that they enforce *copy-on-write (COW)* behavior. Whenever a write to memory occurs, it is redirected to the COW memory space (shown in the right panel of Figure 1); when a read occurs, it first checks whether the data exists in COW memory. If so, it reads from the COW space. Otherwise, the read is serviced directly from guest memory. A similar scheme is used to handle register access. This allows introspections to access the state of the running system without interfering with its operation.

There is some subtlety to the question of how to obtain values from the memory of the guest. Certain data seen during training may be specific to the training program (for example, static data such as strings), while other data may be part of the system's state as a whole. In the former case, we would want to use the values seen in training (as they may not be available in the environment in which the introspection program is run), whereas in the latter we would want to obtain the values from the guest itself (since the point of introspection, after all, is to obtain the state of the system). To differentiate between these two types of data, we currently make the simplifying assumption that *kernel* data seen during training is global, and should be read from the guest, while *userland* data is specific to the training program, and should be saved during training and re-used at runtime. This reflects the nature of the introspections we wish to capture, which obtain information about the state of the system as a whole rather than any individual process.

Although this policy currently limits the type of introspections that can be generated to those that inspect systemwide state, one could define other policies that make introspection into specific processes possible. If Virtuoso were provided with some means of locating the appropriate process context at runtime, one could define a policy that reads all data directly from the guest, allowing introspection into the memory of a specific process.

When performing an introspection (even with hand-generated tools), the guest CPU is paused in order to prevent the contents of memory from changing during the introspection. It might seem that one could keep the CPU running, improving performance by allowing the introspection to be carried out in parallel with the guest's execution. This performance boost would come at the cost of reliability, however, as changes to the underlying data examined by the introspection tool would almost certainly cause it to crash or report incorrect results. In addition, we have noticed that most generated introspections must be run when the CPU is in user mode. This is because the traces are recorded based on a userland program, and so some assumptions may be made about the state of global registers in userland that do not hold in kernel mode. To ensure reliability, our runtime environment polls for a user-mode CPU state and pauses the guest before performing any introspection.[3]

[3]Rather than polling, it is also possible to set up a callback within the guest whenever there is a transition from kernel to user mode, using the techniques described by Dinaburg et al. [7].

Finally, the Instruction Translator has special handling for instructions marked by the Trace Analyzer as inputs or outputs. Instructions that require input are replaced with equivalent instructions that explicitly take input from the environment in which the introspection tool runs. Output-producing instructions are translated so that their output is placed in a special buffer at a known location; when the program terminates, the contents of this buffer containing the desired introspection quantity are returned so they may be used by security tools. In our running example (Figure 1), we can see that the generated introspection program is provided with the PID 384 as input, and produces the output "chrome.exe".

Note that the contents of the output may still need to be interpreted (for example, by decoding the binary data as a string or integer, or even more complex interpretations such as displaying a timestamp in human-readable form). We provide a basic facility by which the user can provide a function that interprets the output from the runtime environment; however, we do not attempt to include this functionality in the generated introspection tool itself. We feel that this approach is justified, as the output will be the same as that produced by APIs within the guest OS, which we have assumed are well-documented. Therefore, the format of their outputs will most likely also be documented by the OS vendor.

V. Implementation

Having given a high-level view of our system, we now turn to the details of its implementation. Virtuoso's trace logging portion is built on top of QEMU [4], a fast emulator and binary translator. Our Trace Analyzer is written in Python, and consists of 1,843 SLOC;[4] the majority of this code consists of the data flow transfer functions for each QEMU micro-instruction. Finally, the Instruction Translator and runtime environment are 431 and 1,095 lines of Python code, respectively. These components, and their logical relationship, are shown in Figure 1 on page 4. We will discuss in detail the implementation of each component in this section.

A. Trace Logging

Our trace logger, which is shown on the left in Figure 1, is a modified version of QEMU. QEMU works by dynamically translating each guest instruction to a series of operations in a micro-instruction language. The micro-instructions are implemented by small C functions, which are compiled to host-level code and chained together to emulate the guest code on the host.

To enable logging, we inserted small logging statements into each micro-instruction's implementation. These logging statements record the current operation, along with its operands and any dynamic information needed to enable the data flow analysis (see Section IV-B for more details).

[4]Source line counts were generated using David A. Wheeler's 'SLOC-Count'.

Original x86 instruction:

```
MOV EDI, DWORD PTR [EBP+0x10]
```

QEMU micro-operations:

```
MOVL A0,  EBP
ADDL A0,  0x10
LDL  T0,  A0      ; 0x1b1acf00
MOVL EDI, T0
```

Figure 4. An example of an x86 instruction and the corresponding logged micro-instructions. The implicit pointer dereference has been made explicit, and the referenced address has been recorded to enable data flow analysis.

Thus, when the guest code is translated by QEMU, the generated code will be interleaved with logging functions that record the micro-instructions executed. This interleaving is necessary because accurate logging depends on dynamic information that will only be available once the previous instruction has finished executing.

Our logging functions record the current micro-operation, its parameters, and concrete *physical* memory addresses for each load and store instruction. Using the physical address allows data flow to be reconstructed even when multiple virtual addresses reference the same physical memory (this often occurs when a buffer is shared between a user application and the kernel, for example). Finally, we record the page table dependencies for each memory operation. This is necessary because each memory load or store implicitly depends on the addressing structures used: if another instruction in the trace has modified the virtual to physical address map, that instruction must also be included in order for the resulting program to function correctly. By recording the address of the page directory and page table entries used in calculating the address of the load or store, our dynamic slicing algorithm (described in Section V-C) can automatically determine what page table-manipulating code must also be included.

In Figure 4, we give an example of an x86 instruction and the corresponding QEMU micro-operations that are recorded by the Trace Logger. The x86 instruction is decomposed into a series of simpler operations: first, a memory address is computed by taking the value in the `EBP` register and adding `0x10` to it. Next, the `LDL` micro-op retrieves the value from memory at that address and stores it in the temporary register `T0`. Along with the micro-op, the logger records the address the memory was read from to capture any data flow dependencies. Finally, the value read is transferred to the `EDI` register.

These micro-instructions provide a natural IR for our analyzer. Not only are QEMU micro-ops simple to log, they also satisfy the requirement of performing relatively simple operations with few side effects. This greatly simplifies the task of tracking data flow, as the data defined and used by each micro-instruction is generally easy to understand.

Inside the guest, the *training program* (described in Section IV-A) must be able to signal the Trace Logger and inform it of the beginning and end of the introspection

```
#define MAGIC_IN  0xdeadbeef

void vm_mark_buf_in(void *buf, int len) {
  void *b = buf;
  int n = len;

  // [ register save omitted ]
  __asm MOV EAX, 0
  __asm MOV EBX, MAGIC_IN
  __asm MOV ECX, b
  __asm MOV EDX, n
  __asm CPUID
  // [ register restore omitted ]
}
```

Figure 5. In-guest code that signals the Trace Logger to begin tracing. Code that simply saves and restores clobbered registers has been omitted.

operation, as well as the location of the buffer where the output has been placed. Our implementation accomplishes this by co-opting the x86 `CPUID` instruction. When a magic value is placed in the `EBX` register, our modified QEMU will interpret `CPUID` as a signal to start or stop logging, and will record the values of `ECX` and `EDX` as the buffer's start and size, respectively. This allows us to bound our trace so that it includes only the operations relevant to introspection, and also provides the Trace Analyzer with the locations of the input and output buffers. The in-guest code that implements the notification is given in Figure 5.

B. Preprocessing

As discussed in Section IV-B, the trace is preprocessed before slicing and translation in order to remove hardware interrupts and replace calls to memory-allocating functions with function summaries (this is part of the analysis phase, i.e., the middle portion of Figure 1). Filtering interrupts has two benefits: it reduces the amount of code we need to analyze, and it eliminates many accesses to hardware devices (which may not be available at run time). Replacing memory allocation functions (such as malloc and realloc), while not necessary in theory, is currently required by our implementation, as portions of the OS-provided memory allocation facilities are carried out in the page fault handler, and we do not support hardware exceptions in our runtime environment.

Interrupt filtering is performed in the obvious way, by matching up pairs of interrupts (which are logged by the Trace Logger) and `iret` instructions (which are used in the x86 architecture to return from an interrupt). Because interrupts may be nested arbitrarily deep, we use a counter to ensure that we have matched a given `iret` with the appropriate interrupt. Once the outermost interrupt and `iret` have been identified, they can be excised from the trace.

There are two exceptions to interrupt filtering, however. First, *software interrupts* (i.e., those triggered by an `int` instruction) are part of the code of the program and are treated in much the same way as a standard function call. Second, the x86 architecture, starting with the Pentium,

```
GET_ARG 2
MALLOC          ; placeholder
MOVL A0, ESP
LDL T0, A0      ; 0x1e7beb4
ADDL ESP, 0x10
JMP T0
```

Figure 6. A malloc replacement that summarizes the effect of `RtlAllocateHeap` in Windows using QEMU micro-ops. The summary reads the size of the allocation from the stack, has a placeholder op for the actual allocation, reads the return address, cleans arguments from the stack, and executes the return.

includes an Advanced Programmable Interrupt Controller (APIC) unit, which gives the operating system more control over how and when it receives interrupts. Among the capabilities provided by the APIC is the ability to send interrupts to other processors (including itself). When combined with the ability to defer interrupts by setting the Task Priority Register (TPR) on the APIC, this provides a mechanism that can be used by the OS to perform *asynchronous procedure calls*. Because these are effectively software interrupts (they originate in OS code), we include them in our analysis and in the generated program.

Finally, we provide replacements for the memory allocation functions of some operating systems. To do so, we make use of three pieces of domain knowledge about the OS being analyzed: the virtual address of the function, the number of argument bytes (to enable callee-cleanup, if the calling convention requires it), and the position of the "size" argument relative to the stack. With this information, one can edit the trace and replace a call to (e.g.) malloc with a summary that allocates memory on the host. A summary for RtlAllocateHeap on Windows is shown in Figure 6. We currently replace `RtlAllocateHeap`/`RtlFreeHeap` [23] on Windows, and `malloc`/`realloc`/`calloc`/`free` on Linux. In our testing, programs generated for the Haiku operating system did not need malloc summaries.

C. Dynamic Slicing and Trace Merging

After preprocessing the trace, our Trace Analyzer uses *executable dynamic slicing* [17] to trace the flow of information through the program, starting with the output buffer specified in the log. For a detailed discussion of the algorithm, refer to Korel and Laski [17]; however, we will summarize the idea here. Note that because we do not have access to the full program, our dynamic slicing algorithm cannot calculate full control dependency information as in the original dynamic slicing algorithm. Instead, we include *every* control flow statement (and its dependencies) that was observed to have more than one successor in the dynamic control flow graph. This is safe (in the program analysis sense of the word), but may over-approximate the dynamic slice.

Dynamic slicing works backward through the trace, looking for instructions that define the desired output data. For each instruction, the set of data *defined* by the instruction is examined. If the data it defines overlaps with the working set, then the instruction handles tracked data, so we must add

Algorithm 1 Trace Merging Algorithm. P is the final, merged program, which consists of a number of $blocks$ and corresponding successors. T is the set of traces. Each trace $t \in T$ consists of a sequence of ops. $slice[t]$ is the subset of ops in t that are needed, initialized with a dynamic data slice on the output. $dyn\_slice()$ is an instantiation of the dynamic slicing algorithm, and $find\_block()$ identifies a basic block to which an op belongs.

```
1:  $slices \leftarrow \emptyset, blocks \leftarrow \emptyset$
2:  for $t \in T$ do
3:      $slice[t] \leftarrow dyn\_slice(t, t.output)$
4:      $blocks[t] \leftarrow split\_basic\_blocks(t)$
5:  end for
6:  repeat
7:      $P.blocks \leftarrow \emptyset, P.succs \leftarrow \emptyset$
8:      for $t \in T$ do
9:          for $op \in slice[t]$ do
10:             $op\_block \leftarrow find\_block(blocks[t], op)$
11:             if $(op\_block \in P.blocks)$ then
12:                 $p\_op\_block \leftarrow find\_block(P.blocks, op)$
13:                 $p\_op\_block.add(op)$
14:             else
15:                 $P.succs[op\_block] \leftarrow op\_block.succs$
16:                 $P.blocks \leftarrow P.blocks \cup new\_block(op)$
17:             end if
18:         end for
19:     end for
20:     for $block \in P.blocks$ do
21:         if $P.succs[block].size > 1$ then
22:             $conds \leftarrow find\_exit\_conds(block)$
23:             for $t \in T, block \in t$ do
24:                 $br\_ops \leftarrow dyn\_slice(t, uses(conds))$
25:                 $slice[t] \leftarrow slice[t] \cup br\_ops$
26:             end for
27:         end if
28:     end for
29:     for $t \in T$ do
30:         for $op \in slice[t]$ do
31:             for $t\prime \in T, t\prime \neq t, op \in t\prime$ do
32:                 $slice[t\prime] \leftarrow slice[t\prime] \cup op \cup dyn\_slice(t\prime, uses(op))$
33:             end for
34:         end for
35:     end for
36: until nothing changed
```

it to the slice. The working set is then updated by removing the data defined by the instruction and adding the data *used* by the instruction. The algorithm terminates when top of the trace is reached. The output, an *executable dynamic slice*, is precisely the sequence of instructions used to compute the data in the output buffer.

Virtuoso builds the final introspection program from a training set of more than one trace. This is crucial to achieving reliability: except in the case of fairly trivial introspections, a single trace is unlikely to include enough code to adequately cover all important functionality. Consider the fact that many of the kinds of introspections we want to support, such as enumerating running processes or loaded modules, involve traversing collections such as linked lists or arrays. Typically, the code that accesses these data structures includes logic (and therefore multiple paths through the program) to cover corner cases, such as when the collection is empty. If we want our final introspection tool to be able to handle these uncommon (but hardly rare) situations correctly, we must build our program from multiple traces. In practice, to ensure reliability, we generate increasingly reliable programs iteratively: after generating an initial program, it is tested on a variety of system states, and the failing test cases are then used as new training examples. Although this cycle of testing and training is currently done manually, it could easily be automated.

The trace merging algorithm works as follows. First, a merged control-flow graph is constructed from the control flow graphs of the individual traces (lines 8–19 in Algorithm 1). Second (lines 20–28), for each block which has more than one successor (meaning there was a branch or a dynamic jump), a slice is performed on the data neccessary to compute the branch condition (i.e., x86 status and condition flags or the dynamically computed jump target); this step ensures that both loops and branches are faithfully reproduced in the generated program. Third, Virtuoso performs a *slice closure* (lines 29–35): if an op is in some but not all of the slices, then it is added to the other slices, and a new dynamic slice is performed on the dependencies of that op (this has the same effect, and is done for the same reason, as Korel and Laski's Identity Relation (IR) [17]). Finally, because each of these steps may have introduced new ops into the slices in ways that require recomputing information in previous steps, we repeat the process until a fixed point is reached.

1) Correctness: We argue the correctness of our trace merging algorithm in two parts. First, the algorithm certainly reaches a fixed point and terminates within a finite number of iterations. The traces used as input are finite in number and length. This means that the number of micro-operations initially marked as "in a slice" by the dynamic slicing algorithm of Korel (which has, itself, been shown to complete in a finite number of steps) is also finite. Thus, the number of micro-operations as-yet unmarked is also finite. With each iteration of the algorithm, we either add no new ops to a slice, in which case we have reached a fixed point and terminate, or we add some finite number of ops to some slices and loop. Thus, the algorithm will, at worst, add one op to each slice with each iteration, ending with every op in every trace marked, but even this will happen in a finite number of iterations of the algorithm.

Second, to see that our algorithm correctly merges multiple traces, consider combining traces T and T', each of which (when we run our algorithm on it individually) produces programs P_T and $P_{T'}$, which each have the correct output given their respective input (we can assume this because in this base case our algorithm is equivalent to that of Korel and Laski [17]). By examining Algorithm 1, when we produce a program P from the combination of T and T', we can see that there are only two cases where P will differ from P_T or $P_{T'}$:

Case 1: We include a branch statement that was not included in T or T' by itself. But by including this branch, we include the ops necessary to decide which way to go at that branch. So when P is run with the inputs provided to T, the branch will evaluate as it did in T, and the successor will be the same as for P_T. The same argument applies when P is run with the inputs for T'.

Case 2: We include an op (and the ops necessary to compute the dependencies of that op) that was in T and not T' (or vice versa). Assume without loss of generality that the op was in T and not T'. Then the ops added to T' do not affect the output value along the path taken by T' (or they would have been added by the dynamic slice that initialized $slice(T')$). So when P is run with the inputs for T', it will produce the same output as $P_{T'}$.

This argument extends by induction to show that given an arbitrary number of traces $\{T_1, \ldots, T_n\}$, if the corresponding programs $\{P_1, \ldots, P_n\}$ are correct with respect to their inputs, the merged program will be correct with respect to all their inputs. We have also validated the correctness of our algorithm experimentally, as we describe in Section VI-B.

2) Fortuitous Coverage Enhancement: We have demonstrated that a merged program P can surely execute correctly on guest states from one of its constituent traces. However, aside from this additive effect, combining traces can also allow the program to generalize better to guest states not seen during training. To see why this is the case, consider Figure 7, which depicts two program paths that are merged by our algorithm. In addition to the two paths covered by each individual trace ($A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$ and $A \rightarrow B \rightarrow D \rightarrow F \rightarrow G$), the merged program also covers paths $A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$ and $A \rightarrow B \rightarrow D \rightarrow E \rightarrow G$). Depending on the relationship between the branch conditions at A and D, these paths may be infeasible, but in the best case, the merged program will be able to generalize to guest states that differ from the training states along these paths.

In our experience with Virtuoso, we have noticed several instances of this "fortuitous" coverage improvement. For example, while testing the **pslist** program for Windows (see Section VI), we generated two versions of the program from two different training runs. Out of three system states in our testing corpus, we observed the following behavior:

- The first program worked correctly on states 1 and 2, but not state 3.
- The second program worked correctly only on state 1.
- The merged program, generated from both training runs, worked correctly on all three system states.

While this is an isolated example, it suggests that such non-additive coverage improvements can boost the reliability of the generated introspection programs. We hope to make

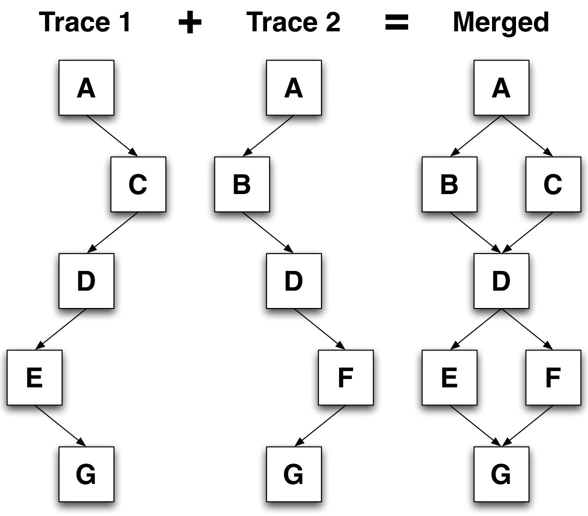


Figure 7. An example of a non-additive coverage improvement from combining two traces. Rather than the expected two paths covered, we in fact cover four program paths.

this notion more rigorous in future work, and explore its potential for producing more reliable introspection programs in conjunction with static analysis.

D. Translation and Runtime Environment

Our Instruction Translator (shown below the Trace Analyzer in Figure 1) takes the sliced micro-op traces and creates an equivalent Python program that can be run outside the VM. Currently, the code produced takes the form of a plugin for Volatility [32], a framework for volatile memory analysis written in Python. Volatility was chosen because it already provides certain facilities needed in our runtime environment, such as x86 virtual address translation, and the ability to analyze the memory of a virtual machine running under Xen [2] (through the third-party extension PyXa, included in the XenAccess library [26]). Moreover, it features an API that makes it easy to layer new functionality on top of existing address space objects, which we use to implement copy-on-write semantics for the guest memory (see below).

The translation of QEMU micro-instructions to host code is fairly direct. The translation works one basic block at a time, starting with the output of Algorithm 1. Each individual op is mapped to a simple Python statement. At the end of the block, if there is only one successor, an unconditional jump to that successor is produced. Otherwise, the appropriate conditional or dynamic jump is output.

The runtime environment itself, shown on the right in Figure 1 is implemented as a plugin for Volatility. The introspection tool is output by the trace analyzer as a dictionary of blocks of Python code, keyed by the EIP of the original x86 code. The plugin performs some initial setup, and then loads and executes the block marked "START". When the block finishes, it sets a variable named `label` that determines the successor (this variable can be set conditionally, unconditionally, or dynamically, to implement the three successor cases outlined above). This process continues until the successor returned is "EXIT". At the end, the contents of the output buffer are displayed to the user. The details of the runtime environment, including input and output handling, are presented below.

Registers are modeled as variables in the generated code; thus, to implement copy-on-write behavior for registers, we simply initialize the values of the register variables before the generated code. Any references to a register that occur before the generated code assigns to it will therefore retrieve the value from the guest VM, as desired. Assignments to registers, by contrast, will simply overwrite the Python variable, and will not affect the running VM.

Access to memory is handled by translating load and store operations into `read` and `write` operations on a copy-on-write address space object in Volatility. The COW space is initialized by passing it the currently active virtual address space for the VM (on x86 guests, this boils down to using the current value of the guest's `CR3` register to map virtual addresses to physical). All reads and writes are then done on this space, which implements copy-on-write as described in Section IV-C.

The runtime environment also provides a host-side memory allocator for use with the malloc summaries described in Section V-B. Before the program is run, the runtime environment scans the guest page tables for an unused virtual address range. It then adds page table entries (as always, making changes to the copy-on-write space—the guest's state is not modified) to create a heap that can be addressed as though it were actually inside the guest. To prevent conflicts, the physical pages are reserved by choosing a range above the amount of physical memory available to the guest.[5] Reads and writes to that physical range are redirected to the host memory area.

Inputs to the program are represented by an array named `inputs`. When the Instruction Translator comes across a micro-op that is marked as reading input data, it translates it into a simple assignment that takes the value from the input array. For example, if a micro-op such as `LDL T0, A0` is marked as needing the first input parameter to the program, the code produced will be `T0 = inputs[0]`. The `inputs` array is populated at runtime via a command line option to the Volatility plugin.

Finally, output-producing operations are translated into two separate Python statements. The first performs the write to memory normally, while the second writes the data into a special output memory space and tags it with a label that identifies which output buffer it was associated with. Once the plugin has finished executing the translated code,

[5]Although this means that in a 32-bit environment we cannot support guests with 4 gigabytes of RAM, we note that this is only a peculiarity of our implementation; see Section VII for details of how we plan to remove this limitation. Also, the move to 64-bit architectures will provide ample physical address space for Virtuoso to use if needed.

it dumps the contents of the output memory space. The user can also optionally pass in a function to interpret the output (e.g., by converting it from little-endian Unicode to a string, or to interpret a 64-bit integer as a human-readable time).

VI. Evaluation

In this section, we evaluate Virtuoso using a number of different criteria: generality, reliability, security, and performance. To evaluate the generality of Virtuoso, we look at the diversity of operating systems and the different types of programs for which introspections can be generated. Next we discuss the *reliability* of the resulting introspection programs—that is, their ability to function correctly when examining a system at runtime. Finally, we examine the resilience to attack of programs output by Virtuoso, and we discuss their runtime performance.

A. Generality

Because Virtuoso uses very little domain knowledge about the target OS, it is easy to generate new introspection programs for operating systems that run on the x86 architecture. To demonstrate this capability, we created six training programs for three different OSes. These programs were chosen because they represent common actions that might be needed for passive and active security monitors: a security system may need to list drivers to audit the integrity of kernel code, or enumerate processes in order to scan them for malicious code, and so on. Some of these functions are also implemented for Windows targets in tools such as Volatility [32]; however, our own tools required no reverse engineering and were generated automatically.

The six programs performed the same function on all three operating systems:

- **getpid** Gets the process ID of the currently running process.
- **gettime** Retrieves the current system time.
- **pslist** Computes a list of the PIDs of all currently running processes.
- **lsmod** Retrieves a list of the base addresses of all loaded kernel modules.[6]
- **getpsfile** Retrieves the name of the executable associated with a given PID.
- **getdrvfile** Retrieves the name of a loaded kernel module, given its base address.

The three operating systems used were Windows XP SP2 (kernel version 5.1.2600.2180), Ubuntu Linux 8.10 (kernel version 2.6.27-11), and Haiku R1 Alpha 2 [12]. Windows and Linux were chosen because they are both in common use,[7] and thus represent practical, real-world scenarios. Haiku, a relatively obscure clone of BeOS, was chosen for two reasons: (1) to our knowledge, we are the first to do virtual machine introspection or memory analysis on a Haiku target, making it a compelling example of Virtuoso's ability to deal with diverse operating systems; and (2) none of the authors had any prior knowledge of the internals of the Haiku kernel, so there was no way to "cheat" by incorporating additional domain knowledge.

[6]In the Linux version, this reads the contents of `/proc/modules`, which contains both the names and base addresses of kernel modules, so `getdrvfile` is not needed on Linux.

[7]Although use of OS X is also widespread, its design isolates the kernel in a separate address space. Due to limitations in our current implementation discussed in Section VII, we were unable to test OS X introspection.

OS	Program	Original	Post	Final
Windows	getpid	3549	106	12
	gettime	7715	1081	233
	pslist	302082	230366	75141
	lsmod	195488	157440	84973
	getpsfile	49588	23865	10074
	getdrvfile	194765	157696	92109
Linux	getpid	133047	4055	1706
	gettime	75074	3882	1592
	pslist	6107214	2265667	1095963
	lsmod	1936439	852299	414670
	getpsfile	14752561	6323249	2913064
Haiku	getpid	18242	3985	1719
	gettime	9982	585	237
	pslist	362127	290078	160830
	lsmod	850363	702277	423438
	getpsfile	249663	152896	78107
	getdrvfile	522299	399175	234527

Figure 8. Results of testing a total of 17 programs across three different operating systems. "Original" refers to the size of the trace before any processing, "Post" is the size after interrupt filtering and malloc replacement, and "Final" gives the size of the program after slicing; all sizes given are in number of IR ops. The numbers given refer to processing of a single dynamic trace.

The results of our testing are shown in Figure 8. To verify that each program worked correctly, we ran it on a sample memory image taken at a different time from when the trace was captured, and then verified the output by hand (with help from existing in-guest tools). The numbers shown are for programs generated from a single trace.

With one exception, the generated programs all worked correctly—`gettime` for Haiku failed to obtain the correct system time. Upon inspection, we determined that this was due to the way Haiku keeps time: rather than continuously updating some global location, Haiku stores the boot time and then calculates the current time by calling `rdtsc` to read the Time Stamp Counter (TSC), which contains the number of cycles executed since boot. Because our runtime environment does not have access to the TSC when operating on a memory image, the time returned by `gettime` is incorrect. However, we manually verified that when provided with the correct TSC value, the generated program functions correctly.

Figure 8 also shows the effect of preprocessing and slicing on the size of the generated program. One clear point emerges from examining these figures. Preprocessing removed between 17% and 97% of the initial traces (it removed 55% on average), and slicing reduced the number

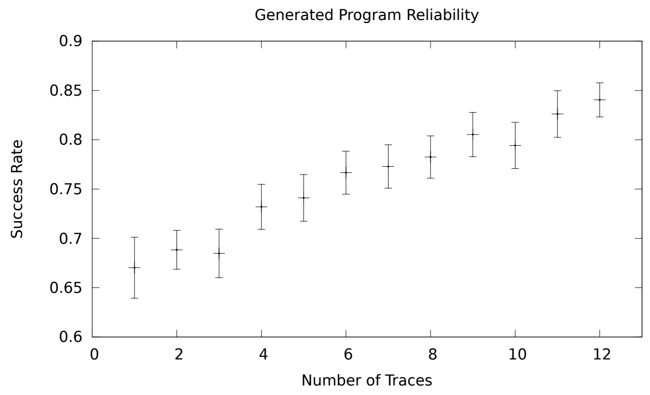


Figure 9. Results of cross-validation for the `pslist` program on Windows. The error bars indicate the standard error of each sample.

of remaining instructions in the trace by 40%–89% (56% on average). Because the resulting programs do, in fact, work, we can conclude that a large portion of the computation seen in the initial traces is irrelevant to the final output. Also, we can observe that were we to run all the code seen in the trace, the time required to run the introspection programs would dramatically increase.

B. *Reliability*

Because our introspection programs are based on dynamic analysis of the training programs, they do not, in general, contain all the code needed to account for every eventuality the introspection program may encounter. This limitation is inherent to the use of dynamic methods and is a well-known problem in program testing. In this section, we describe the results of our experiments on the reliability of the generated introspection programs, techniques for improving their coverage, and the motivation behind our use of dynamic analysis.

Testing: To empirically test the reliability of programs generated using Virtuoso, we examined how the reliability of a single program (`pslist`, described in Section VI-A) was affected by training. We generated 24 traces of `pslist` running under Windows XP, taking a trace once every five minutes, starting just after the desktop appeared. During this time, we did the following actions:

- Opened a connection to a remote machine with PuTTY.
- Browsed web pages with Google Chrome and Internet Explorer.
- Played a game of Minesweeper.
- Closed a non-responsive instance of Internet Explorer, and sent a crash report.
- Played a game of Solitaire.

No special effort was made to induce a wide variety of system states such as low-memory conditions. Along with each trace, we also captured a snapshot of the memory and CPU state.

We then used cross-validation to estimate the reliability of the program as a whole: for each k in the range $\{1 \dots 12\}$ we took 50 random subsets of size k from the set of all traces captured,[8] used these traces to generate an introspection program, and then tested the reliability of the program on the $(24 - k)$ images corresponding to the remaining traces. We judged that the generated program had executed correctly if it produced the same output as the training program for that image. Figure 9 shows the results of this testing. Each point represents the mean reliability of programs generated from k traces.

We note that reliability appears to increase linearly with the number of traces used; since the reliability is bounded above by 1, it most likely approaches an asymptote as more traces are added and the number of unexplored paths in the program grows smaller. We also ran the generated programs on their training images, and, as expected, achieved a success rate of 100%, validating the correctness of our trace merging algorithm.

In examining the data more closely, however, we observed that traces taken earlier in time (closer to system boot) were more reliable. To test this effect, we created an additional test data set of 24 images, again captured once every five minutes starting shortly after system boot. We then generated the `pslist` program from the first trace captured during our cross-validation test and no others. To our surprise, we found that this program had 100% reliability on our new test data set.

By inspecting the source code to the Windows Research Kernel [24], we determined that the Windows operating system stores the image name of the process *lazily*, waiting to generate the attribute until the first time some program requests it. So the first trace includes all the code needed to generate the process name, whereas later traces simply read the already-initialized data. When programs produced from these latter traces encounter a case where the process name is not yet initialized, they will lack the code necessary to do so. Thus, this caching effect may cause the cross-validation numbers given above to overestimate the difficulty of building reliable programs; in this case, the natural testing strategy of booting the system and taking several traces would have yielded a reliable program.

Improving Coverage: In any case where testing finds that the generated program fails to function correctly, we can take a new trace based on the failing snapshot that should work correctly in that case. Although we will not be able to run the program precisely from the point where the failure occurred, in practice we have found that failing states tend to persist long enough that we can take a new trace that covers the failed snapshot. This allows us to iteratively improve coverage based on gaps found during training, until we reach a point where no further gaps are found. In future work, we hope to explore how techniques from the software testing community can help make this approach more rigorous. For example, we may be able to perform static analysis of the OS code to reveal missing branches, and then apply symbolic

[8]For $k = 1$ there are only 24 subsets, so our number of subsets in this case is 24.

or concolic execution to determine what system state is necessary to go down that branch and improve coverage.

Static vs. Dynamic Analysis: A natural question is why, given the coverage issues associated with dynamic analysis, our approach does not make use of static analysis. Although we considered static analysis, a number of difficulties make it an unattractive choice in this context. First, static analysis relies on the ability to reliably identify all the code associated with a piece of functionality. In the general case, accurate disassembly of x86 code is undecidable [22], and even on non-malicious code, it can be quite difficult. Second, our slicing approach would be much less accurate if performed statically, particularly in the presence of dynamic jumps (which are common in kernel code, which makes extensive use of function pointer-based indirection) and dynamically allocated memory. Finally, static analysis requires much more domain knowledge about the target operating system: at minimum, the executable file format and the details of the loader must be known just to find the code to analyze.

To demonstrate the reasons why static analysis is inappropriate in this case, it is useful to look at a recent system, BCR (Binary Code Reutilization) [5], which also attempts to extract portions of a binary program for later re-execution, but does so using static analysis. First, we note that BCR does not use a purely static approach: it relies on dynamic analysis in its disassembly phase to resolve the targets of dynamic jumps. The authors do not clearly indicate how their system deals with any coverage issues that may arise from this use of dynamic analysis. Second, BCR employs significant amounts of domain knowledge to perform its code analysis. The executable file format, system API calls, and the mechanism by which imported functions are resolved must all be known in order to extract assembly functions. This reliance on domain knowledge both restricts the generality of BCR and limits the kinds of code it can extract—for example, BCR is unable to extract functions that make system calls directly. Virtuoso, by contrast, relies on no such domain knowledge and works with any code constructs supported by the x86 architecture.

Even if it were possible in our case to statically find every code path that could be taken by the program, it may not be desirable. For example, consider the case of a program written for Linux that retrieves sensitive information from the `/proc` filesystem. At some point in the execution of this program, the kernel will likely perform a check along the lines of `if (uid == 0)`. Although this check is a valid part of the overall program, it does not actually affect the *value* computed by the program. If we use dynamic analysis and collect traces that succeeded, we will only ever see the branch where this condition evaluated true, and the check will not be incorporated into the resulting program. Although this is, strictly speaking, incorrect, it has the side effect of *generalizing* the program: whereas the original code could only be executed in a context where the current user was root, the generated introspection utility will be able to compute the correct result from the context of any user. Additional research is required to determine if these "undesirable" branches can be automatically eliminated using a static approach.

C. Security

The main motivation in the design of Virtuoso is to enable the rapid creation of secure introspection-based programs. Were security and unobtrusiveness of no concern, it would be sufficient to deploy an in-guest agent that queried existing OS APIs and simply trust that they had not been compromised. However, as security *is* a concern, we must test that our system does, in fact, provide the hoped-for isolation from malicious changes to the guest operating system.

To demonstrate that the generated programs are immune to malicious changes in kernel code, we generated our Windows process lister (`pslist`, described in Section VI-A) on a clean Windows XP system. Next, we obtained the Hacker Defender [9] rootkit, which (among many other stealth techniques) hides processes by injecting into all processes on the system and hooking API calls using inline code modification. We configured it to hide any process named `rcmd.exe`. We then renamed a copy of Notepad to `rcmd.exe`, and infected the system. After infection, we checked that `rcmd.exe` was no longer visible from the Task Manager. We then ran our generated introspection program, and found that it successfully listed both our hidden process and the rootkit's own userland component (`hxdef100.exe`).

Beyond verifying that our program is immune to existing rootkits that alter kernel code, we also need to consider the possibility that someone might actively attempt to evade our introspections. First, because Virtuoso attempts to faithfully replicate the functioning of actual system APIs, vulnerabilities that allow attackers to evade the standard APIs will be reflected in the generated introspection programs. Data-only attacks (i.e. Direct Kernel Object Manipulation, or DKOM) also poses a challenge to our system, because in this case even uncompromised OS APIs will report incorrect results. For example, the standard Windows process listing API can be evaded by rootkits that directly manipulate the process data structure to unlink a malicious process, hiding it from the `EnumProcesses` API. Careful combinations of existing introspections can still thwart this attack, however: by calling **getpid** every time the `CR3` register changes and comparing this list against the one returned by **pslist**, hidden processes can still be detected. Further research is necessary to see if such defenses can be found for other kinds of data-only attack.

Second, an attacker may attempt to take advantage of the fact that Virtuoso uses dynamic analysis to cause it to malfunction. Specifically, he may attempt to set the *free variables* used by the program in such a way as to cause some conditional branch that does not have full coverage to be exercised. The opportunity for evasion afforded by this is minimal: if an attacker does find a way to reliably manipulate the system into a state that causes an introspection program to fail, the fix can be generated quickly, as the attack could be used for a new training run that would update the existing introspection program. Extra care must be exercised in this case, however: if the attack also alters code that the introspection depends upon, updating the introspection program could cause malicious code to be incorporated into the generated program. Kernel code integrity checks (i.e., verifying that the code seen in training matches that found in an uncompromised version of the OS) would help in this scenario, but can be difficult to implement in practice, and would require more domain knowledge than we currently assume.

Additionally, we note that our dynamic slicing method allows us to quantify precisely the set of global kernel data that a given introspection depends on (note that data from userland is read from the training data, and is not available to the attacker for manipulation). Even within this set of kernel data, only a subset may be susceptible to attacker manipulation: as Dolan-Gavitt et al. [8] describe, arbitrary modification of global kernel data can cause system instability. Ultimately, this problem is best resolved by improving training, and we hope to explore automated means of improving coverage in future work.

The general problem of defending against malware that is VMI-aware, and has the ability to tamper with kernel data structure layouts and algorithms, remains open. Such malware, which would affect most current VMI solutions, has been discussed in other work [1], and defending against it is a difficult problem. Although we consider this problem out of scope for Virtuoso, we hope to explore more general defenses in future work.

Finally, if a vulnerability is found in the APIs that support the introspection, an attacker may attempt to compromise the security of the monitor by causing it to execute malicious code. Because the runtime environment has no facility for translating new code, it is effectively a Harvard architecture, so standard code injection techniques are not possible against programs generated by Virtuoso. However, in recent years a new technique, *return-oriented programming* [29], has demonstrated that malicious computation can be performed by chaining together snippets of code linked by dynamic jumps (e.g., returns), and subsequent work has shown that this allows for exploitation of Harvard architecture devices [6]. Despite these developments, we believe that the relatively small size of our programs (the largest in our test set has only 4030 basic blocks, and only 140 of these contain a dynamic jump), combined with the fact that the runtime environment executes code at the granularity of a basic block (i.e., it is not possible to execute only part of a block), will make it impossible to construct the necessary set of gadgets. More work, however, is necessary to prove this intuition definitively.

D. Performance

Performance results for the programs described in Section VI-A are shown in Figure 10. The tests were carried out on a Intel® Core^{TM} 2 Quad 2.4 GHz CPU machine with 4 gigabytes of RAM running Debian/GNU Linux unstable (kernel 2.6.32-amd64). Each test was run 100 times, and the results were averaged.

OS	Program	Time (ms)
Windows	getpid	0.2
	gettime	1.5
	pslist	450.2
	lsmod	698.1
	getpsfile	59.8
	getdrvfile	751.9
Linux	getpid	9.6
	gettime	9.6
	pslist	6394.1
	lsmod	2437.0
	getpsfile	20723.9
Haiku	getpid	33.7
	gettime	1.8
	pslist	712.1
	lsmod	3351.7
	getpsfile	479.6
	getdrvfile	1901.0

Figure 10. Runtime performance of generated programs. Times given are in milliseconds, and are averaged over 100 runs.

Of the three operating systems shown, the results for Linux stand out as being particularly slow. We investigated this result further and found that the overhead was caused by the interface Linux uses to retrieve system information. In contrast to Haiku and Windows, which have specific system calls to inspect the state of the system, Linux exposes these details through the `/proc` filesystem. This means that the code that implements tools such as `pslist` needs to open files, list directories, and so on, all of which results in much more code being executed than on Windows and Haiku.

Although the times shown are not currently fast enough to enable online monitoring, we stress that our current implementation is unoptimized and translates the x86 code to Python, which it then runs in an environment that is also written in Python. One performance improvement would be to port the runtime to C and generate x86 binary code that can be run on the host (after transforming potentially dangerous instructions into safe equivalents, redirecting memory loads and stores to the guest VM, and so on). We expect that this would provide performance at least on par with QEMU, which uses similar techniques in its whole-system emulation. It is also worth noting that even unoptimized, the programs generated by Virtuoso are fast enough for use in forensic analysis of memory dumps.

VII. Limitations and Future Work

Although Virtuoso currently supports many useful introspections on a variety of operating systems, there are still a number of areas in which it could be extended. In this section, we describe the current limitations of Virtuoso and how we might deal with these cases.

Multiple address space support: Perhaps the most significant limitation of Virtuoso is its inability to correctly handle traces that span multiple virtual address spaces (that is, traces in which another process than the training program appears). In the simplest case, the other processes do no work related to the introspection at hand, but they are difficult to filter out because the trace will also contain portions of scheduler code to switch to the other process, and identifying and removing this code automatically is difficult. Virtuoso currently sidesteps this issue by running its training programs with high priority (e.g., using `start /realtime` on Windows and `chrt` on Linux).

More difficult, however, is the problem of analyzing and extracting programs that make use of interprocess communication (IPC). Solving this problem is particularly important for supporting introspection of microkernel architectures, where many tasks are performed by collections of cooperating processes. The major challenge here is that although we may be able to extract the relevant code running in each process, there will be data from other processes that must be read from the guest operating system at runtime. The generated program, then, will need to have some way of finding the appropriate cooperating process at runtime in order to read this data; this is likely to require significant domain knowledge. More research is required to determine the extent to which this can be automated.

Self-modifying code: As we mention in Section VI-C, the runtime environment of Virtuoso is effectively a Harvard architecture machine, with no facility for loading new code. This means that we do not support self-modifying code. Although this has not caused problems with any of our introspections (operating systems rarely make use of self-modifying code), such support is necessary for working with malicious code, an area we hope to explore in the future.

Relocation and ASLR: Currently, Virtuoso assumes that the modules containing the data it needs have not been relocated. This assumption may not hold in general, however: modules may be loaded at different base addresses for a variety of reasons, including security (i.e., Address Space Layout Randomization, or ASLR).[9] If this occurs, data read from those modules will be found at a different address, and the introspection program may not be able to locate it correctly. Although this problem can be resolved by incorporating more domain knowledge (such as the mechanics of the executable loader), doing so would make supporting new operating systems more difficult. One possible solution to this problem is to attempt to *relocate* portions of the generated program at runtime by scanning the memory of the guest operating system for the relevant code and then applying the appropriate offset for accesses to memory.

Aside from improving the analysis capabilities of Virtuoso, we also hope to examine its use in areas outside the realm of introspection. For example, by tracing calls made by programs such as `ls`, we might be able to produce programs that can understand the format of undocumented filesystems without relying on native filesystem drivers. Such analysis capabilities could aid significantly in cross-platform interoperability efforts.

[9]The alert reader will note that the version of Linux used enables ASLR by default. However, this did not affect our introspections, as the only libraries randomized are in user space, and we take userland data references from training.

Finally, there are some minor implementation details that could be improved. In particular, we could do away with needing to know the details of malloc for specific operating systems by adding support for x86 hardware exceptions and extracting the page fault handler. As we already detect page faults in our runtime environment (they currently raise an exception), we would just need to extract the page fault handler code and cause memory exceptions to trigger this code (similar to the way the TPR is currently handled—see Section V-B for details). This would eliminate the only piece of OS-specific knowledge used by Virtuoso, allowing it to generalize to any x86-based operating system.

VIII. Conclusion

We have presented Virtuoso, a system for automatically generating introspection tools that can retrieve semantically meaningful information based on low-level data sources. By applying a novel whole-system executable dynamic slicing technique, Virtuoso turns a task which once took hours or weeks of reverse engineering by an expert into one that requires only a small amount of effort by a programmer of modest skill and a few minutes of computation time—and in doing so, helps ensure that introspection programs exactly model the behavior of the operating system. Moreover, its analysis capabilities are operating system-agnostic, removing the need for developers to constantly play catch-up as OS vendors release new versions of their products. These contributions help narrow the semantic gap, and should remove a significant roadblock in the areas of forensic analysis, virtualization-based security and low-artifact malware analysis.

Acknowledgments

We would like to thank our anonymous reviwers for their comments, and in particular our shepherd, Jon McCune, for his help in addressing the concerns of those reviewers. We would also like to thank Bryan Payne, Annuska Riedlmayer, Ying Xiao, Patrick Traynor, and Andrew Case for their comments on early drafts of this paper. This material is based upon work supported in part by the National Science Foundation under grants no. CNS-0845309 and no. CNS-0831300, and the Office of Naval Research under grants no. N000140710907 and no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

References

[1] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. DKSM: Subverting virtual machine

introspection for fun and profit. *IEEE Symposium on Reliable Distributed Systems*, 2010.

[2] 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. In *ACM Symposium on Operating System Principles (SOSP)*, 2003.

[3] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2009.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (USENIX ATC)*, Anaheim, CA, 2005.

[5] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2010.

[6] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *Electronic Voting Technology / Workshop on Trustworthy Elections (EVT/WOTE)*, Montreal, Canada, 2009.

[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Computer and Communications Security*, Alexandria, VA, 2008.

[8] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *ACM Computer and Communications Security (CCS)*, Chicago, IL, 2009.

[9] F-Secure. Rootkit:W32/HacDef description. http://www.f-secure.com/v-descs/hacdef.shtml.

[10] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2003.

[11] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2003.

[12] Haiku OS project. http://haiku-os.org/.

[13] X. Jiang, D. Xu, and X. Wang. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *ACM Computer and Communications Security (CCS)*, Alexandria, VA, 2007.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[15] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, Montreal, Canada, 2009.

[16] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 2010.

[17] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

[18] A. Lanzi, M. I. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2009.

[19] M. Laureano, C. Maziero, and E. Jamnhour. Intrusion detection in virtual machine environments. In *Euromicro Conference*, 2004.

[20] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2011.

[21] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2010.

[22] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Computer and Communications Security (CCS)*, Washington, D.C., 2003.

[23] Microsoft Corporation. RtlAllocateHeap on MSDN. http://msdn.microsoft.com/en-us/library/ff552108(VS.85).aspx.

[24] Microsoft Corporation. Windows research kernel. http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx.

[25] Mission Critical Linux. Core analysis suite (crash). http://www.missioncriticallinux.com/projects/crash/.

[26] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, 2007.

[27] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 2008.

[28] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, San Diego, CA, 2004.

[29] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Computer and Communications Security (CCS)*, Alexandria, VA, 2007.

[30] A. Slowinska, T. Stancescu, and H. Bos. Towards precise data structure recognition in stripped binaries. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2011.

[31] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. *Recent Advances in Intrusion Detection*, 2008.

[32] A. Walters. The Volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.

[33] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Recent Advances in Intrusion Detection*, Cambridge, MA, 2008.

[34] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security & Privacy*, 5(2), March 2007.