A Design for Comprehensive Kernel Instrumentation

Peter Feiner peter@cs.toronto.edu
Angela Demke Brown demke@cs.toronto.edu
Ashvin Goel ashvin@eecg.toronto.edu

University of Toronto

Abstract

Dynamic binary instrumentation (DBI) has been used extensively at the user level to develop bug-finding and security tools, such as Memcheck and Program Shepherding. However, comprehensive DBI frameworks do not exist for operating system kernels, thwarting the development of dependability and security tools for kernels. In this paper, we identify the key challenges in designing an in-kernel DBI framework and propose a design that addresses them.

1 Introduction

Dynamic binary instrumentation (DBI) entails monitoring and potentially manipulating every instruction in an existing binary before its execution. Several popular frameworks, such as DynamoRIO [4], Pin [12], and Valgrind [14] make it easy to use DBI in user-level applications, helping improve application dependability greatly. For example, DBI is used by Memcheck to detect memory errors [19], Program Shepherding to improve security [11], and vx32 to enforce modularity [9].

Unfortunately, these powerful debugging and security tools do not exist for operating system kernels. The reason is that existing DBI techniques that target kernel code (e.g., JIFL [15] and PinOS [6]) are not comprehensive, i.e., they are limited with respect to the code that they cover. JIFL provides an API for instrumenting system calls. However, it does not cover interrupt handlers and kernel threads. PinOS allows whole-system instrumentation, providing an API for instrumenting kernel code, including interrupt handlers, and user-space code. However, because PinOS relies on virtualization, it is only capable of instrumenting drivers for devices that the virtual machine monitor emulates. Since it is infeasible to emulate complex and proprietary hardware, PinOS's approach precludes comprehensive instrumentation. A similar issue arises with other virtual machine monitors [1] and emulators [20] that use DBI.

This lack of coverage severely limits the utility of in-kernel applications of DBI. Without comprehensive coverage, debugging tools that track memory are inaccurate and security tools can easily be thwarted. Without being compatible with most device drivers, most driver code cannot be debugged or secured – an unfortunate limitation because drivers often contain the most buggy code in operating systems [8].

A comprehensive in-kernel DBI framework would enable many dependability-enhancing techniques. Such a framework would make it much simpler to implement or port some existing user-space techniques, such as Program Shepherding and Memcheck [17], to the kernel. Moreover, existing dependability techniques based on static instrumentation (i.e., requiring source code recompilation) could be made transparent with DBI. For example, Byte Granularity Isolation [7], which isolates device drivers that are generated with a special compiler, could be adapted to use DBI on existing driver binaries.

This paper makes two contributions: 1) we identify the challenges in comprehensively instrumenting operating system kernels, and 2) we propose the design of a novel DBI framework that addresses these challenges. Comprehensive in-kernel DBI raises five key challenges outlined below.

Detecting Kernel Execution A specific DBI tool might be interested in instrumenting a subset of kernel code, such as driver code. However, ensuring comprehensive instrumentation requires monitoring several data structures and registers that can be modified whenever the CPU is running in supervisor mode. Thus it is necessary for the framework to interpose on all code running in supervisor mode, i.e., the entire kernel must be instrumented. The resulting challenge lies in detecting all kernel entry and exit points.

Interrupts and Exceptions Comprehensiveness requires instrumenting exception and interrupt handlers. Faithfully emulating native handling of an interrupt raised during instrumentation poses several challenges, such as providing a plausible exception stack frame.

Concurrency Instrumenting a multithreaded kernel requires careful management of shared data structures. We address the trade off between memory use, implementation complexity, and performance in designing a concurrency strategy. Kernel preemption further complicates concurrency management.

Reentrancy A region of code is non-reentrant when it is unsafe to begin executing it before other executions of the region have finished. Reentrancy is problematic when the instrumentation code needs to use the instrumented code. The standard solution – reimplementing non-reentrant code – is not viable in the kernel.

Code Cache Consistency Kernel code is copied before instrumentation. If the kernel code changes, e.g, on a module unload, then the instrumented copy needs to be updated. Moreover, changes to code permissions need to

be identified and handled.

In the rest of the paper, we describe these issues and our design for addressing them in detail. Currently, we are in the process of implementing and evaluating this design. Below, we first provide some background on DBI.

2 Background

DBI implementations copy basic blocks of source code (binary x86 code residing in memory that the program counter normally points to) into a *code cache* prior to their execution. A basic block is a straight-line sequence of instructions that ends with a control transfer instruction (CTI), such as `call, ret,` or `jmp.` The DBI implementation ensures that only cached code is executed. To ensure that source code never executes, the DBI implementation manipulates the CTIs that terminate basic blocks to return execution to the *dispatcher*. If the CTI's target has already been copied into the cache, the dispatcher transfers control to the target's copy. Otherwise, the dispatcher interprets the target's code, which involves finding the extent of the basic block starting at the target's address. The block found by interpretation is copied into the code cache. Finally, the dispatcher transfers control to the newly admitted block.

3 Analysis and Design

We present the design of our DBI framework in terms of the challenges outlined in Section 1. We also describe current approaches, their strengths, and their limitations. Our design targets the *long mode* of the 64-bit x86 architecture [2], which we assume the instrumented operating system, once booted, runs in exclusively. This assumption is valid in Linux, which we confirmed by manual inspection, and Windows [13].

3.1 Detecting Kernel Execution

Interposing on all kernel execution requires each kernel entry point to be replaced with a call to the dispatcher (see Figure 1). We define kernel entry points as any instructions that run immediately after escalation of the CPU's current privilege level (CPL). In the 64-bit x86 architecture, only interrupts, the `syscall` instruction, the `sysenter` instruction, certain far `jmp` instructions, and certain far `call` instructions can escalate the CPL [2] (see [18] for a more concise explanation). In the case of interrupts, far jumps, and far calls, the entry point is stored in one of three descriptor tables. The address of each of these tables is stored in one of three corresponding descriptor table registers. In the case of `syscall` and `sysenter,` the entry point is stored in a model-specific register (MSR). To execute the dispatcher on kernel entry points, instead of executing the entry points directly, we *shadow* the descriptor tables and the MSRs.

We maintain a *shadow descriptor table* for each descriptor table maintained by the kernel; we point the descriptor table registers to these shadow tables. In the shadow tables, each entry effectively calls dispatch on the corresponding entry point stored in the original descriptor table. To maintain transparency, we replace source code instructions that read the descriptor table registers with code cache instructions that load the original tables' addresses, which are stored in memory. Our framework intercepts changes to the original tables and propagates the changes to the shadow tables. Furthermore, we replace code that changes the table descriptor registers in the code cache with instructions that update the shadow registers. We shadow the MSRs used by `sysenter` and `syscall` in the same way as the descriptor table registers.

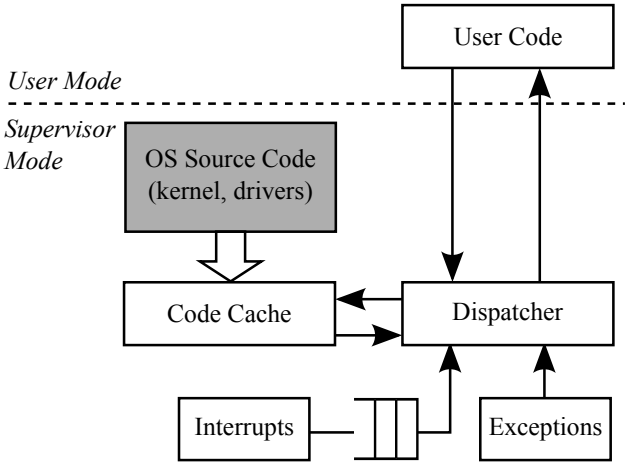

Figure 1: The framework is activated by loading a kernel module that redirects all kernel entry points to the dispatcher (solid black arrows). Once the framework is active, all supervisor-mode execution takes place in the dispatcher and the code cache. Interrupts are queued and their handlers are dispatched at safe points. Exception handlers are dispatched immediately. The fat white arrow indicates that OS source code is copied into the code cache.

The kernel can be exited by `sysexit, sysret, iret,` and `ret` instructions. To resume native execution after exiting the kernel via `sysexit` and `sysret,` we replace these instructions in the code cache with sequences that exit the code cache in a manner that informs the dispatcher to natively execute `sysexit` and `sysret` respectively. Section 3.2 explains how we handle `iret`. The dispatcher identifies kernel exits caused by far `ret` instructions.

3.2 Interrupts and Exceptions

The basic DBI technique rewrites CTIs to ensure that execution stays in the code cache or is redirected to the dispatcher. However, this technique is not adequate for inter-

posing on control flow that does not arise from CTIs, such as interrupts and exceptions.

When an interrupt or exception is delivered, control is redirected to the installed handler. We define *interrupts* to be asynchronous events, not caused by any particular instruction, that redirect execution to their handlers. We define *exceptions* to be synchronous events, triggered by instruction execution, that immediately invoke their handlers. The key distinction between the two is that interrupts can technically be handled at any time whereas exceptions must be handled before the triggering instruction can proceed. For example, an instruction that accesses memory and triggers a page fault exception cannot complete until the page fault is handled.

Interrupts and Exception Analysis To interpose on handlers, DBI frameworks install their own handlers that invoke the dispatcher on the original one (unless the interrupt or exception was caused by the DBI framework).

Because handlers can inspect machine state at the moment before their invocation, DynamoRIO, Pin, PinOS, and Valgrind take care to present plausible machine state – i.e., machine state that could have been observed during native execution. This precaution is essential for correct execution. For example, Linux's page fault handler intentionally panics if a page fault exception arises in the kernel unless the triggering instruction has a pre-allowed address [3]. Under the control of a DBI framework, the pre-allowed instruction never executes – only its copy in the code cache. Hence, to avoid causing a kernel panic, the interposing handler has to translate the code cache address of the faulting instruction to the source instruction's address before invoking the kernel's page fault handler.

Presenting plausible machine state is tricky, particularly if an interrupt or exception arises during the execution of instrumentation code or the dispatcher. In this case, the interrupted instruction has no corresponding source code, thus translation to the source code address is impossible. To circumvent this problem, DBI frameworks queue interrupts and deliver them at the next code cache exit.

Exceptions are handled differently since they cannot be delayed. Exceptions triggered by instrumentation code are handled by restoring native machine state – analogous to how x86 hardware delivers precise exceptions [10] – and invoking the instrumented copy of the original handler. DynamoRIO requires framework users to implement callback functions that translate machine state when exceptions arise during instrumentation code. These callback functions restore registers used by the instrumentation code and provide a source code address for the faulting instruction. PinOS provides instrumentation APIs that automatically perform the required rollback.

For the most part, DBI frameworks avoid triggering exceptions in their dispatchers. However, some exceptions are unavoidable during interpretation. If, for example, an instruction in a basic block being interpreted resides on a non-executable page, then a page fault will be triggered. DBI frameworks handle this situation by executing the instructions preceding the faulting instruction. Once the faulting instruction is reached, the instrumented copy of the original handler is executed.

Interrupts and Exception Design We intercept interrupts by replacing the operating system's interrupt handlers (see Section 3.1). If the interrupted machine state was running in user mode, then we dispatch the kernel's interrupt handler immediately (in this case, the interrupted state is native). Otherwise, our framework atomically adds intercepted interrupts to a CPU-private queue by disabling interrupts. Once queued, we enable interrupts and return execution to the interrupted instruction. When execution next exits the code cache, instead of dispatching the next target, the dispatcher delivers queued interrupts by dispatching on the kernel's interrupt handlers. Once the dispatcher empties the queue, it dispatches the next target.

In the code cache, we replace `iret` instructions with exits to the dispatcher. Following a code cache exit caused by an `iret`, if the interrupted machine state was running in user mode, the dispatcher natively executes `iret`, otherwise it continues delivering queued interrupts.

When dispatching on the kernel's interrupt handlers, we take care to emulate machine state that would have been exposed to the interrupt handler during native execution. Interrupt handlers can expect native machine state and an interrupt stack [2]. However, translating interrupted machine state only requires translating the code cache address; we do not need to translate other registers because interrupts are only delivered at code cache exits (i.e., not in the middle of instrumentation). Once the machine state is translated, we create the interrupt stack.

Our framework handles exceptions triggered during code cache execution immediately. Because machine state might be modified by instrumentation code, we must translate all machine state. Similar to DynamoRIO, we require the instrumentation code to implement a translation callback function: given the triggering instruction, the callback should restore any native state it changed up to that point.

Like other DBI frameworks, we only expect exceptions in the dispatcher while interpreting code. In this case, the dispatcher admits a copy of the code up to the exceptional instruction into the code cache and executes that copy before invoking the instrumented exception handler.

To prevent page fault exceptions while fetching dispatcher and code cache instructions, the dispatcher and code cache are stored in page frames that are always present in all processes' page tables. We assume that the operating system makes such page frames available, an assumption which holds in Linux [3] and Windows [16].

3.3 Concurrency

To handle multithreaded code correctly, DBI frameworks must ensure that the dispatcher and execution in the code cache behave in a thread-safe manner. Concurrency problems arise in the dispatcher when it is used by multiple threads. For example, updates to shared data structures, such as the map of source-code addresses to code-cache addresses, cannot conflict. Concurrency problems arise during code cache execution because the code in the cache contains, in addition to instrumented source code, accesses to data structures shared with the dispatcher.

Concurrency Analysis Valgrind simply serializes all thread execution. In the other user-space frameworks, two approaches are taken to managing concurrency: thread-private data and locking. In the former approach, each thread has its own private code cache and auxiliary data structures. In the locking approach, data is shared between threads and locks are employed to ensure correct access. Thread-private data avoids locking overheads whereas locking avoids redundant interpretation and copying. In practice, the thread-private data approach is only suitable for programs with a small number of threads or programs in which threads access disparate code, such as desktop applications [5].

Kernel DBI frameworks have another concurrency management technique at their disposal: CPU-private data (without control over preemption points, user-space frameworks cannot reliably use CPU-private data). Each CPU has its own private code cache and auxiliary data structures. This approach avoids locking while bounding redundant interpretation and copying. JIFL, which uses CPU-private data, does not work with preemptible variants of the Linux kernel, however.

Concurrency Design Because many threads run inside the kernel (i.e., all user and kernel threads), using thread-private data would have high memory and interpretation overheads. On the other hand, good performance for a lock-based approach is contingent on a sophisticated locking strategy [5]. Because of reentrancy problems, the kernel's locking facilities cannot be used, so all of the required locks would have to be implemented from scratch. Instead, we have elected to use CPU-private data. Each CPU has a private dispatcher and private code cache.

Unlike JIFL, our DBI framework works with preemptible kernels. The CPU-private data approach is sensitive to preemption if any CPU-private machine state is saved when the preempted task context switches. In this case, if the preempted task later resumes on another CPU, then the task will use the wrong CPU's private data. Two problems cause CPU-private state to be saved: exposing a register temporarily holding CPU-private data to the interrupt handler or storing a code cache address on the stack. Our design avoids the first problem by restoring native machine state before delivering interrupts and exceptions. We avoid the second problem by always storing source code addresses, and not code cache addresses, on the stack: we emulate `call` and `ret` by pushing and popping source addresses and calling the dispatcher; as Section 3.2 explains, we ensure that the interrupt stack contains source addresses. JIFL's incompatibility with preemptible kernels arises from not interposing on interrupt handlers: non-native register values may be exposed and code cache addresses are saved on the stack.

3.4 Reentrancy

The problem of reentrancy arises for a DBI framework when its own interpretation code uses library code that it is in the middle of interpreting, for example, heap allocation or I/O.

Reentrancy Analysis DBI frameworks avoid this problem by only using code that they do not interpret, i.e., their own code and the systems beneath them. Because the user-space frameworks do not interpret kernel code, they can safely make system calls. The VMM-based approaches can make use of the VMM itself and virtual machines other than the one being instrumented (e.g., PinOS performs all I/O through Xen's Domain 0 guest). Because JIFL does not sit atop a VMM, it must be entirely self sufficient; JIFL only relies on resources that it allocates when it initializes (i.e., before interpretation begins). To perform I/O, JIFL must first suspend instrumentation.

Reentrancy Design Our design avoids problems arising from non-reentrant functions by not calling any instrumented code from the dispatcher. To accomplish this, the dispatcher is entirely self contained. Specifically, the dispatcher has its own memory allocator. To facilitate I/O, we designed our framework to attach and detach from a running system. To attach, we load a kernel module that notifies each CPU to run an initialization routine that creates the CPU's private data structures, shadows the descriptor tables, and calls dispatch on the initialization routine's next instruction. To detach, on each CPU, we replace shadowed registers with their native contents and return execution to the next source code instruction.

3.5 Code Cache Consistency

To faithfully emulate the native execution of the code being instrumented, DBI frameworks must ensure that the code cache is kept consistent with source code. If some source code is modified after it has been copied into the cache, then the cached code is no longer valid. Likewise, if some source code becomes non-executable, then any cached blocks derived from it are no longer valid. Reliably detecting when source code and permissions change is a complex matter. Consequently, the aforementioned frameworks maintain code cache consistency to various degrees (DynamoRIO, Valgrind, PinOS) or have no de-

tails published (JIFL, Pin, vx32).

Code Cache Consistency Analysis DBI frameworks detect source code changes by tracking changes to virtual address mappings and write-protecting executable pages. When a basic block is copied into the cache, its source's page is recorded and marked read-only. When the source page is unmapped or modified, the block is evicted from the code cache. Unmapping of pages is detected by interposing on system calls in user-space frameworks and as a part of shadow page table maintenance in VMM frameworks.

Code Cache Consistency Design Our framework detects changes to source code and source code permissions using per-CPU shadow page tables. The shadow page table contains the same virtual-to-physical address mappings as the native page table but with more restrictive permissions. When the interpreter copies a block into the code cache, it marks the corresponding page as non-writable in each CPU's shadow page table. We ensure that each CPU's `cr3` register points to its shadow page table instead of the native one. We shadow the `cr3` register like the descriptor table registers.

We maintain shadow page tables by emulating the operation of a translation lookaside buffer (TLB) in software [10]. Initially, and whenever `cr3` is written to, we clear the CPU's shadow page table (analogous to a TLB flush). Similarly, we emulate the `invlpg` instruction in the code cache by removing the corresponding mapping from the shadow page table and evicting code in the cache whose source has been made non-writable. If a page fault occurs because of a write to a page containing source code, then we notify all CPUs, using an interrupt, to invalidate the corresponding code in their caches. We then re-execute the write. Otherwise, if a page fault occurs on an access that the native page table would have permitted, then we copy the native mapping into the shadow page table and re-execute the faulting instruction (analogous to a TLB miss). We handle all other page faults by dispatching on the kernel's handler.

4 Concluding Remarks

Although we are presently targeting Linux, our design is compatible with Windows as well. Our assumptions of exclusive long mode execution and available non-faulting page frames are valid in both Linux and Windows.

We have presented the design for a new DBI framework capable of instrumenting all operating system code. This DBI framework can be used to implement in-kernel dependability-enhancing techniques already deployed in user space. Currently, we are implementing our proposed framework.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, pages 2–13, San Jose, CA, 2006. ACM.

[2] Advanced Micro Devices. *AMD64 Architecture Programmers Manual Volume 2: System Programming*. 3.15 edition, 2009.

[3] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, 3rd edition, 2005.

[4] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.d. thesis, Massachusetts Institute of Technology, 2004.

[5] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *CGO*, New York, NY, 2006. ACM.

[6] P. P. Bungale and C. K. Luk. PinOS: A programmable framework for whole-system dynamic instrumentation. In *VEE*, pages 137–147, San Diego, CA, 2007. ACM.

[7] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, pages 45–58, Big Sky, MT, 2009. ACM.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, Banff, Canada, 2001. ACM.

[9] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX ATC*, Boston, MA, 2008. USENIX Association.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*. 034 edition.

[11] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security*, pages 191–206, San Francisco, CA, 2002. USENIX Association.

[12] 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*, pages 190–200, Chicago, IL, 2005. ACM.

[13] Microsoft. Checklist for 64-bit Microsoft Windows Drivers., 2008. `http://www.microsoft.com/whdc/driver/kernel/64bit_chklist.mspx`.

[14] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, San Diego, CA, 2007. ACM.

[15] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. JIT Instrumentation: a Novel Approach to Dynamically Instrument Operating Systems. In *EUROSYS*, pages 3–16, Lisbon, Portugal, 2007. ACM.

[16] P. Orwick and G. Smith. *Developing Drivers with the Windows Driver Foundation*. Microsoft Press,

Redmond, WA, 2007.

[17] A. Pohle, B. Döbel, M. Roitzsch, and H. Härtig. Capability Wrangling Made Easy: Debugging on a Microkernel with Valgrind. In *VEE*, pages 3–12, Pittsburgh, PA, 2010. ACM.

[18] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor : A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, pages 335–350, Stevenson, WA, 2007. ACM.

[19] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX ATC*, pages 17–30, Anaheim, CA, 2005. USENIX Association.

[20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Gyung Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ISS. Keynote invited paper*, Hyderabad, India, 2008.