Composable Multi-Level Debugging with Stackdb

David Johnson Mike Hibler Eric Eide

University of Utah
Salt Lake City, UT USA
{johnsond, mike, eeide}@cs.utah.edu

Abstract

Virtual machine introspection (VMI) allows users to debug software that executes within a virtual machine. To support rich, whole-system analyses, a VMI tool must inspect and control systems at multiple levels of the software stack. Traditional debuggers enable inspection and control, but they limit users to treating a whole system as just one kind of target: e.g., just a kernel, or just a process, but not both.

We created Stackdb, a debugging library with VMI support that allows one to monitor and control a whole system through multiple, coordinated targets. A target corresponds to a particular level of the system's software stack; multiple targets allow a user to observe a VM guest at several levels of abstraction simultaneously. For example, with Stackdb, one can observe a PHP script running in a Linux process in a Xen VM via three coordinated targets at the language, process, and kernel levels. Within Stackdb, higher-level targets are components that utilize lower-level targets; a key contribution of Stackdb is its API that supports multi-level and flexible "stacks" of targets. This paper describes the challenges we faced in creating Stackdb, presents the solutions we devised, and evaluates Stackdb through its application to a security-focused, whole-system case study.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids; D.3.4 [*Programming Languages*]: Processors—debuggers

Keywords virtualization; virtual machine introspection

1. Introduction

Many virtual machine introspection (VMI) techniques have been developed over the past ten years to analyze, inspect, and reason about the execution of software inside a virtual machine from the outside [3, 5, 6, 11, 17, 22]. VMI-based tools often act like debuggers, using metadata such as debug symbols to interpret data structures and set execution breakpoints. VMI can be a powerful technique to analyze a VM's execution while minimally impacting its internal state. VMI requires no debugging-oriented source-level patches to the software within the VM, and it potentially lowers the odds of detection by the system under inspection.

VMI is often used for whole-system analyses because it can expose the full state of a VM. However, whole-system analyses can be difficult to implement because they involve software components that operate at multiple levels of abstraction over the full software stack: kernel, processes, libraries, and language runtimes. For tasks that involve detailed knowledge of a system's state and structure, such as the analysis of security exploits, a VMI-based tool must overcome the well-known "semantic gap" [2] between the state of a VM and its meaning. For a multi-level analysis, this gap must be crossed at many levels of the software stack.

Crossing the gap is the task of a debugger. Traditional debuggers allow a user to inspect and control one kind of *target* at a time—for instance, GDB [7] supports debugging processes, and KGDB supports debugging kernels [19]. It is uncommon to find a debugger that allows a user to "attach to" a single software system and then debug it at multiple levels of the software stack. For example, if a programmer is using a whole-VM debugger, he or she cannot direct that debugger to also attach to a particular process that is running within the VM under inspection.[1] Moreover, because software stacks are varied, there is a need for a general approach to implementing debuggers that can manage multiple, nested abstractions of a single system. Existing multi-layer debuggers such as Blink [12] and DroidScope [21] do not define general mechanisms for building "stacked" views of a single system.

We developed Stackdb, a debugging library with VMI support that allows users to inspect and analyze software systems at multiple levels of a software stack. In Stackdb, the system being debugged is accessed through one or more

The definitive version was published in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Salt Lake City, UT, Mar. 2014, http://dx.doi.org/10.1145/2576195.2576212

[1] A programmer might use debugger facilities, such as GDB command files, to script deep-inspection tasks such as interpreting and walking a kernel's process list, and thereby construct a view of a particular process. However, this approach is limited: the process is still not a target with its own address space, symbols, threads, breakpoints, and other context resources.

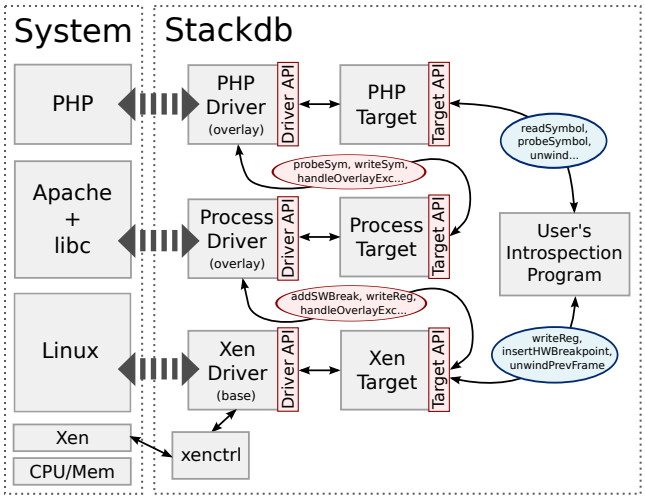

Figure 1. Stackdb applied to a Xen VM running Apache and PHP. In this configuration, three targets provide access to different parts of the system, as shown by wide, dashed arrows. Thin arrows show the actual communication paths.

target objects, as illustrated in Figure 1. A target corresponds to a particular level of abstraction or a portion of the whole system being debugged. Each provides the features of a complete debugger. By invoking *target API* functions, which are common to all targets, a user can install breakpoints, examine software state and symbols, single-step, and potentially modify execution at the level of a particular target.

Figure 1 also shows that each target is paired with a *driver*, whose purpose is to implement debugger-like inspection and control features for a particular software abstraction: e.g., kernel, process, or language runtime. Although all drivers implement a common *driver API*, we distinguish two primary classes of implementation. A *base driver* interacts directly with the system being debugged, e.g., via a hypervisor-provided interface or ptrace(2). An *overlay driver* interacts with the system through another target, i.e., by "stacking" on top of an appropriate underlying target. The overlay driver communicates with the underlying target through the target API.

Because the target API is implemented by every target, a user can easily instantiate multi-level stacks of targets. In addition, the ability to implement drivers in terms of underlying targets greatly eases the process of developing new drivers, e.g., for new language runtimes. Finally, the target API makes it possible to implement generic analyses and utilities that can be applied to multiple levels of a software stack. We believe the "stackability" offered by Stackdb advances the state of the art for debuggers and that it can enable more powerful and detailed VMI-based analyses.

We have implemented four Stackdb drivers: a *Xen driver* (base) for debugging Linux-based guest OSes; a *process driver* (overlay) for debugging user-space processes within Linux-based guests; a *Ptrace driver* (base) for debugging

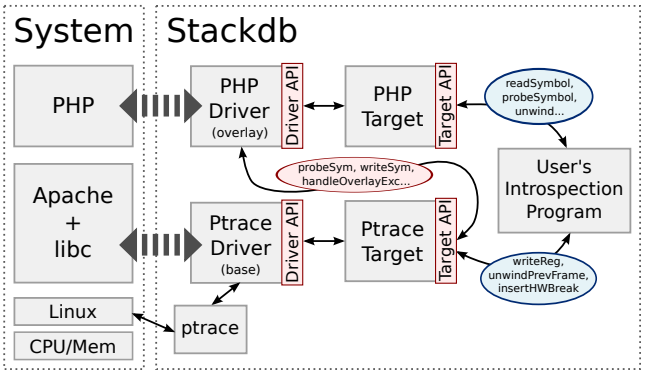

Figure 2. Stackdb applied to a local Apache process and PHP. In this configuration, the PHP driver runs atop a Ptrace target.

local processes; and a *PHP driver* (overlay) for debugging scripts at the PHP level.[2] We chose this set for its practical utility and to exercise Stackdb's APIs. Notably, the PHP driver can sit atop a process target (i.e., a target that uses the process driver) *or* a Ptrace target. The former configuration is for debugging scripts running in a VM; the process driver is an overlay that runs atop a Xen target, so debugging a PHP script within a VM involves two overlays in total (Figure 1). The Ptrace configuration is for debugging PHP scripts in a local web server, as illustrated in Figure 2.

This paper presents Stackdb and describes its application to tracing an example security exploit across software layers within a VM. Our first contribution is Stackdb, a debugger library that supports the development of powerful, programmatic, and whole-system VMI analyses. Our second contribution is Stackdb's design: the architecture that allows overlay drivers to be implemented atop other targets, and Stackdb's solutions to the challenges of implementing stackable targets.

2. Challenges

We encountered several challenges not faced by standard debuggers while building Stackdb because it provides access to a whole system through multiple, coordinated targets.

2.1 Attaching to and Controlling the System

When a Stackdb base driver attaches to a system (e.g., a VM or local process), it does so through an existing API such as ptrace(2). The UNIX ptrace system call allows a debugger to control another process at a fine-grained level, e.g., pausing and resuming threads at will. Kernel-level debuggers such as KGDB [19] rely on internal kernel support or patches that provide a remote debugging protocol.

The existing implementations of these APIs, however, are not available to Stackdb *overlay* drivers. For instance, consider Stackdb's process driver, which provides access

[2] We often refer to targets according to the properties of their drivers. For example, a *PHP target* is one that uses the PHP driver. A *base target* uses a base driver, and an *overlay target* uses an overlay driver. Figures 1 and 2 show examples of this convention.

to Linux processes running in Xen VMs (Figure 1). The process driver does *not* use the ptrace facility provided by the VM's guest OS, for several reasons. First, we do not want to prevent the guest OS from using its implementation of ptrace to provide services within the VM; second, we do not want Stackdb to be observed by the guest OS; and third, in newer Linux kernels, using ptrace would require Stackdb to use kmalloc to obtain memory for required data structures, which would alter the state of the guest OS. These arguments apply to the implementation of Stackdb's drivers in general.

To avoid perturbing the system under inspection, an overlay driver cannot depend on debugging APIs that are implemented within that system.[3] Moreover, a developer may want to implement an overlay driver for a part of the software stack that does not implement any internal debugging API. Thus, a key challenge for Stackdb is to support **attachment to and control of parts of a system** without use of that system's internal debugging features. This challenge is addressed by Stackdb's target API and its ability to implement drivers atop the target API (i.e., stacking). A base driver attaches to a system not through that system's internal APIs, but through an API beneath that system: e.g., through a hypervisor-provided interface for VMs. Likewise, an overlay driver does not attach to a system directly via one of that system's internal APIs. Instead, it invokes the target API of an underlying Stackdb target. The underlying target provides a rich debugging interface to the system being controlled.

2.2 Staying in Control

Once attached, a debugger must stay in control. This requires observing and managing the execution of the threads within the system being debugged. In Stackdb, this can be tricky, because an overlay driver may not have control over the scheduling of the threads that it manages. This is a consequence of Stackdb's goal of making it possible to implement overlay drivers without perturbing the system being inspected.

To make this clearer, consider the situations faced by a traditional ptrace-based debugger (e.g., GDB) and Stackdb's process overlay driver when debugging a multithreaded process. When a process thread hits a breakpoint under GDB, the kernel pauses all the other threads in the process. The kernel changes the states of those threads, allowing GDB to handle the breakpoint exception atomically with respect to the process' execution. In contrast, when a thread hits a breakpoint under Stackdb's process driver, Stackdb does not require that the underlying OS suspend the other threads within the process being observed. The process driver must handle the exception without thread-scheduling assistance from the OS within the VM being inspected.

The challenge faced by an overlay driver is **retaining execution control without thread-scheduling assistance.** For example, consider what happens when a process thread hits a breakpoint: typically, the breakpoint is removed, the thread is single-stepped, and the breakpoint is restored. Without scheduling assistance, it is possible that some other process thread will traverse the breakpoint location while it is temporarily removed—a loss of debugger control. Other problems arise due to thread context switches, thread privilege-level changes, and exceptions that occur while another exception is being handled. These issues are addressed by Stackdb's target implementation and its drivers. In contrast to drivers, which have multiple implementations for different software layers, there is a single implementation of the target API in Stackdb. In OOP terms, all targets are instances of a single class. The target API implementation provides general functions that allow targets' "client" overlay drivers to retain execution control, e.g., by signaling underlying scheduling events "up the stack."

2.3 Minimizing Overhead

All debuggers add overhead, and because Stackdb provides stacked debugger targets, it is important that Stackdb **minimize the performance overhead at each level of the stack.** Every driver must implement Stackdb's driver API for the level of the software stack it addresses. To do this, a driver must find its subject within the system being debugged—e.g., locate a particular process inside a whole VM—and provide debugger-like inspection and control operations for that subject. These parsing and control tasks are potentially expensive. In addition, overlay drivers should be informed of thread and address-space state changes so that they can maintain accurate models of their subjects.

Suppose a developer wants to examine a PHP process spawned by a web server running atop Linux within a Xen VM. To do this, the developer uses Stackdb to create a stack of three targets (Figure 1). If the developer places a breakpoint on a PHP function, the resulting debug exception will cause the entire VM to pause. When the exception occurs, each driver in the stack may need to examine its subject to detect state changes (e.g., new threads, exited threads, and address-space changes) so that it can appropriately handle the exception. A base driver queries the system directly; an overlay driver queries its underlying target.

To minimize overhead, Stackdb provides features to track state changes quickly. These include (1) thread and address-space change notifications and (2) caching of parts of a system's state. These reduce the cost of providing debugging interfaces at every level of the target stack.

2.4 Easing Implementation

To be flexible for whole-system debugging, Stackdb must **simplify the task of adding new overlay drivers.** Beyond the benefits of target stacking, Stackdb eases driver implementation by providing access to an underlying target's debug symbols: variables, functions, types, and data locations. By referring to symbols within an underlying target, an overlay

[3] One can imagine special overlay drivers that *do* use a system's internal debugging APIs, when circumstances allow. Our goal, however, was to design Stackdb so as not to depend on a system's internal debugging features.

driver can potentially work across many different versions and compilations of the underlying target's software.

Stackdb's process driver, for example, must parse a Linux process control block (PCB) and memory mappings in order to establish control over a process. By accessing the Linux kernel through symbols, rather than hard-coded addresses and offsets, the process driver works atop Linux kernel versions from 2.6.18 to 3.8.0.

To support targets for programs written in different languages, Stackdb's symbol-loading and querying API supports different kinds of symbols: bare ELF symbols; C/C++ symbol, type, and location data encoded in DWARF; and dynamic, variably typed symbols for languages like PHP. Stackdb handles dynamically loaded code, as well as changes to the protections and sizes of memory regions, so that it can manage changes to symbol availability and variable accessibility.

2.5 Handling Heterogeneity

Whole software systems are made from layers that implement different abstractions. The challenge for Stackdb is to **support a variety of different execution models**—from a kernel executing in a VM to an interpreted language running in a user-space process—while retaining a single target API that enables composition through stacking. We addressed this challenge by designing the target API according to a general model of machine-based execution and debugging: a model that includes threads, memory, address spaces, registers, symbols, and breakpoints. The implementation of targets is generic, with layer-specific details modularized within drivers. Not all of Stackdb's target API will apply to all layers, but its uniformity is important for stacking and implementing analyses that can be applied to multiple levels of a system.

Group	Target API	Driver API
	User-invoked	
build	create	*same*
	open	init, attach
	close	detach, kill
model	open	load{Spaces, Regions,Debugfiles}
control	pause, resume	*same; opt. for overlays*
	monitor, poll	monitor, poll, handleExc, handleInterruptedStep; *all opt. for overlays*
	stepStart, stepEnd	*same; opt. for overlays*
overlay	lookupOverlayThread	*same*
	createOverlay	*same*
	User- and Driver-invoked	
sym	lookup{Sym,Addr,Line}, loadVal	readSym; *optional*
value	store, refresh, convert	writeSym; *optional*
cpu	readReg, writeReg	*same; optional*
mem	read, write	*same; optional*
	v2p,{read,write}Phys	*same; optional*
thread	{load,pause,flush}Thread	*same*
probe	probe{SymName,Addr, Line}	{add,del}SWBreak, {add,del}HWBreak
	probeSym	*same; optional*
unwind	unwindStack, prevFrame	*same; optional*
maint	setActiveProbing	*same; optional*
	Driver-invoked	
overlay	notifyOverlay	handleOverlayExc

Table 1. Summary of Stackdb's target and driver APIs

3. Stackdb Architecture

Stackdb helps users to write programs that analyze live, whole-system executions. Users write programs that attach to *targets*, where each target provides access to and control over some part of the whole system being debugged. Users' programs then use the *target API* to pause, single-step, and resume targets' execution; query symbol data; modify targets' memory and CPU state; and insert *probes* (breakpoints and watchpoints) that notify user-specified handlers when traversed by a thread of execution. User programs can also load and modify symbol values based on source-language datatype information; unwind thread stacks; disassemble code; and extend the base probe libraries with new kinds of probes.

Figure 1 provides an overview of Stackdb's core abstractions and how they can be combined to debug whole systems. The target API provides a uniform interface to all targets. Internally, each target utilizes a *driver*, which implements control and inspection functions for a particular part of the software stack. A driver attaches to its part of the system being debugged, constructs a model of that part (with address spaces, memory regions, threads, and debug symbols), receives and handles debug exceptions, and performs the details of reading and manipulating state and execution. A target communicates with its driver through the *driver API*, which is implemented by all drivers. (However, not all drivers implement every part of the API.) A *base driver* communicates directly with the system being debugged (e.g., via VMI), and an *overlay driver* communicates via an underlying target.

Table 1 summarizes the two APIs and illustrates how target API functions map to driver API functions where applicable. (Many utility and helper target API functions are omitted for brevity; Table 1 lists only the functions that are most important to Stackdb's design.) The table is organized into three parts, corresponding to parts of the target API that are invoked by different clients. The first contains functions that are invoked by user-written analyses and debugging programs. The second lists functions that are called by user analyses and by drivers, and the third lists functions called only by drivers. Although the mapping seems one-to-one for many operations, in fact, the implementations of the target API functions do much driver-independent processing (Section 2.2). Where the mapping is not one-to-one, the table shows the driver API functions called by each target API operation. Optional driver functions are also noted.

3.1 Targets

A target is the primary object with which a user interacts. It corresponds to an executing program: a kernel, a process, or a higher-level execution context such as a script. A target may be created by spawning a new program or by attaching to an existing program, depending on the driver.

Stackdb's target model supports multiple threads that share a single address space. An address space is divided into regions, which are further subdivided into ranges. Typically, a range models a contiguous chunk of memory with uniform protection bits. A region models a related group of ranges (e.g., from a single shared library or executable). For instance, ELF binaries are typically loaded into several ranges: one for program text, another for read-only data, and another for writable data. It is convenient to group these related ranges into a region, since debugfiles usually cover an entire region. (A Stackdb driver will attempt to load a debugfile for each region that contains symbols loaded from a binary program or library.) Helper functions translate between object-file virtual addresses and program-image (linked) virtual addresses.

Stackdb's targets keep state that is sometimes needed for controlling the execution of multithreaded software. Remember that a debugger that supports software breakpoints in multithreaded programs must handle debug exceptions whenever any thread traverses a breakpoint (Section 2.2). When a debugging API (e.g., Ptrace) allows for thread-control operations, the debugger can atomically single-step one thread while all others remain paused. However, Stackdb is designed to allow debugging even when underlying thread-scheduling assistance is not available. For systems that do not support individual thread control, a Stackdb target must keep state associated with the breakpoint being handled. Moreover, since Stackdb supports unlimited single-stepping at a breakpoint, a target must store a stack of states so that it can encounter a new breakpoint while stepping on behalf of a previously hit breakpoint.

CPU state is accessed on a per-thread basis. Memory values are loaded relative to a thread, because some debug-symbol formats encode locations relative to the values of CPU registers.

3.2 Drivers

To attach to a portion of the whole system being debugged, a target uses a driver and invokes its functions through the driver API. A driver corresponds to a particular software layer or abstraction, and some driver API functions might not apply to all drivers: for example, it does not make sense for a PHP script-level driver to provide access to CPU registers. A useful driver, however, should implement as many of the optional API functions as it can.

To attach the driver to its "subject"—i.e., its portion of the whole system being debugged—a Stackdb target invokes functions in the *build* group; it later uses these to detach from and/or terminate the subject. Once attached, a target calls the *model* functions to create a representation of the subject: cataloging its address spaces, regions, ranges, and loading debugfiles for those regions. At this point, a driver can start optimizing its internal maintenance of the model. For example, it can cache symbol values for later use, and it can install probes (Section 3.6) in order to track state changes.

The driver *control* functions allow targets to control the execution of the system being debugged. The system can be interrupted and resumed, and the driver API provides various event-loop mechanisms (blocking, polling, or a combination) that allow one to wait for events. The functions in the *cpu*, *mem*, and *thread* groups allow targets to read and write memory and CPU state. The *readSym* function can be provided by drivers that do not provide raw memory or CPU access, but do provide symbols and values, like Stackdb's PHP driver.

Although Stackdb manages high-level probe creation and state (Section 3.6), drivers provide the low-level implementations of software breakpoints, hardware debug-register-based breakpoints, and watchpoints. Furthermore, for software abstractions that do not expose numeric memory addresses, a driver may implement the *probeSym* function to place a probe on a symbol instead of an address.

Drivers can provide their own exception handlers for breakpoint and single-step events, although the default handlers are very powerful. (The default handlers support probe actions, discussed in Section 3.7.) The *handleInterruptedStep* function allows a driver to handle cases where the driver's subject steps from one thread into another, or into another thread context (i.e., from user to kernel space). This can arise when the driver does not provide individual thread control. In this case, the single-step is effectively paused until the system returns to the thread or context the step was triggered in; then the probe library can continue to handle the stepping thread.

Stackdb also allows drivers to provide their own low-level *unwind* functions to implement the target API's generic unwinder. Stackdb provides a default x86-based unwinder.

The *setActiveProbing* function allows users to control whether or not drivers can actively maintain their internal state. It can be important for a driver to monitor thread events and memory-region changes in its subject; the latter is especially important for handling dynamically loaded code. Rather than rescanning its subject's data structures at each debug exception, or selectively by heuristic, it may be more efficient for a driver to use probes to detect these changes. Stackdb refers to this as "active probing," and a user can enable it when the overhead of active probing is less than the overhead of periodic data-structure scans.

The *overlay* function group is described in Section 4.2.

3.3 Target Personalities

Each target can provide a *personality*, which is implemented by its driver. A personality defines an abstract interface for accessing features common to different types of software that one might want to debug with Stackdb. We envision three

common personalities: an *OS personality* that wraps common OS objects, e.g., processes, threads, address spaces, files, sockets, system calls, loadable modules, and users; a *process personality* that abstracts common process-level objects, e.g., command lines, threads, and memory mappings; and an *application personality* that exposes idioms for program runtimes. Stackdb currently defines an OS personality API that provides kernel-version information and a system-call abstraction; the latter is useful to placing probes on the implementations of system calls. Although we are still developing the personality interfaces, we envision that they will simplify driver development and make it easier to stack targets.

3.4 Debugfiles and Symbols

When a driver attaches to its subject, it analyzes the subject and loads as many sources of debug symbols as possible. Drivers can load symbols from the binary files that compose the subject's software, and they can search for more detailed sources of debug symbols (e.g., DWARF debuginfo files) that correspond to those binaries. A target receives the symbol data collected by its driver and makes that data available through the target API. Both user-written programs and overlay drivers use the target API, so both can easily look up symbols, addresses, and source code lines.

Stackdb's *dwdebug* library loads ELF symbols and DWARF debuginfo from binary files and constructs fast search indices. Its core data structures flexibly describe scoped hierarchies of functions, variables, data types, namespaces, and aggregate types like C structs and C++ classes. The dwdebug library also manages location information (telling the target API how to load and locate functions and variables), call-frame information (allowing targets to unwind stacks), source-line information (linking symbols to source code), and address information (linking symbols to the binary compilation). Its core abstractions are general enough to support symbols from radically different languages. We have used it to implement excellent C support, good C++ coverage, and partial PHP support.

3.5 Values

Through the target API, it is possible to read and write the memory of the system being debugged. When possible, however, it is better to load data into *values* rather than employing raw memory access. The functions in the *value* group of Table 1 allow a user to read or write typed values from symbolic locations or raw addresses in the system being debugged. Users can load and display decoded basic types, and they can load members from data structures such as C structs and C++ classes. The target API provides numeric type wrappers, freeing users from size and encoding concerns.

3.6 Probes

Stackdb provides powerful, abstract, per-thread breakpoint and watchpoint support via *probes*. Normally, users register a probe atop a symbol; the details of the probe's manifestation are hidden. Probes support a normal breakpoint interaction pattern. The user supplies a *pre-handler* that is executed before the breaking instruction is executed, and also a *post-handler* that is executed after the breaking instruction has successfully executed. Users may also schedule *actions* (Section 3.7), such as single-steps, that occur between the handlers.

Basic probes may be placed on addresses, source lines, or symbols (that are resolvable to addresses). A *probepoint* represents the implementation of a breakpoint or watchpoint in the system being debugged: e.g., modifications to a CPU's hardware debug registers or the installation of a soft breakpoint instruction inside the program text. Multiple probes can be registered atop probepoints, supporting cases in which different probes care about the same event. When a probepoint's implementation is triggered, the probe's pre-handler is fired. After the instruction at the probepoint is executed, the post-handler is executed. Watchpoints are similar.

Probes are hierarchical: high-level *metaprobes* can register atop basic probes, or other metaprobes, to receive the events that trigger their pre- and post-handlers. This hierarchy can be used to build complex, stateful metaprobes that are composed of many basic probes. This flexible probing infrastructure is key to implementing overlay drivers.

Stackdb's probe library provides several important metaprobes. A *function entry/exit metaprobe* fires its pre-handler when the entry point of a function is reached; it fires its post-handler when any of the exit points of a function are reached. An *inlined-symbol metaprobe* registers atop basic probes placed on all inlined instances of a particular symbol. A *function-instruction metaprobe* allows a user to place basic probes on all instances of chosen x86 instructions within a function; the metaprobe's handlers fire when any of the selected instructions are hit. A *function-invocation metaprobe* allow users to catch invocations of a user-specified function that occur within another function.

A *symbol-value metaprobe* allows a user to register a probe on a function or variable symbol, and additionally place regular-expression filters over named values associated with that probe. When the filters match the values, the metaprobe's pre- or post-handlers are fired. If the probed symbol names a function, the associated values are the function's arguments (and, when the post-handler is fired, its return value). The values are automatically obtained and string-ified for easy comparison. If the probed symbol names a variable, the pre-handler is fired if the (string-ified) previous value of the variable matches; the post-handler is fired if the new value of variable matches. Symbol-value metaprobes are particularly powerful because they maintain state. A symbol-value metaprobe on a variable requires the metaprobe to recall the variable's previous value. A symbol-value metaprobe on a function must keep a per-thread stack of invocations of itself, since the post-handler can only be evaluated for firing when the function returns.

3.7 Probe Actions

Probe actions help users script actions to be taken when probes are hit. Actions are executed between the firing of the probe's pre- and post-handlers. Users may schedule one-shot or recurring actions for probes in advance or from within a handler. Actions allow users to single-step a target at a probepoint and/or modify CPU registers and memory.

Stackdb provides a particularly powerful action called "abort." This action does not execute the original instruction at the probepoint, but instead temporarily replaces it with an x86 return instruction, effectively aborting the current function. This is handy for exploring alternate executions or avoiding side effects. For example, it can be used to study malware while suppressing the malware's harmful activity. The abort action attempts to use debug symbol information to determine how much to adjust the stack pointer to clean up the returning stack frame; it can also try to infer this amount via disassembly of the function's prologue.

4. Overlays: Building Stacks of Targets

Stackdb allows a user to debug a whole system at multiple levels of the software stack. It does this by making each level of the stack that the user wishes to debug accessible as an individual target. Stackdb attaches to the lowest-level part of the whole system via a base target. It then allows a user to create overlay targets for each interesting higher level of the system's software stack. A user can interact with an overlay target via the target API, just as he or she would interact with a base target. Each overlay target utilizes an overlay driver, which is "stacked" on top of an underlying target.

Well-designed overlay drivers should be able to sit atop any target that provides the personality they require. For instance, a driver for a higher-level language would sit atop any driver that provides the process personality, for any process that is executing that language's interpreter or runtime. This section discusses the details of implementing overlay drivers for Stackdb.

4.1 Overlay Driver Implementation Strategies

Stackdb supports several strategies for building overlay drivers. A sophisticated overlay driver might receive and process exceptions from its underlying target; act like a base target by subscribing to its own debug-exception stream (Section 5.1); and also forward events to higher-level overlay drivers. A simpler overlay driver might receive and process exceptions only from its underlying target and consume them all, if it is not meaningful to forward them as debug exceptions to a higher-level overlay. The process driver described in Section 5.2 is an example of this simpler strategy. In this case, the higher-level overlay driver stacked atop the lower-level target can insert probes in the lower-level overlay, and fire its own higher-level events when those probes are hit and some set of conditions match.

Overlay drivers can be quite simple: a new one can be built by implementing a small subset of the driver API. For instance, consider developing a new overlay driver supporting a high-level language. This overlay's implementation of the driver API would perform the following steps:

1. Create a single address space containing a single region and range (the *init*, *attach*, *loadSpaces*, and *loadRegions* driver API functions).
2. Create one or more threads corresponding to the threads in the underlying target. Use direct correspondence if the language's threads are 1-to-1 mapped to underlying-target threads, or an M×N mapping if the language uses virtual threads (the {*load*, *pause*, *flush*}*Thread* driver API functions).
3. Associate a higher-level language debugfile with the region, populate it with symbols, and support loading and interpreting the values of symbols (the *loadDebugfiles* and *readSym* driver API functions).
4. If the language does not provide raw memory or CPU state access, disable those parts of the driver API.
5. Place probes on key functions in the language interpreter, which is accessed through the underlying target. Use these probes to implement the *probeSym* driver API function, and disable support for other kinds of breakpoints and watchpoints.
6. When *probeSym* is invoked (because a client wants to place a probe in the program that the interpreter is executing), implement the requested probe as a metaprobe. The metaprobe sits atop the probes that this driver placed in the language interpreter in step 5.
7. Implement single-stepping and stack unwinding if meaningful. Single-stepping might be implemented by stepping statement executions instead of individual instructions.
8. Reuse the underlying target's functionality for other driver API functions, or do not provide implementations of them because they do not apply.

Section 5.3 describes the implementation of the PHP driver, which generally follows the recipe above.

4.2 API Functions for Overlays

Section 3.2 describes most of core functions that drivers should implement. In this section, we focus on the portions of the APIs that are specific to implementing stacks of targets, shown in the *overlay* group in Table 1.

Recall that the target used by an overlay driver is referred to as that driver's "underlying target." If a target T is to be used as an underlying target, then T's driver must implement the *lookupOverlayThread* function. This function locates threads that can be mapped to threads in an overlay target. For instance, consider a driver that examines an OS kernel. A kernel-only thread in an OS would not support the notion of a

thread in an overlay target, but a user thread would, because user threads run programs at higher levels of the system software stack. Thus, the *lookupOverlayThread* function for an OS-level driver would return references to user threads, and not to kernel-only threads.

The driver of an underlying target T must also implement the *createOverlay* function. This is called to help instantiate any overlay driver that will sit on top of T. We involve the underlying target so that it can influence the overlay-creation process. For example, an OS target might help a user create a process overlay target by ensuring the user creates the overlay using the process's thread group leader—a detail that the user might not be aware of, but that matters greatly on Linux.

An overlay driver may need to implement the *handleOverlayExc* function to receive events forwarded from its underlying target. This is necessary when the underlying and overlay targets share an execution model. For example, an x86-based OS and its user processes both execute code on the system CPU, in different privilege levels. In this case, the OS target's driver will be subscribed to the debug exceptions coming from the CPU, and it will receive exceptions that apply to both itself *and* its process overlay targets. The OS target's driver can forward those exceptions that apply to its process overlay targets via the *notifyOverlay* function.

An overlay driver that does not share an execution model with its underlying target does not need to implement *handleOverlayExc*. For instance, the driver for a high-level language can implement its probes by instrumenting key locations within a language interpreter (Section 4.1, steps 5–6). In this case, the triggers for probes at different levels of the software stack are distinct, and do not need to be disambiguated.

4.3 Controlling Threads in Overlay Targets

The semantics of thread control in overlay targets can be confusing. Pausing a single thread in an overlay target generally causes *all* of its threads to pause. Because the overlay cannot use thread-scheduling features that are internal to the system being debugged (Section 2.2), it must instead pause threads through its underlying target. Going down the stack, the base target ultimately pauses the entire system being debugged. Resuming threads in overlay targets is similarly all-or-nothing. For these reasons, we expect that overlay drivers will only rarely implement the *control* driver API functions listed in Table 1. Users can simply monitor, poll, pause, and resume the base target instead. This is not a problem in our experience, since a user can easily pause the base target and then inspect the states of overlay targets.

5. Implementation

Stackdb is written in C and supports the x86 and x86_64 platforms. Its core libraries use the `elfutils` library for reading ELF and DWARF information from binary files, and it uses the `distorm` [4] library for x86 and x86_64 disassembly. Stackdb also provides an SOAP service that exports both a low-level interface for debugging (e.g., RPCs to install breakpoints) and a high-level interface for running analysis programs written using Stackdb.

Component	LOC
Target library *impl. of the target API*	21,625
dwdebug library *handles debuginfo*	23,784
Ptrace base driver	5,167
Xen base driver	10,497
Process overlay driver	1,886
PHP overlay driver	2,949

Table 2. Lines of code in Stackdb components

Stackdb contains more than 100 KLOC and required approximately two person-years of development effort. Table 2 summarizes the lines of code within several of Stackdb's components. The target and dwdebug libraries provide a significant amount of generic target, thread, probe, and symbol-handling functions to both users and drivers. The Ptrace base driver is a relatively straightforward implementation of Stackdb's driver API on top of the standard ptrace(2) facility. The Xen driver is more sophisticated; much of its complexity stems from the careful exception handling necessary to support both paravirtualized and HVM Xen guests running atop Xen 3.3 to 4.3 hypervisors. Because the Xen base driver handles Linux's inherent complexity, the process overlay driver is relatively simple. Its implementation is focused on interpreting the kernel data structures that define a process; in comparison to the Xen driver, the process driver needs much less code to handle debug exceptions. Similarly, the PHP overlay driver focuses on tasks that are particular to PHP, because process-level details are handled by its underlying target. The data in Table 2 suggests that, in comparison to base drivers, overlay drivers can indeed be simple.

Below, we further describe the implementations of three of Stackdb's drivers. We do not describe the Ptrace driver; its use of the ptrace(2) debugging API to attach to a multithreaded UNIX process is very standard.

5.1 Xen Driver

Stackdb's Xen driver supports Xen hypervisor versions from 3.3 to 4.3, running paravirtualized or HVM Linux guest kernels ranging from 2.6.18 to at least 3.8.0. (We have not tested all kernel versions in that range.)

The Xen driver is a base driver that supports the *overlay* functions shown in Table 1; thus, it supports overlay targets. It uses Xen's standard `xenctrl` library to attach to and control VMs, and to read and write CPU registers and state. It receives debug exception notifications for VMs on Xen's virtual debugger IRQ port. It employs `libvmi` [15] (or its predecessor, XenAccess) to handle virtual-to-physical-to-machine memory translation and mapping.

The Xen driver constructs a model of the VM it attaches to by implementing the *model* driver API functions. In this

driver, those functions look up and load key kernel variables and pointers by type. To obtain the list of threads running in the OS, the driver walks the kernel's task list. The driver creates a single address space (since the kernel can address all memory in the VM) and regions corresponding to the kernel program text and to dynamically loaded modules. To obtain loaded-module information, the driver walks the kernel's module list. The driver looks for debugging symbol files corresponding to the kernel and its modules in the dom0 filesystem (where Stackdb runs).

Although the Xen driver is a base driver and ultimately interacts with its VM through `xenctrl`, it also invokes the API functions on its own target in order to reuse the generic features provided by Stackdb's target library. It makes these "self-target invocations" to place probes on key symbols within the VM/kernel being debugged, to look up kernel symbols, and to read kernel data structures in a type-aware manner. Only some driver API functions can use this implementation strategy; in particular, the *build* and *open* function groups should avoid this style of implementation.

The Xen driver uses "self-target invocations" to implement its *setActiveProbing* driver API function. To implement this function, which sets up event notifications for an overlay driver, the Xen driver places probes atop key functions on the kernel's thread-creation and destruction paths, as well as on the module-load and unload paths. This allows a user to configure the Xen driver to actively track new threads and kernel modules instead of repeatedly scanning memory to find new ones (Section 2.4). Without Stackdb's support for debug symbols, the Xen driver could not feasibly support active probing across a wide range of kernels, because the necessary monitoring points change across kernel versions.

A complicating factor for the Xen driver is that it must implement single-stepped execution in two ways: by setting the x86 TF bit in the EFLAGS register for paravirtualized guests, and by setting the Monitor Trap Flag in HVM guests. Xen requires that HVM guests be stepped using the MTF. The MTF is a per-HVM flag that is global to the VM, and unlike like the TF bit in the EFLAGS register, the MTF is not changed at thread context switches. Thus, in an HVM guest, a single-step begun in one thread or context might continue into another thread or context. Single-stepping must therefore be handled carefully to ensure that the Xen driver does not attempt to handle single-steps in user space.

Finally, the Xen driver must support a limited notion of per-thread virtual-to-physical memory translation, since user-space threads have their own virtual address spaces. This is necessary so that the process overlay target can place software breakpoints inside shared libraries. If the breakpoint were placed at a virtual address, and if it were then hit by some process that was not monitored by a process overlay target, the Xen driver would not recognize it as a hit of a valid breakpoint; the Xen driver would assume instead that it was caused by the process itself. The only way for the Xen driver to recognize these events as valid debug exceptions is to place them on physical Xen target addresses. Depending on the user-space thread in which they occur, the Xen driver will invoke the target API's *notifyOverlay* function to allow the overlay target's driver to handle the exception. If, however, the thread that hits the breakpoint has no associated overlay, the Xen driver must *emulate* the breakpoint in that thread. Otherwise, the thread will suffer a fatal exception.

5.2 Process Driver

The process overlay driver allows a user to debug a Linux user-space process running in a Xen VM. A process has the same execution model as its underlying OS, so the process driver does not need to re-implement much of the driver API, especially the *cpu*, *mem*, *probe*, and *thread* group functions. It can implement those by invoking the target API functions of its underlying target. To model a process's address space, the driver reads the process's memory-mapping data structures from kernel memory, finds the names of files that were mapped or loaded into memory, and searches for matching debuginfo. The process driver implements *handleOverlayExc*, allowing it to receive debug exceptions from its underlying target, and also *lookupOverlayThread* and *createOverlay*, which support overlays atop the process driver.

Two aspects of this driver's implementation are notable. First, its implementation of *handleOverlayExc* must handle cases in which a user-space thread is single-stepped into the kernel. It uses the target library's default implementation of the *handleInterruptedStep* function in this case, to pause handling of the single-step (as well as the breakpoint being handled, if any) until the thread returns to user context. Second, the driver must handle software breakpoints specially by implementing a version of *addSWBreak*. Most modern operating systems allow read-only program text pages to be shared among processes. This means that the process driver must place breakpoints at physical memory addresses in its underlying target, not just at virtual addresses in a process. By setting a breakpoint at a physical address, the underlying (Xen) target can recognize debug exceptions that occur in processes that are not attached to a process overlay.

The process driver's implementation currently has two artifacts that violate the clean stacking semantics that Stackdb seeks to provide. First, it must be placed atop a Xen target. We want the process driver to handle processes in Xen HVM guests, and this requires use of the MTF for single-stepping—but Stackdb does not yet abstract the state of the MTF. Second, the process driver handles only Linux processes. We expect that once Stackdb more fully implements OS personalities—to include abstractions of processes and their metadata—the process driver will be able to sit atop any target that provides the OS personality.

5.3 PHP Driver

The PHP overlay driver allows a user to debug a PHP process *at the PHP-script source level*. Thus, it contrasts

with the process driver, which allows the same PHP process to be debugged, but at the C or C++ source level. PHP is an interpreted language with dynamically typed variables, dynamic compilation, and multithread support. Stackdb's PHP driver is a prototype, but still powerful. It supports built-in functions, user-defined functions, and function arguments. It can load values that belong to several PHP datatypes (null, long, double, string), but it does not yet support more complex PHP types (associative arrays and classes). It can install breakpoints on functions, but it does not yet support single-stepped execution. The PHP driver can be stacked atop either the Ptrace driver or the process driver.

Stackdb's other drivers assume a direct and x86-based execution environment, but PHP's engine executes a custom opcode-based intermediate representation. At the source level, it provides no access to raw memory or CPU state. Thus, the PHP driver need not provide the *cpu* and *mem* driver API functions, nor functions to install breakpoints. Instead, it implements the *readSym* and *probeSym* driver API functions so that users can read and install probes on symbols.

The PHP driver uses its underlying target to install probes on important C functions in PHP's execution engine. In particular, these probes monitor functions that handle PHP's opcodes. When the PHP driver is invoked to place a PHP-level probe, that probe is implemented as a metaprobe (Section 3.6) that sits atop the C-level probes. This implementation strategy was previously described in Section 4.1.

When the PHP driver attaches to a process, it unwinds the stack of the underlying target to determine if PHP's execution engine has started executing scripts. If it has, the driver dynamically generates a debugfile containing PHP base types, class types, and both user-defined and internal functions. It does this by applying the target API to its underlying target: loading the C-level data structures that describe PHP types, functions, and values, and converting them into Stackdb datatypes, function types, and values. After this step, a user can then install PHP-level probes onto PHP functions.

The implementation of the PHP driver required about three person-weeks of effort by a skilled developer (the primary author of Stackdb, who has moderate PHP experience). One person-week was spent understanding the PHP engine and developing a driver implementation strategy. These tasks required reading online documentation about how to write PHP extension libraries in C, and also reading the source code for PHP's Zend [16] compilation and execution engine. Implementing the driver to its current level took two person-weeks; a key complication was finding a way to gain access to PHP's thread-local storage to obtain symbol information. Stackdb architecture allowed us to focus our development effort on PHP-specific issues alone. General target issues, and details below PHP's implementation, were handled by other components of Stackdb.

6. Backtracking an Exploit Attempt

We illustrate the usefulness of Stackdb through a case study. Our goal is to showcase the investigatory power of a Stackdb-based analysis that (1) applies to multiple targets, corresponding to different levels of a software stack, and (2) makes use of those targets to analyze cross-layer behavior.

In this case study, we use Stackdb to detect and trace back a privilege-escalation exploit enabled by a buggy PHP script. To set up the scenario, we run a Xen VM containing an Apache web server and a simple PHP script that has a remote command-execution vulnerability. We then use this script to download and execute a published exploit of CVE–2013–1763 [14], an array-index error in the Linux kernel. The steps we follow to detect and backtrack this exploit are illustrated in Figure 3.

Because this is a privilege-escalation vulnerability, a reasonable first step is to watch for threads' attempts to raise their privilege. We therefore start by using the Xen target to place a probe on `commit_creds`, the Linux kernel function responsible for setting thread privileges, and watch for threads that raise their privilege (i.e., become root). In general, we would not know when the exploit will occur and thus would have to insert the probe at system boot time. That would be undesirable, since `commit_creds` is a high-traffic function that would trigger often. Moreover, as becoming root is a common activity, we would be flooded with false positives. Stackdb provides solutions to these problems. It allows a probe to be installed in a disabled state; later, the probe can be enabled as a side-effect of triggering another probe. So we might instead place a probe on the `__sock_diag_rcv_msg` function, mentioned in the CVE as the source of the bad array reference. This function is not frequently used; by placing a probe there, we could dynamically enable the `commit_creds` probe. Stackdb also allows context-sensitive triggering of probes. We can further restrict our `commit_creds` probe to trigger only when it is invoked by the same thread that called `__sock_diag_rcv_msg`.

Once the `commit_creds` probe is triggered (Figure 3, step 1), we are provided with the thread id along with the new credentials as shown. When we detect an escalation to root privileges, the Xen domain is suspended. With the domain suspended, we can run multiple analyses against it.

The second step is to run a Stackdb-based backtracer over the Xen target to obtain a backtrace of all kernel threads and additional information about each thread. The stack trace and information for the offending thread (1081) are shown in the upper-left part of Figure 3, and they reveal crucial information. We see that `commit_creds` was called from a user-mode address with no backtrace information, not from a kernel address. This is a clue that kernel control flow passed through an invalid pointer that wound up in user space. The thread information shows the lineage of the offending thread. Importantly, it is the descendant of an `apache` process.

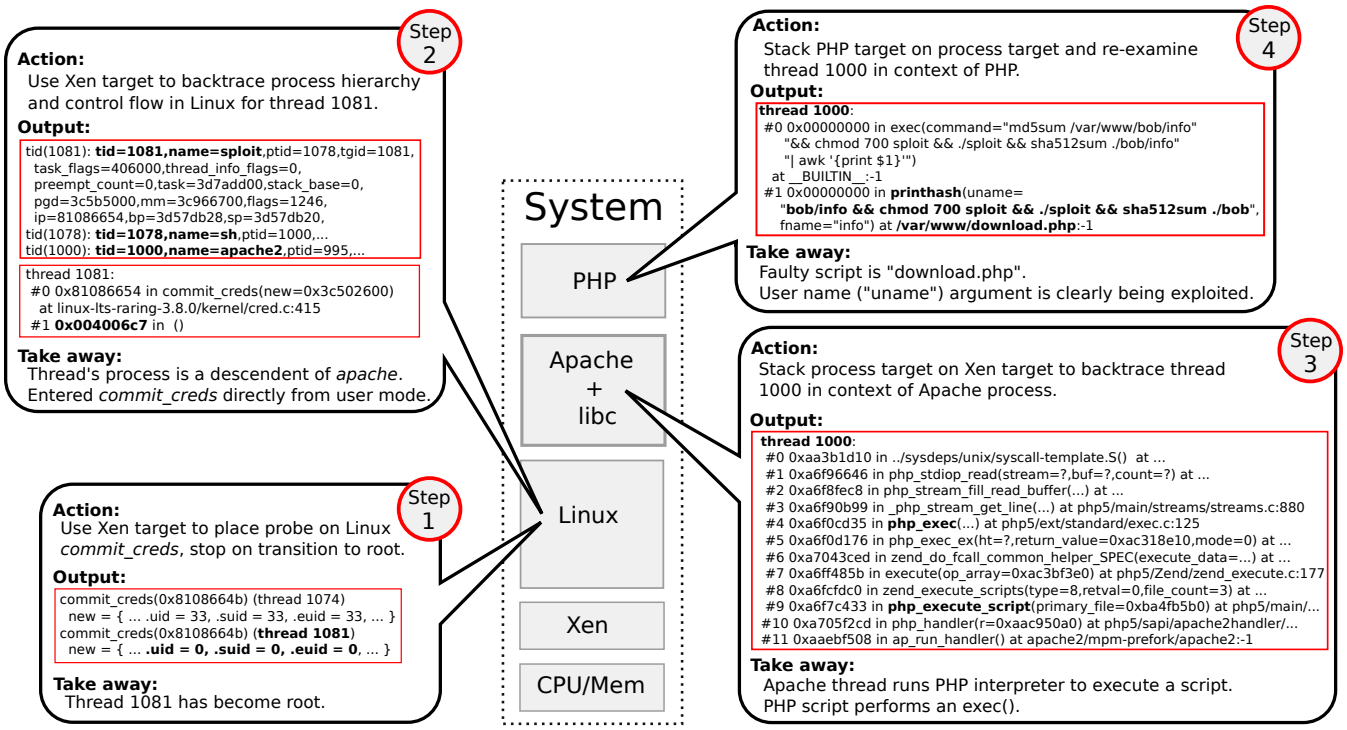

Figure 3. Multi-level analysis of a security exploit using three Stackdb targets

Now we use the multi-target capability of Stackdb. Our Apache server is configured with a fixed set of single-threaded server processes that run PHP. In step 3 (lower right), we run the Stackdb backtracing tool again, this time using a process target to focus on the user process in question (1000). The result is a C-level backtrace, which confirms that the Apache process is running the PHP interpreter and that the PHP script being run made an `exec` call.

In step 4 (upper right), we stack the PHP target, allowing us to trace back into the PHP code. Again, vital information is revealed. First, we discover the name of the executing script ("download.php"), which we can now examine in light of the stack trace. We also see a PHP `exec` call that executes multiple commands and clearly does more than just compute the hash of a file. From the arguments to the `printhash` function, we see that it is the user name (uname) parameter that introduces the multiple commands. By examining the PHP source in a text editor, we find the root cause of the command-injection exploit: the user-name variable is initialized from an unchecked HTML form variable.

7. Performance

We ran a set of experiments to characterize the performance of Stackdb's probes and the overhead introduced by target stacking. We ran our experiments on a Dell R710 with a single quad-core 2.4 GHz 64-bit Xeon E5530 "Nehalem" processor and 12 GB of RAM. The machine ran Xen 4.3 with a paravirtualized Linux 3.8 kernel on an Ubuntu 12.04 LTS base in both dom0 and the domU. The domU ran with a single virtual CPU and 1 GB of RAM.

We wrote two versions of a microbenchmark that calls an open-file function and the corresponding close function in a tight loop for a fixed number of iterations. One version is written in C and uses the Linux `open` system call; the other is written in PHP and uses PHP's `fopen` function. We ran our programs in domU and measured the time that each takes to make one loop iteration.

We then ran our programs again, under five configurations of Stackdb. We set up three configurations using the Xen base driver. (In all of these, our programs run in domU, and Stackdb runs in dom0.) The first uses only the Xen target, the second adds the process overlay target, and the third adds the PHP overlay target. We also set up two Stackdb configurations using the Ptrace base driver. (Our program and Stackdb all run in dom0.) The first uses only the Ptrace target, and the second adds the PHP overlay target. For each configuration of Stackdb, we placed a probe on the appropriate open-file function using the target at the top of the target stack. When the Xen target was the topmost (and only) target, we probed the `sys_open` function in the Linux kernel; when the process or Ptrace target was topmost, we probed the `open` function in libc; and when the PHP target was topmost, we probed PHP's `fopen` function. Finally, we ran our microbenchmarks in every Stackdb configuration and measured the time needed to make one loop iteration. The time includes the cost of handling one probe-hit event for the target at the top of the target stack.

Prog. Vers.	*Base-line*	Xen Base			Ptrace Base	
		Xen	Process	PHP	Ptrace	PHP
C	3.95	1,449	1,308	N/A	391	N/A
PHP	8.15	1,477	1,314	8,897	1,412	3,194

Table 3. Time (μsec.) to execute one open/close iteration of our microbenchmarks with a probe placed at various targets.

Table 3 shows our results. The second column shows the iteration times of the microbenchmark programs when no probes are installed, i.e., without debugger overhead. The next three columns show the timing results under the configurations of Stackdb that use the Xen base driver. The final two columns show the timing results under the Stackdb configurations that use the Ptrace base driver.

The Xen-based configurations rely on VMI-based probes, and our results show that there is significant overhead associated with this mechanism. This is due in part to the cost of virtualization: switching between between domU and dom0 on every probe, and translating addresses and reading memory across the domains. Even with this cost, the absolute performance overhead is not large (1–9 ms), and thus should be acceptable for interactive and scripted debugging.

An apparent anomaly is that probing a Linux-kernel symbol in the Xen target is *slower* than probing a user-process symbol in the process target, which is stacked on the Xen target (column 3 vs. column 4). This is an artifact of the open-file functions that we chose to probe at each level. In column 3, we used Stackdb to probe the Linux kernel's `sys_open` symbol, for which Stackdb found information about the function's arguments. In column 4, we used Stackdb to probe glibc's `open` symbol, and Stackdb did not find type information for this symbol. Every time the `sys_open` probe was triggered, Stackdb read the values of the function arguments, requiring multiple memory accesses. These additional accesses were not performed for `open`.

Columns 5 and 7 show that the PHP driver adds significant overhead. This is not surprising, given the amount of work that it must do to construct a PHP-level view of execution. The cost of reconstructing this view increases with the cost of accessing memory pages in its underlying target.

8. Related Work

Stackdb allows one to examine and control a single software system at multiple levels of abstraction. Overlay drivers and targets allow for multiple views of a system, and effectively, this means that one can combine multiple debuggers in order to better understand a whole system. The Blink debugger by Lee et al. [12] is also based on composition. Blink combines separate single-environment debuggers to create a unified debugger for systems that utilize multiple environments. For example, by sitting atop GDB and JDB, Blink implements a debugger for programs that utilize both C and Java. One can think of this as "horizontal" composition because the composed debuggers operate as peers—independent viewers—with Blink monitoring and managing the transitions between the two. In contrast, Stackdb can be thought of as "vertical" composition. A Stackdb overlay is not an independent debugger, but is stacked on top an underlying debugger (a Stackdb target). This is different from the composition implemented by Blink and leads to different capabilities. For instance, Blink would not be able to compose a process debugger with a VM debugger in a way that yields a debugger for processes within a VM. This is exactly the style of composition, however, that Stackdb supports.

One of the authors' primary uses for Stackdb is the analysis of VM guests, including both kernel-mode and user-mode processing. The Volatility Framework [18] is a related platform for examining VM guests. Volatility is extended by Python scripts that perform memory forensics, and there are libraries for tasks such as creating objects that represent user-mode processes. In contrast to Volatility, which focuses on the analysis of (static) memory snapshots, Stackdb supports analyses that are driven by the *execution* of the VM guest. Stackdb's target API allows a programmer to write analyses that install probes into a guest, extract data values, and potentially alter the guest's state. The primary challenges in building Stackdb arise from the implementation of debugger services, not just memory-analysis services.

Ho et al. describe PDB, a "pervasive debugger" for debugging Xen-based systems [9, 10]. PDB was integrated with the Xen VMM: a PDB server ran within the hypervisor and received commands from a PDB client, which could run on a separate host. The PDB server could access the VMM, guest kernels, and processes within guests. Like Stackdb, PDB required information about guest operating systems in order to locate process-level data. Unlike Stackdb, PDB lacked an internal abstraction for "stacking" targets in a general way. PDB was removed from the Xen source tree in September 2006 because it was not a core Xen feature [8].

Unlike Stackdb and PDB, which use breakpoints to implement debugger functionality, PinOS uses dynamic binary translation to perform whole-system analyses of Xen guests [1]. PinOS can run unmodified operating systems as Xen guests: it uses Pin [13] to rewrite the code of the guest just before it is executed, and as part of this rewriting, Pin can insert instrumentation as needed for analyses such as whole-system profilers. PinOS and Stackdb are therefore similar in that both are designed to implement analyses of unmodified guests. PinOS is well-suited to fine-grained analyses, such as instruction-level profiling, whereas Stackdb is well-suited to understanding behaviors at the level of source code, e.g., by setting breakpoints and watchpoints on source functions and variables. PinOS and Stackdb differ in the abstractions they provide to analyses. PinOS effectively interposes between the guest and the hardware, where as Stackdb provides whole-system, debugger-like access at multiple levels of abstraction: kernel, process, and application/language.

DroidScope is a platform for analyzing Android malware at multiple levels of abstraction [21]. Using dynamic binary translation, DroidScope tracks the execution of an Android hardware-platform emulator; using its knowledge of the Android software stack, it instruments the emulator's guest in order to find important OS-level and Dalvik VM-level events and data. DroidScope provides three APIs so that analyses can track the guest at the hardware, OS, and Dalvik VM levels. DroidScope is thus like Stackdb in that it provides multi-level debugging APIs, but unlike Stackdb, DroidScope's APIs are different at every level. Stackdb defines a single target API, implemented all targets, used both by analyses and for building target stacks. In contrast, DroidScope does not define an internal abstraction for composing targets.

9. Conclusion

Stackdb is a debugging library that allows a client to observe and control a whole system, such as a VM guest, at multiple levels of the system's software stack. A Stackdb target corresponds to a particular abstraction layer or a portion of the system being debugged. The key insight of Stackdb is that a debugger can be organized as a stack of targets, in which the targets for the higher levels of a system are implemented atop those for the lower levels. Our implementation of Stackdb supports various combinations of targets into stacks, including three-level stacks that provide access to the OS, process, and language-runtime layers of a VM guest. As detailed in our security-focused case study, Stackdb can help to close the semantic gap that is often encountered in VMI-based and whole-system analyses.

Software Stackdb is open source and available for download at `http://www.flux.utah.edu/project/a3`.

Acknowledgments

We thank the anonymous VEE '14 reviewers and our shepherd, Galen Hunt, for their comments on drafts of this paper. We performed our experiments on machines provided by the Utah Emulab testbed [20]. This work was supported by the Air Force Research Laboratory and DARPA under Contract No. FA8750–10–C–0242.

References

[1] P. P. Bungale and C.-K. Luk. PinOS: A programmable framework for whole-system dynamic instrumentation. In *Proc. VEE*, pages 137–147, June 2007.

[2] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proc. HotOS*, pages 133–138, May 2001.

[3] J.-H. Chiang, H.-L. Li, and T. Chiueh. Introspection-based memory de-duplication and migration. In *Proc. VEE*, pages 51–61, Mar. 2013.

[4] distorm@gmail.com. distorm - Powerful Disassembler Library For x86/AMD64. `http://code.google.com/p/distorm/`.

[5] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. IEEE S&P*, pages 297–312, May 2011.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. NDSS*, Feb. 2003.

[7] GDB Developers. GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[8] A. Ho. Personal communication, Nov. 2013.

[9] A. Ho and S. Hand. On the design of a pervasive debugger. In *Proc. AADEBUG*, pages 117–122, Sept. 2005.

[10] A. Ho, S. Hand, and T. Harris. PDB: Pervasive debugging with Xen. In *Proc. GRID*, pages 260–265, Nov. 2004.

[11] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proc. SOSP*, pages 91–104, Oct. 2005.

[12] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: Portable mixed-environment debugging. In *Proc. OOPSLA*, pages 207–226, Oct. 2009.

[13] 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 *Proc. PLDI*, pages 190–200, June 2005.

[14] The MITRE Corporation. CVE–2013–1763, Feb. 19, 2013. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1763`.

[15] B. Payne et al. vmitools - virtual machine introspection tools. `http://code.google.com/p/vmitools/`.

[16] The PHP Group. PHP at the Core: A Hacker's Guide. `http://www.php.net/manual/en/internals2.php`.

[17] A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In *Recent Advances in Intrusion Detection*, volume 6307 of *LNCS*, pages 97–117. Springer, 2010.

[18] Volatile Systems. The Volatility Framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.

[19] J. Wessel. Using kgdb, kdb and the kernel debugger internals. `http://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/`.

[20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[21] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. USENIX Security*, pages 569–584, Aug. 2012.

[22] F. Zhang, K. Leach, K. Sun, and A. Stavrou. SPECTRE: A dependable introspection framework via system management mode. In *Proc. DSN*, pages 1–12, June 2013.