Traps, Events, Emulation, and Enforcement: Managing the Yin and Yang of Virtualization-based Security

Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith
Dartmouth College
{sergey,locasto,ashwinr,sws}@cs.dartmouth.edu

ABSTRACT

We question current trends that attempt to leverage virtualization techniques to achieve security goals. We suggest that the security role of a virtual machine centers on being a policy interpreter rather than a resource provider. These two roles (security reference monitor and resource emulator) are currently conflated within the context of virtual machines and VMMs. We believe that this "double-duty" leads to both a significant performance impact as well as a bloated virtualization layer. Increased complexity reduces confidence that the code is elementary enough to verify or trust from a security perspective. Ironically, as more security–related functionality is shoved into a VM platform, the system becomes less *trustworthy* as it becomes increasingly *trusted*.

We argue that a principle reason for such an unfortunate situation is the lack of efficient hardware trapping mechanisms. We propose an architecture to help ameliorate this problem by transferring the security enforcement and program analysis roles from the virtualization component to a policy-directed FPGA.

Categories and Subject Descriptors

B.7.1 [**Hardware**]: Types and Design Styles—*Gate Arrays*; C.1.3 [**Computer Systems Organization**]: Other Architecture Styles—*Data-flow architectures*

General Terms

Design, Performance, Security

Keywords

debugging, virtualization, traps, security policy

1. INTRODUCTION

Virtualization serves as a method to intercept device access or system execution in order to multiplex the underlying hardware. Naturally, the isolation properties of this technique are attractive from a security perspective, as is the ability to inspect the details of that execution or the ability to tweak the isolation boundaries. Thus, in addition to providing multiplexing or emulation[1] of devices, virtualization becomes a convenient mechanism for examining and enforcing security policies. The conceptual simplicity of isolation as a security primitive (and the resulting relative simplicity of isolation-based policies) have made virtualization a popular policy implementation tool.

Yet, the conceptual simplicity of isolation-based policies comes at the high price of having to emulate entire "machines." This emulation necessitates the creation of numerous "virtual device" drivers that then form part of the trusted code base, greatly increasing its size and complexity [11]. This bloat is hardly conducive to improving the security of virtualization-based policy solutions.

Another important practical aspect of using multiple virtual machines is the need for a privileged management and administrative interface. Such an interface can help an administrator deal with a collection of virtual machines in a more streamlined fashion than managing the equivalent collection of physical hardware. These interfaces, however, represent an attractive attack surface, as demonstrated by several virtual machine escape techniques[2]. We see this avenue of attack as yet another consequence of expanding the trusted code base due to the practicalities of using emulation to achieve security goals.

Of course, the practical benefits of virtualization's security uses are hard to deny, which suggests that these benefits are due to a deeper underlying principle than *isolation through emulation*. The activity of trapping (the ability to intercept execution at just the "right" moments), is overloaded: different approaches to virtualization are predicated on trapping certain classes of events of interest needed to accomplish a VM's role as a resource provider. Furthermore, the need to trap other classes of events (such as those that matter from a security perspective) has, to a large extent, been constrained by the types of events that the VM system might already be trapping to accomplish its role as a resource provider. We assert that these classes of events are quite distinct, and this paper attempts to elucidate this principle by identifying the (mis)use of trapping in VMs for security benefits. In other words, our goal is to *untangle security and virtualization* by differentiating between the twin needs to trap both security-related events and emulation-related events (of which there may be overlap).

1.1 Approach: Focus on Traps

To this end, we stress this key principle: it is the ability to comprehensively and flexibly *trap program execution* that is fundamental to all virtualization technologies. Moreover, the trapping capa-

[1]We use the term *emulation* loosely to denote a variety of approaches to supervised or instrumented execution.

[2]E.g., `http://www.itsecurity.com/features/virtualization-security-061708/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VMSEC'08, October 31, 2008, Fairfax, Virginia, USA.
Copyright 2008 ACM 978-1-60558-298-6/08/10 ...$5.00.

bilities of a virtual platform are central to ensuring any trustworthiness properties of that platform, by determining which program behaviors will be trapped (and thus mediated) by the platform.

In a nutshell, a comprehensive trapping system provides the basis of all virtualization technologies. We argue that it is also the primary source of virtualization's power as a security primitive, and we believe that it should be separated from emulation proper, which, although it relies on trapping, in practice can do more harm than good to achieving security goals.

We note that *debugging* is another area where comprehensive trapping systems play a central role in tracking programs' behavior (and ultimately validating the programmer's assumptions about its trustworthiness). Thus, there exists a deep link between debugging and policy enforcement, since both activities aim to relate the expected program's behavior (on which the programmer or the administrator base their trust in the program) with its actual behavior.

We therefore approach the problem of untangling security and virtualization from a rather unconventional direction: we are motivated by the need to greatly decrease the performance cost of debugging and runtime program analysis. As such, this paper largely focuses on issues related to debugging and program analysis rather than traditional virtualization-related security topics like isolation, compartmentalization/MLS, or honeypots.

While unconventional, this approach is not out of scope: virtualization is useful for analysis as well as isolation. For instance, debugging software manually remains one of the few effective means of addressing security and reliability issues in program code. In addition, anti-malware researchers and professionals employ debugging techniques to reverse–engineer malcode, and they often run malware samples in emulated or virtualized environments to discover their dynamic runtime behavior.

1.2 Limitations of Trapping Support

Despite the increasing complexity and burgeoning feature set of modern virtualization environments, debuggers, and IDEs, software analysis remains a manual art: an iterative, interactive, and time–consuming process. Although one can perform analysis of small programs by using ordinary breakpoint or single-step routines in combination with manual checking for conditions of interest, such techniques no longer suffice.

This state of affairs is primarily derived from the limitations of trapping support in modern hardware. Designs of hardware and memory systems rarely contain efficient interception and inspection facilities to aid debugging efforts; rather, these systems are meant to run program code as quickly as possible. Debugging or analysis frameworks often perch on an overtaxed, ungainly hardware kludge. A structured approach to this problem can both obviate the need for the current use of non–conventional, obscure CPU features and facilitate building semantically rich trapping systems. It seems that we need to increase the expressive power of descriptions of breakpoint (*i.e.,* fault or trap) conditions.

We must match this increase in expressive power with a corresponding change in hardware support. We envision a trapping system that automatically dispatches "false positives" (*i.e.,* events that are not of interest to the analyst) *without* significantly slowing the program. Currently, all such breakpoints involve an interrupt of program execution, even if the event is ultimately not of interest.

Our key proposal is to offload condition checking to a "debugging policy" interpreter encoded on an FPGA. Figure 1 demonstrates how our proposed architecture affects the current service path for traps. Although current traps might be relatively cheap, gathering state and evaluating the conditions on that state at each trapped event is expensive. We aim to greatly reduce the cost of

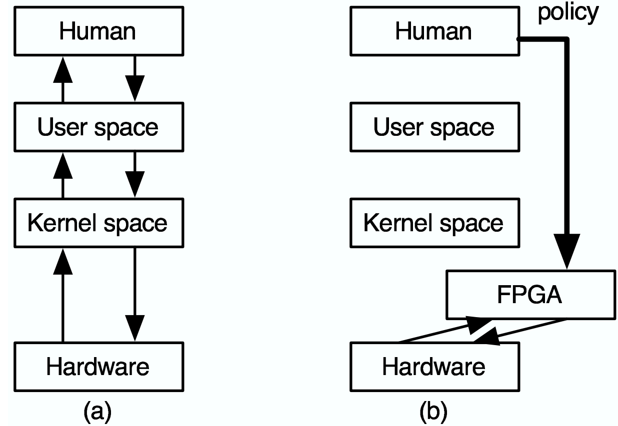

Figure 1: ***Transforming the Memory Trap Service Path*****. Currently, as shown in (a), servicing an interrupt for debugging purposes involves traversing the system stack twice, which incurs both human processing time as well as two context switches (between the traced and tracing process). We propose to insert an analysis policy as well as an FPGA to interpret that policy (see (b)), thus avoiding this traversal and concomitant context switches for the average case. See Figure 2 for an illustration of where particular analysis tools and infrastructure reside in this service path.**

condition evaluation while increasing the expressive power of debugging statements. One potential benefit of doing so is to achieve a degree of automated, policy–driven debugging, where the developer or reverse engineer encodes a debugging or analysis plan rather than undertaking a manual sequence of specific memory manipulations and examinations.

1.3 Better Trap Semantics

What does this all have to do with virtualization? Besides the high-level use of virtualized environments to support efforts like malware analysis [24], the application of virtualization to the systems security domain has worked to conflate two roles within the virtualization system: the role of reference monitor (enforcement) and the role of resource provider (emulation). Each of these roles is predicated on the ability to intercept and measure execution by means of an event trapping framework. It is unclear, however, if the events needed to support emulation remain the same as the events of interest for interpreting some security policy.

The connection between a trap system for a particular platform and the security properties of that platform are directly and intimately related. Abnormal system transitions should cause traps, after which the system's state may no longer be considered trustworthy. Accordingly, much of a security system's essential functionality is implemented inside trap handlers. While such mechanisms can be implemented purely in software, in practice they rely on hardware-supported traps whenever possible so that application code executes at full speed between mediated events. Thus, traps form a core mechanism upon which to implement security policy interpreters. As such, they directly or indirectly influence all aspects of the latter. In our opinion, the details of the trap system de-facto shape the capabilities and performance of the policy system. Therefore, we believe that a flexible hardware trap system (one that allows execution to proceed at the highest possible CPU

speeds until an event of interest occurs) is a necessary condition of increasing trustworthiness — and therefore security, of software.

The key issue here is that the set of trapping conditions that we can describe with hardware primitives is severely limited. On the x86, reads, writes, or instruction fetches from fixed memory locations can trigger traps, but the x86 neither provides hardware nor enables efficient software support for describing any *combinations* of these events, such as simple *predicate logic* (*e.g.,* for specifying additional conditions being satisfied at the time of the access), *temporal logic* (*e.g.,* for specifying that an event B should only be trapped if it occurs after an event A), or *linear logic* (for tracking use of finite resources and state changes). In fact, it seems that the current design does not target any formal logic model.

In other words, evaluating program logic conditions described with any of the logics above requires costly propagation of the relevant information to the tracing process (in most cases, the debugger process), which stores it and performs the actual evaluation. We believe that including support for expressing such common program conditions efficiently, with hardware support for recording even a limited amount of state and appropriate logic, will bring about qualitative improvements to both debugging and verifying policy assertions about programs.

Recent examples of unorthodox ways (see Section 2) to both store high–level state in hardware page table control bits and manipulate existing trap handling mechanisms strongly suggests that developers require a more efficient mechanism to evaluate the types of logical conditions we mention above. These mechanisms should support checking and setting hardware-maintained data as well as acting (by invoking a higher–level handler) only on the relevant combinations of conditions they process.

1.4 Contributions

Trying to eliminate software bugs before deployment and tracking down reported errors post deployment remains one of the primary methods for evolving a software system toward a more secure state. In addition, reverse engineering and malware analysis often require a virtual environment to aid inspection. Unfortunately, the process of program analysis does not seem to scale with the size of new software systems or the rates of exploitable bugs. Although Agrawal [2] introduced novel debugging techniques (*e.g.,* program slicing and backtracking), almost no progress has occurred on increasing the efficiency of trapping on memory events: the core operational requirement of debugging. Furthermore, very little progress has been made to automate the process of reasoning about errors during debugging.

We identify a new approach for program analysis and security policy support to both increase the expressive power of this analysis as well as reduce the performance cost of intercepting execution. We begin by examining several motivating examples and several systems that aim to provide richer debugging primitives. We then generalize them to propose a set of hardware-supported features that would make the implementation of these mechanisms both simpler (by delegating much of the present functionality down the system stack) and more efficient (by delegating the most frequently performed tasks to kernel and hardware levels).

2. MOTIVATION

This section considers several motivating examples that highlight the need for a faster and more expressive trap system. As Figure 1 shows, the use of an FPGA and a debugging policy can remove the need to completely traverse the software stack (and incur a context switch) twice for each trappable event.

Not surprisingly, a number of ad-hoc solutions recently arose to help fill this growing need for better trapping mechanisms. Many of these techniques rely on non-conventional use of x86 memory architecture features and provide far from a comprehensive solution. For example, in order to express the policy: *"an instruction was executed from a memory page recently written to by the current process"*, the OllyBone[3] debugger extension relies on exploiting a combination of two hardware features. First, the x86 uses separate ICACHE and DCACHE TLBs for fetching instructions and data, respectively. Second, it relies on invalidating TLB entries and adding the respective clause to the page fault handler.

Note that none of the available built-in x86 debugging mechanisms offers a simple way to simultaneously define this kind of a trappable event while allowing the code to execute at natural speed prior to its occurrence. Compared to traditional debugging functionality, emerging approaches to debugging exhibit several common trends (listed below). These trends provide an implied set of desiderata for a more mature debugging and trapping system.

- Developers or reverse engineers must find creative ways to manipulate the hardware "trapping bits" and their trap handlers (*i.e.,* the x86 special register flag and descriptor entry bits) to express Boolean or temporal combinations of conditions that are ultimately trappable with built-in traps.
- Some frameworks increase the extent and nature of context pertaining to the previous history of the process so that the developer or system can use this context at the point of deciding whether or not to trap.
- Some frameworks allow access to the debugged process's OS environment, including its process information block.

2.1 Debugger Watchpoints

Debugger watchpoints [23], as implemented in the popular debugger GDB, enable the debugger to break execution of the traced process when both a memory access to a variable occurs *and* a certain predicate evalutes as true. Watchpoints can provide a powerful debugging mechanism that allows the user to automate checking for additional conditions of interest, thereby saving the time that needed to manually check the predicate at the breakpoint prompt.

Unfortunately, as the GDB manual states, execution in the presence of watchpoints can be hundreds of times slower that normal. This slowdown is due to both the limitations of hardware support for memory breakpointing and the high cost of context switches between the debugged process and the debugger process, where the information and logic necessary to evaluate the predicate is actually kept. Thus, the watchpoint mechanism arguably presents one of the biggest disappointments of code development to novice debugger users — despite the great power of this method in theory, it often turns out to be unsupportable in practice. Even so, as the authors of GDB note, "this may still be worth it, to catch errors where you have no clue what part of your program is the culprit."

From our point of view, watchpoints highlight the fundamental imbalance between event primitives available for code and data: code can be instrumented at instruction granularity, via either single-stepping instruction execution mode or as many breakpoints as necessary (the basic trappable event being "instruction at a given address was fetched to be executed"). Memory event instrumentation, on the other hand, is limited to either a few hardware watchpoints or page granularity. In other words, the primitives available for debugging are heavily biased toward code granularity rather than

[3]OllyBone (http://www.joestewart.org/ollybone/) can catch the moment when the main body of polymorphic malware payload finishes decrypting and begins execution.

memory granularity. This continued, implicit preference toward analyzing code based on control flow stands in apparent contrast to the old dictum that understanding data gets one much further towards overall program understanding than control flow[4].

2.2 Costly State

Consider the following example of using a watchpoint. Suppose we want to break on the 200th write to the watched variable. To accomplish this goal, the debugger's watchpoint handler needs to increment and keep track of a counter. Note that we assume the best possible operating situation for the current generation of watchpoint–based debugging, where the watchpoint mechanism is hardware-assisted (*e.g.,* by the debugger process setting an `x86` DR register pair with the watched address and the descriptor of the watched span of memory locations at that address).

Generally speaking, for this scenario, we need to maintain 8 bits of state. Unfortunately, these bits and the logic to process them will be kept in the debugger process, necessitating control transitions all the way from the hardware layer to the user space software layer and back (as shown in the first half of Figure 1); also, we incur the costs of switching between the original process's virtual address space and that of the debugger where the counter is kept[5]. Therefore, including these bits of state in the description of our desired trapped events translates to substantial costs.

Moreover, even a single bit of state would incur the same costs. For example, in the case when our desired event includes temporal logic such as "write to a watched variable AFTER another variable was written" (or, in general, "write to a watched variable AFTER another event occurred"). When debugging information flow, these types of events are of interest; they are also the core events to watch for a security policy regarding information flow to be efficiently enforced[6]. Again, the debugger must maintain that single bit of state needed to indicate the occurrence of the precondition event.

2.3 Catching Malware Unpackers

Malware analysis provides another motivating example of the need for more expressive debugging primitives and less expensive debugging support mechanisms. In order to frustrate static analysis attempts, malware authors often protect their code by one or more layers of encryption or packing: an "unpacker" or decoder preamble extracts the malware. Catching the moments when the encrypted or packed malcode is extracted and begins execution is crucial for analysis. In addition, reverse engineers often employ a virtual machine environment or emulator such as Bochs[7] or QEMU [5] to analyze the behavior of a malcode sample.

The malware author can purposefully lengthen the unpacker section to make manual tracing challenging for the analyst. In addition, longer preambles can delay execution of the code in a virtual machine environment, thereby making tracing a time-consuming activity for an analyst. Longer preambles can also help malcode evade automated analysis mechanisms that timeout after a certain number of instructions or wall clock time passes without "interesting" events. Finally, malcode can detect a VM environment (*e.g.,* by executing an instruction that is not virtualizable — several such instructions exist on `x86`) and abort execution.

An analyst can greatly benefit from a trapping mechanism that would allow the unpacker to execute at (close to) normal hardware speed, yet trap near the point where the unpacked code started executing. In essence, such a mechanism would fool the malware into thinking it is executing natively (for all practical purposes, it is), yet enable the analyst to step in at a crucial point in execution.

As we illustrated earlier, the OllyBone debugger provides a mechanism that does just that by manipulating the separate `x86` TLBs for code and data in a fashion unintended by the designers of IA-32. In essence, OllyBone can automatically trap the event of "instruction was fetched from a memory location previously written by the process", executing up to that point with minimal overhead compared to bare hardware speed. We see this example as highly significant for two reasons. First, it makes use of `x86` hardware trapping features to let the execution proceed at almost normal hardware speed — an essential requirement for the task, which cannot be accomplished by existing conventional techniques. Second, it uses hardware bits to track the state and co-opts elements of the hard-coded address translation logic to trap the desired complex trapped event ("execution from a location after a write"), essentially pushing most of the time-critical work down from the expensive debugger userspace process to the much more efficient hardware layer.

3. INSTRUMENTATION ENVIRONMENTS

Recent debugging frameworks can intercept process execution in a number of ways. Here, we summarize three approaches to program interception at various levels of the system stack, including Pin [14], DTrace [10], and Kprobes and SystemTap [19]; we covered some informal "hacker debugging" techniques used in OllyBone in Section 2 (other examples of these approaches include ProcessStalker and RastaDebug). Two important points of comparison include each system's flexibility to specify "interception conditions" as well as whether the trap is precise or asynchronous (*i.e.,* is the trap on the critical path or alongside it or activated within some time limit).

3.1 Hierarchy of Trap Handlers

Given that the underlying `x86` hardware can trap memory access and instruction execution events with only the simplest descriptions such as *instruction fetch at an address* or *read/write from/to an address*, the rest of the "intelligence" that drives a trap system has be handled in the upper layers of the software stack. For the sake of the following discussion, we display these layers and our example systems in Figure 2.

Human use of debugging tools like gdb, IDAPro, and strace involves careful, repeated, and interactive management of the debugger software. Although debuggers and user-level program supervision frameworks contain many features, extensive programmability rapidly begins to degrade performance, which prohibits debugging live production systems. The kernel can provide almost zero-cost hooks, but still reflects the tension resulting from the lack of support from below and the demand for flexible debugging from above. Although VMMs like Xen are "physically" positioned between a full-blown OS and the hardware, they experience the same constraints that the OS has (static and limited trap filtering capabilities of the underlying hardware). We propose to insert a programmable hardware component to help with the cost of servicing expressive debugging policies, as shown in Figure 1.

[4] *"Show me your flow charts and conceal your tables and I shall continue to be mystified, show me your tables and I won't usually need your flow charts; they'll be obvious." – Brooks,* The Mythical Man-Month *[7]*

[5] Although some architectures (*e.g.,* modern SPARC) might decrease this cost by supporting an alternate address space where a copy of the debugged process can live, the issue here is the separation between the debugger and the debugged process. The possibility of doing debugging more cheaply because we can abuse a unique hardware feature is a symptom of exactly the problem we try to address.

[6] Consider a policy based on the proposal `http://cr.yp.to/unix/disablenetwork.html`.

[7] `http://bochs.sourceforge.net`

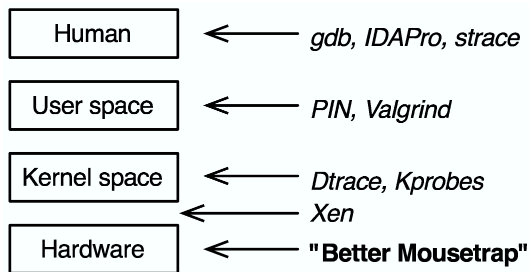

Figure 2: ***Handler Hierarchy.*** **Forcing humans to manually evaluate a condition results in a slow and iterative debugging procedure. Although software can help alleviate the demand for more flexible debugging and program analysis procedures, this level of programability begins to rapidly degrade performance. The OS can provide some support, but it remains limited by the capabilities of the underlying hardware. We propose adding our "better mousetrap" at the hardware layer.**

Our proposed changes, described in Section 4, focus on the hardware level. The DTrace and Kprobes systems have their core functionality implemented at the kernel level. Decisions made here incur the cost of a trap handler invocation through the `x86 IDT` and are typically measured in microseconds. Program instrumentation frameworks like `strace(1)` and Pin [14] are userland facilities that rely on the underlying OS debugging support but do not modify it. Decisions made here incur the cost of a context switch to the process that evaluates the decision's logic as well as the cost of copying appropriate data structures between the virtual address spaces. Typically, we can measure the total cost of these operations in at least tens of microseconds. The most expensive conceptual level of debug event handling is the human level. Ultimately, the human analyst decides whether a given trapped event was of interest to the task or in fact a "false positive." By increasing the expressive power of watchpoint policies, we aim to limit the amount of complex event filtering decisions made by a human. This goal complements our aim to reduce the cost of servicing a trapped event.

3.2 Pin

Both Valgrind [16] and Pin [14] provide customizable runtime interception of program execution (*i.e.,* something more feature–rich than `ptrace(2)`). Pin provides an API that exposes a number of ways to instrument a program during runtime, including intercepting and observing properties of individual instructions, basic blocks, functions, binary sections, and binary images. The Pin API provides a number of methods for obtaining the values of both hardware and process–level context, including access to function arguments and return values, the contents of hardware registers (and thus the process stack) and thread state.

Despite the capabilities of the Pin API, the ability to write Pin tools as ordinary C++ programs, and Pin's ability to instrument programs at a very fine granularity, Pin still imposes a considerable performance penalty: depending on the nature of the instrumentation, Pin can insert tens or hundreds of extra instructions *per native instruction*. Even if this instrumentation executes at native speed after the first time it is exercised, this work represents a noticeable slowdown. For some tools we have implemented, this slowdown can amount from a 2X to a 30X performance hit. In addition, Pin supervises only user space code; it does not instrument the kernel (although PinOS uses the Xen VMM to partially address this issue).

3.3 DTrace

Sun Microsystems designed, built, and ships DTrace [10], a dynamic tracing toolkit for the Solaris 10 (and Mac OSX) operating system. It provides the ability to instrument any part of the running kernel by inserting "probes" (usually at function boundary points) and specifying actions to be performed when the probes are encountered during both kernel and user space execution. DTrace interprets scripts in the D language; these scripts trace or instrument the kernel. A D program is compiled into an intermediate format before executing it against the running kernel, and the language and the runtime environment guarantees some protection against system abuse or accidental programmer errors.

One of DTrace's drawbacks is that it cannot be used for *strict* policy enforcement since the probe handlers are only partially inline, i.e. they do not divert and "grab" kernel control flow like traditional system-call wrappers do, since probe handlers are executed aynchronously within the kernel. And also the policies, if written, would be limited by the capabilities of the D language itself. While DTrace serves as an excellent low-latency observation mechanism, one cannot use it to enforce traditional security policies.

Recently, impressive DTrace extensions RE:Trace and RE:Dbg have been presented by Beauchamp and Weston [4, 3], which take advantage of the DTrace architecture and combine it with disassembling and debugging capabilities. Notably, these tools extend DTrace trap and script semantics to the full power of the Ruby object oriented language, providing even greater flexibility and expressive power than the D language of DTrace. We refer the reader to the conference presentations of the above tools for their motivation and the unprecedented set of features.

3.4 Kprobes

Kprobes is a dynamic instrumentation framework for Linux that allows the user to insert arbitrary instruction–level breakpoints in the kernel and handle them with C code residing in one or more kernel modules. Both DTrace and Kprobes provide minimal latency when no probes are activated on the running kernel. We believe that this property is critical for tracing systems.

3.5 SystemTap

While Kprobes provides finer granularity than DTrace in terms of probe location, probe handlers are defined in kernel modules and run in the same context as the kernel. Therefore, faulty code in these probe handlers has the ability to crash the kernel (unlike DTrace where all exceptions are handled gracefully). Furthermore, the programming environment of Kprobes remains inaccessible to anyone not familiar with careful handling of the kernel locks, data structures, and APIs.

The SystemTap project [19] by Redhat, IBM, and Intel attempts to address the limitations of Kprobes by augmenting it with a user–level scripting language. The semantics of this scripting language are similar to *AWK*, and the language contains some features from C. One major benefit of SystemTap is that the scripting language insulates the programmer from directly manipulating code that is inserted into the kernel.

4. PROPOSED HARDWARE FEATURES

Changing the way memory events are trapped and serviced requires both programmability and speed. In essence, we need an architecture that will simultaneously allow more complex analysis and a faster overall (amortized) trap service speed. We propose an architecture that contains two primary components. First, an FPGA configured to act as a memory event stream parser interacts with the CPU and MMU to obtain a stream of memory events and a series of

interrupts. Second, a memory event analysis policy is loaded into the memory of the FPGA to direct the actions of the FPGA parser circuit. With this architecture, we hope to satisfy the twin demands of more flexible analysis and better trap handling performance.

4.1 Hardware Modifications for VM interaction with the FPGA

In hardware, we physically align the FPGA near the MMU so that the MMU can consult the FPGA for validating memory pages. But the MMU somehow needs to know which pages to consult the FPGA for, and we propose modifying the MMU to add a `trace` bit to each page table entry (PTE) that indicates if the MMU needs to watch that page.

When loading a policy, the FPGA first determines the addresses specified by the policy that is being loaded. It then sets the `trace` bit for the pages corresponding to those addresses. But these addresses are virtual, meaning there might not yet exist a virtual-to-physical mapping for these pages. To establish this mapping and thus ensure that the MMU has a PTE entry for each virtual page, the FPGA first generates page faults on behalf of the process whose policy is being loaded. This forces the MMU to establish the virtual-to-physical mapping. Then the FPGA retrieves the necessary PTEs, sets the `trace` bit and activates the policy. Similarly, when deactivating a policy, the FPGA unsets the `trace` bit for each page that it was previously monitoring.

We now consider how the MMU traps to either the kernel or the FPGA depending on which memory address is being accessed by the processor. Traditionally, the MMU generates a page fault which is then handled by the kernel. The kernel then services the fault by mapping in the requested page. We still retain this basic model, but in addition to it, the MMU checks if the `trace` bit is set for the page being accessed, and if so, consults the FPGA. The FPGA in turn checks the validity of the access against the activated policy and performs the neccessary actions (such as `ALLOW`, `DENY` etc).

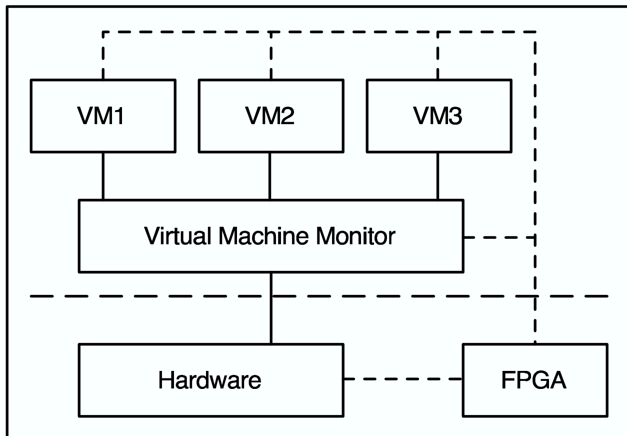

Figure 3: ***A Block Diagram of the proposed architecture.*** **The bold lines indicate traditional VM interactions and the dotted lines indicate interactions with the FPGA.**

The MMU and the FPGA in our design interact to accomplish the following:

1. When a process acccesses a memory page, if the PTE for that page has its `trace` bit set, then the MMU transfers control to the FPGA to determine the validity of the access.

2. The FPGA then searches through its policy list to find a policy that moderates that memory event. The FPGA verifies the memory event using the policy and retrieves any needed state from the kernel.

3. If the memory event satisifies the policy, the FPGA returns `SUCCESS` to the MMU; otherwise, a violation exists, and the FPGA signals `FAILURE`.

4. In the case of a `SUCCESS`, the MMU proceeds as usual, *i.e.*, it performs the action indicated by the memory event (read/write) on the requested page. In case of an error, the MMU raises a page fault to the kernel, where our policy-handler portion of the page fault handler recognizes that a policy violation has occured, and takes the necessary action, which might include logging the access, killing the process, panicking the system *etc.*, as dictated by the policy itself.

In addition to the collaboration with the MMU, the FPGA also needs to retrieve processor state such as the condition of the registers, TLB entries, L1/L2 cache contents etc. In order to facilitate this, we suggest placing the FPGA in close proximity to both the CPU and the MMU hardware.

4.2 VMM Interaction with the FPGA

While the primary purpose of the FPGA is policy interpretation for each individual VM, we also construe the FPGA as a fast computation unit, capable of interpreting policies dictated at the VMM level, specifically policies that deal with controlling the kind of shared data and general communication that occurs between guest virtual machines, whose policy dataset can typically be huge. The VMM interaction with the FPGA is necessary for two reasons:

1. To allow the FPGA to recognize which VM is currently running on the system.

2. To interpret policies for the VMM itself.

Because of the likely existence of more than one virtual machine on the same system, the FPGA needs to identify the currently running VM and enforce policies on that VM's applications only. In order to accomplish this, the VMM conveys sufficient information regarding the currently executing VM to the FPGA on every VM context-switch. The FPGA then saves its policy dataset for the current VM and subsequently loads the policies for the new VM.

Additionally, the VMM also has access to the FPGA to enforce policies on the virtual machines themselves. This happens by first preloading the FPGA (typically on bootup) with the necessary policies, and then activating them as needed at runtime.

5. POLICY LANGUAGE

Developers sometimes find it difficult to undertake accurate behavioral analysis (an essential part of reverse engineering) precisely because malware (and even benign software) spends large amounts of time engaged in startup routines: work common to most applications or irrelevant to the behavioral analysis. As we mention in Section 2, software tracing using standard breakpoints and watchpoints can slow this analysis down. Given that malware rapidly evolves to purposefully frustrate *ad hoc* unpacking tricks, we believe a structured approach to providing comprehensive debugging hardware primitives will catalyze the malware analysis field.

We are in the process of designing a policy language for debugging. At present, we have not specified a formal, complete grammar. The examples that follow are slightly more structured than psuedocode, and we express them in a C-like dialect with several

Table 1: ***Features of Instrumentation Tools.*** **Here we compare the feature set of existing approaches to debugging with our Better Mousetrap proposal. The "performance" row indicates a system that imposes a relatively small overhead during normal system operation. Flexibility indicates that the system provides a rich API for accessing and modifying process state and context. Interposition indicates that the system is able to "stand in the path" of a program's process (*i.e.,* it does not operate asynchronously). Asynchronous operation leaves the door open for an attacker to bypass the system.**

	Pin	DTrace	Kprobes	gdb	*BMT*
performance		√	√		√
flexibility	√				√
interposition	√		√	√	√

```
define var memFoo = MEM[ 0xc0801040 ..
                         0xc0803000 ];
define var instr =
           FUNC[list_add,list_del];

rule moderateFoo
{
  if( w(memFoo) &&
      !within(REG[EIP], instr))
    ALERT();
}
```

Figure 4: ***Policy MemAccess.***

```
// decide on virt/phys address
define var foo = MEM[ 0xd0130560 ..
                      0xd0130564 ];
define lock fooLock = foo + 16;
define var instrs = FUNC[mutex_lock];

rule protectFoo
{
  if( (lock_status(fooLock) !=
       LOCK_HELD) && ( w(foo) ))
    ALERT();
}

rule protectFooLock
{
  if( w(fooLock) &&
      !within(REG[EIP], instrs) )
    ALERT();
}
```

Figure 5: ***Policy LockControl.***

built-in features, keywords, and predicates. This current, unfinished form of the policy language contains the primitives listed in Table 5. We next consider a selection of some policies (in a rough form of our envisioned memory–event based debugging policy language) that are impossible or extremely inefficient to write with traditional trapping models.

Policy MemAccess.

Frequently, access to a particular block of memory needs to be moderated by allowing only a predefined set of instructions. For example, a particular instance of this policy (see Figure 4 for an example) might allow only the kernel function `list_add` and the kernel function `list_del` to modify the `next` and `prev` pointers of any doubly-linked list in the kernel. This arrangement would essentially prevent malicious code striving to hide processes, files, or Loadable Kernel Modules (LKM) by modifying the lists directly through the $kmem$ device. Efficiently protecting data structures in the kernel from such code while retaining the ability to dynamically patch or extend the kernel through the same $kmem$ or LKM interfaces remains an active area of research.

Policy LockControl.

Since the Linux kernel is multi-threaded, data structures in the kernel are frequently protected by locks embedded in the structure itself. Code accessing portions of the structure follow the familiar pattern: acquire the lock, access the fields in the structure, release the lock. However the above sequence does not apply to $kmem$ rootkits that usually skip the tedious procedure of acquiring and releasing the lock, since that would lead to complications.

Also, such rootkits might not be aware of the addresses of mutex lock and unlock functions, and even then, scalability becomes an issue as it gets cumbersome to keep track of all mappings between locks and structure fields. We now use this distinction to write a policy on the data structure itself. Our policy would allow access to the specified fields only if the appropriate lock is held. This can be detected by the pseudocode in Figure 5.

While one might argue that using call-stacks would be more efficient in detecting calls to these mutex functions, many times such functions are inlined[8] to improve efficiency; invoking them does not modify the kernel's call stack.

Policy MemReadOnly.

To ensure that pages mapped read-only remain unmodified, it is sufficient to ensure that the "read-only" permission in the page-table is set for that page. However this mechanism is not completely secure since any process with $kmem$ access can modify the page table entries to give write permissions to any user or kernel page. To protect the page-tables against such modifications, we need a process operating at a higher privilege than the kernel itself. Although VMMs achieve this, it involves the overhead of tracking each memory access and checking the address and operation performed against the policy. By enforcing policies at the hardware level, we can ensure both safety and performance guarantees.

To illustrate an example, the Linux kernel supports cryptographically signed LKMs, which are currently insecure since the signature verification happens in the binary `insmod`. Imposing read-only permissions on the text region of `insmod` would not prevent rootkits from modifying its page table, mapping the page containing the verification code "read-write", and then overwriting the memory to always pass verification. As such, enforcing the policy in Figure 6 from hardware would help prevent such attacks.

Policy ExecWrite.

A policy that detects access patterns similar to those used by OllyDbg to detect packer code, i.e. detect instruction fetches from addresses which were previously written into. While OllyDbg does

[8] `http://lxr.linux.no/linux+v2.6.24.1/include/asm-x86/semaphore_32.h`

Table 2: ***Debugging Policy Language Primitives.***

Primitive	Explanation
MEM[a..b,c..d]	Identifies memory addresses ranging from address `a` to `b`, `c` to `d`,...
REG[reg]	Identifies the contents of register `reg`
FUNC[f]	Identifies the address range (start .. end) of the function `f`
ALERT()	Raises an alert to the user/kernel
w(var)	True if any of the addresses in the range indicated by `var` is being written.
r(var)	True if any of the addresses in the range indicated by `var` is being read.
x(var)	True if any of the instructions in the range indicated by `var` is being executed.
var.wc	Contains the last time when the address(es) indicated by `var` were written into.
curtime	Time in nanoseconds or jiffies since the system was booted.
within(var1, var2)	Returns true if the value of `var1` is contained in the address range identified by `var2`.
lock_status(lck)	Returns `LOCK_HELD` or `LOCK_FREE`

```
define var bar = MEM[ 0xc3541ab0 ..
                      0xc4500130 ];
rule readonlyMem
{
  if (w(bar))
    ALERT();
}
```

Figure 6: ***Policy MemReadOnly.***

```
define var packedMem = MEM[ 0xc0131040 ..
                            0xc0132000 ];
rule detectUnpacker
{

  if( x(packedMem) &&
  ( (curtime - packedMem.wtime) <= 1s) )
  {
    ALERT();
  }
}
```

Figure 7: ***Policy ExecWrite.***

this through ITLB flushing, the FPGA provides a simpler and faster solution to the same problem, as shown in Figure 7.

Policy LinkProtect.

Protecting tables of links is also of concern, as shown in Figure 8.

6. DISCUSSION

Alternatives to using an FPGA include using page protection and instruction emulation, but it remains unclear whether the cost of emulating each instruction (even with translation caches) and the cost of page faults (pages are large, and faults are common) is actually cheaper than having an inline FPGA that operates at hardware speed in observing memory events. In any event, as we highlight in Section 2, most "hackish" solutions for policy-driven analysis already rely on unconventional use of page protection traps in one form or another.

We also note that some of the performance problems of current debuggers stem from implementation issues. For example, some breakpoint processing to evaluate conditions, expressions, or variable values does not necessarily require a context switch. The ERESI[9] suite of tools does enable in–process evaluation, but at the cost of introducing a whole variety of hacks (hackish enough to make two Phracks in a row [15, 22]).

[9] `http://www.eresi-project.org`

```
define var table = MEM[ 0x1000 ..
                        0x3200 ];
define var fLink = FUNC[ldd_link];
define var fUnlink = FUNC[ldd_unlink];

rule guardLinkTable
{
  if( w(table) )
  {
    if(table.wc == 0 &&
       !within(REG[eip], fLink) )
      ALERT();

    if(table.wc == 1 &&
       !within(REG[eip], fUnlink) )
      ALERT();
  }
}
```

Figure 8: ***Policy LinkProtect.***

6.1 VMM and TCB "Bloat"

In this paper, we have emphasized the link between a bloated VMM and what is generally thought to be a dictum: larger TCBs containing more code and complexity tend to have lower levels of security assurance. The extent to which this folk wisdom is true for any particular system is difficult to gauge. One reviewer pointed out that the instrumentation code for trapping low level events forms a relatively small part of a modern VMM; removing such code to an FPGA might seem like a small victory in terms of reducing the TCB size.

From our perspective, the actual trapping mechanism is something of value for both resource emulation and security policy enforcement. While the current code in the VMM responsible for trapping events of interest may be a relatively small part of the code base (as opposed to device drivers, etc.), it hardly allows the kind of flexibility that better policy enforcement or debugging requires (we give examples of this need in Section 2).

Further, our position is not simply to remove basic trapping code from the VMM, but rather the handling of *security–related* traps. Bellovin points out [6] that the real issue that virtualization has yet to solve is the need to provide customizable, expressive control of I/O between isolated VMs. In an abstract sense, the trapping present in a VMM supports such isolation by managing access to the MMU; in reality, the VMM management interface and various performance hacks allowing neighboring guest VMs to communicate violates this isolation.

Our proposal introduces an efficient hardware–based facility for trapping on memory events: such a facility may be of use for the

VMM in its need to supervise the MMU to enforce separation between guest VMs. Our point is not so much about removing the burden from the VMM — it is about independent progress and the growing requirements of debugging and security (authentication, authorization, self–healing, attestation) policy making this burden so great that a VMM would no longer be a good place for it.

Accordingly, we assert what may seem to be a rather controversial opinion in a forum that focuses on the marriage between virtualization and systems security. Our opinion is that virtualization is not the answer for better security policies. Instead, we assert that something that underlies virtualization — a hardware trap system with its own logic — is that answer. As a result, this paper mainly focuses on the properties of this system that help support faster instrumentation execution (instead of traditional virtualization topics like isolation).

6.2 Using an FPGA

We focus on the addition of an FPGA for several reasons, even though it represents a significant deviation from existing architectures. An FPGA is programmable and provides hardware speed monitoring. We assert that such properties are exactly what is needed to tackle the problems of malware analysis, reverse engineering, and security policy enforcement that we highlight earlier in the paper. Despite the fact that combinations of slight architecture extensions, binary translation, and microcode tweaks seem like safer, more conservative options (and hence more likely to be adopted by a hardware industry anxious for security–related chip improvements that will not seriously disrupt their production pipeline), we believe that there is significant value in the research community examining a paradigm shift in how such event handling might be accomplished. An FPGA seems like the most feasible method of producing an academic research prototype in the near future without having to undertake chip fabrication. Finally, an FPGA prototype is within our capabilities, unlike more conservative architectural changes that would have required leveraging our existing industry relationships to undertake an effort which processor designers rarely rush into. Our aim is to capture the interest of that community by explaining what one can do with a better trap system and convince them that designing such a system for production is worthwhile.

7. RELATED WORK

Interesting alternative approaches to improving the performance of security-related checks exist; for example, Nightengale *et al.* [17] describe how such checks can be performed on spare cores of a multicore system, and Oplinger and Lam [18] discuss the use of Thread-Level Speculation (TLS) for achieving much the same effect. Note that these approaches can benefit from more efficient trapping to their measurement or security check execution code; even if such code is executed on another core, this code still needs to be invoked by some trapping mechanism, and the that code still requires efficient access to security–related program state.

7.1 Program Instrumentation

The recent work of King *et al.* [12] provides an existence proof that the types of designs we envision are feasible. This work demonstrates how software can trap into the FPGA logic. In turn, the FPGA will return control to the software in such a way that the program can transparently continue execution when an event of interest is judged by the FPGA logic to not have occurred. The interplay between the FPGA-based logic and the regular processor instructions (software or firmware) generalizes that of the virtual memory translation and interrupt handling mechanisms, which were, up to now, the only such examples of fast hard-coded and software logic fitting together into a single execution stream.

Although King *et al.* discuss their implementation only in the context of malicious hardware undermining the security properties of the platform, we note that the ability of an FPGA-based mechanism to mesh with the code transparently and efficiently opens much broader prospects. Namely, it allows for creating trap systems of unprecedented expressive power, which will translate to both new and more efficient security policy enforcement and much richer debugging and reverse engineering primitives.

Systems like Valgrind [16] and Pin [14] have recently emerged that enable a programmer or software tester to interweave complex programmatic instrumentation at runtime into an existing software system. These systems use dynamic binary rewriting and do not require access to the source code. Similar environments include the Rio architecture [8] and Dyninst [9].

Solaris zones[10] from Sun Microsystems is a weak implementation of full-fledged virtualization that provides application-level and not OS-level virtualization. It is similar in concept to FreeBSD Jails and provides a secure isolated environment for applications within each zone. Solaris zones can be of two flavors: sparse or whole root. Sparse zones share common system files with other zones through a loopback system interface mounted read-only within each zone, while whole root zones provide an individual private copy of the necessary system directories. Both kinds of zones can co-exist within a single system. Solaris also comes packaged with resource management facilities that provide administrative interfaces for allocating system resources such as processor, memory etc to each individual zone. Zones and Jails are customized and tightly integrated with the operating system to provide certain virtualization functionalities such as isolation and abstraction for applications, but they should not be confused with pure virtualizing systems such as Xen and Vmware.

Program shepherding [13] focuses on ensuring that control flow transfers of a process remain within the bounds of some policy. For example, the technique uses the Rio [8] system to ensure that code in library routines is only accessed via the entry point of the particular library function. Control Flow Integrity (CFI) [1] is a similar idea in which a program's static control flow graph acts like a policy for the runtime behavior of the system.

7.2 Other Work

The Spyder project at Purdue introduced several seminal techniques in debugging, among them program slicing and backtracking, to help provide an automated debugging tool [2]. MemSherlock [21] proposes automated debugging of memory corruption vulnerabilities. It uses combination of source code analysis and tainted data flow analysis to discover the vulnerability path through a program and associate it with source–level statements. MemSherlock is relevant to our work because it relies on creating a "write set": the set of all possible statements in a program that write to a particular memory location.

In what we believe to be a seminal call to action for the operating systems research community, Roscoe *et al.* [20] argue that current OS research utilizing hypervisors should move away from endlessly refining traditional approaches aimed at Unix/Windows ABI model compatibility. In essence, the hypervisor presents a useful backwards-compatible interface, and Roscoe's paper argues that the problems that we currently tackle at the VMM level (such as inter-VM communication, resource sharing among VMs, VM isolation) have all been solved at the OS-level, and adding more

[10] `http://www.sun.com/bigadmin/features/articles/solaris_zones.jsp`

functionality in the VMM is largely wasted effort. Furthermore, adding needless functionality to the VMM simply increases the size of the trusted computing base (TCB), and hence it becomes harder to prove VMM correctness or security properties. The size and nature of this complexity are discussed in an article by Karger and Safford [11]; they make many of the same points (and ably illustrate the various interactions) we do with respect to the complexity of VMM I/O systems.

8. CONCLUSION

Current hardware provides basic and limited trapping and inspection facilities on which entire supervision, debugging, and analysis infrastructures must be built from the ground up. We believe that this situation contributes to the incorporation of security functionality into software virtualization components. We argue that this is exactly the wrong place to put it.

A flexible and robust program supervision and debugging capability depends on having an efficient and expressive trap mechanism. We propose that such a trap system can be based on an FPGA placed at the interface between the CPU and the memory cache. We believe it will allow for checking complex conditions (and trapping, should these conditions be satisfied) at almost the native speed of the computing platform.

Acknowledgments

We are grateful to many fellow researchers and our reviewers for providing feedback and suggestions. In particular, the first author would like to thank Hugo Fortier and the RECON conference[11] presenters for the many inspirations, and Daniel Bilar and Victor Grinberg for useful discussions of the ideas that lead to this paper. This work was supported in part by the National Science Foundation, under grant CNS-0524695, the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, and the Institute for Security Technology Studies, under Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. The views and conclusions do not necessarily represent those of the sponsors.

9. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] H. Agrawal. Towards Automatic Debugging of Computer Programs, August 1991.

[3] T. Beauchamp and D. Weston. Re:Trace – Applied Reverse Engineering on OS X.

[4] T. Beauchamp and D. Weston. Re:Trace – Applied Reverse Engineering on OS X. RECON 2008, 2008. Montreal, Quebec.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, April 2005.

[6] S. M. Bellovin. Virtual Machines, Virtual Security. *Communications of the ACM*, 49(10), October 2006.

[7] F. Brooks. *The Mythical Man Month*. Addison-Wesley Professional, 2 edition, 1995.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[9] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[10] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[11] P. A. Karger and D. R. Safford. Security and Performance Trade-Offs in I/O Operations for Virtual Machine Monitors. In *IBM Research Technical Report RC24500 (W0802-069)*, February 2008.

[12] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the* 1st *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[13] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11th *USENIX Security Symposium*, August 2002.

[14] 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 *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[15] mayhem. The Cerberus ELF Interface. *Phrack*, 2003.

[16] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, June 2007.

[17] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[18] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[19] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. 2005.

[20] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and Virtue. In *Proceedings of the* 11th *Workshop on Hot Topics in Operating Systems (HOTOS XI)*, May 2007.

[21] E. C. Sezer, P. Ning, C. Kil, and J. Xu. MemSherlock: an Automated Debugger for Unknown Memory Corruption Vulnerabilities. In *Proceedings of the* 14th *ACM conference on Computer and communications security (CCS 2007)*, pages 562–572, New York, NY, USA, 2007. ACM.

[22] T. E. shell crew. Embedded ELF Debugging : the middle head of Cerberus. *Phrack*, 2003.

[23] R. M. Stallman, R. H. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 2003.

[24] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the* 15th *Annual Network and Distributed System Security Symposium (NDSS)*, February 2008.

[11] `http://recon.cx/2008/archive.html`