Building a Better Mousetrap: Scriptable and Semantically Expressive Hardware-assisted Memory Trapping *

Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith †
Department of Computer Science
Dartmouth College
Dartmouth Computer Science Technical Report TR2008-627

October 22, 2008

Abstract

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. Improvements in debugging technology and support can have a positive impact in these areas as well as the more traditional use of resolving issues with system features or performance. Despite the increasing complexity and burgeoning feature set of modern debuggers and IDEs, however, debugging software systems remains a manual art: an iterative, interactive, and time–consuming process.

This process imposes a performance penalty on the user: a penalty that is primarily derived from the limitations of trapping support in modern hardware. Hardware and memory systems designs rarely contain efficient interception and inspection facilities to aid debugging efforts; rather, they are meant to run program code as quickly as possible. Debugging frameworks often perch on an overtaxed, ungainly hardware kludge. Support for scriptable, policy–driven debugging remains limited.

As a result, debugging occurs through an *ad hoc* procedure driven by repeated conjecture and test cases rather than an automated exploration of a constrained solution space. We advocate the design of better hardware support to enable efficient, policy–driven debugging. 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.

1 Introduction

The rising complexity of modern programs requires more efficient debugging and reverse engineering support in both software and hardware. This observation applies especially well to the current generation of malware, which employs additional obfuscation to resist such analysis. For smaller programs and less

*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.

†sergey,locasto,ashwinr,sws@cs.dartmouth.edu

complex program analysis, these tasks could be performed with reasonable efficiency by using ordinary breakpoint or single-step routines in combination with manual checking for conditions of interest. Such simple techniques no longer suffice; we argue that they restrict debugging to a manual, time–consuming, and *offline* process.

In our personal experience debugging systems, the number of "false positive" breakpoint events that a programmer or a reverser would need to examine can vary from inordinately large to completely unmanageable. We postulate that this situation exists either due to a natural increase in the complexity of software or in the case of malware, through the use of various obfuscation techniques. Accordingly, it seems that we need to increase the expressive power of descriptions of breakpoint (*i.e.,* fault or trap) conditions. This increase in expressive power must be matched by 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.

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 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 encodes a debugging plan rather than undertakes a manual sequence of specific memory manipulations and examinations.

1.1 Signs of a Paradigm Shift

Not surprisingly, a number of *adhoc* solutions recently arose to help fill this growing need (a clear sign that a paradigm shift in this area is overdue). 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[1] 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. We discuss OllyBone's operation in more detail in Section 2.3.

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.

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

- Some frameworks allow access to the debugged process's OS environment, including its process information block.

1.2 Better Trap Semantics

We next highlight the relationship between trap semantics, the implementation of a trap system, debugging, and systems security. As a way to illustrate part of this relationship, we also discuss some new exploitation techniques. While several solutions have been proposed to address these exploit techniques, these solutions are typically threat-specific. We argue that a common underlying mechanism general enough to express most of these solutions is possible and necessary.

1.2.1 Traps and Security

At first glance, the connection between a trap system for a particular platform and the security properties of that platform may not seem obvious. They are, however, directly and intimately related.

We believe that it is natural to formulate security properties as those preserved across normal transitions in the system's state space, given that the system starts in a trustworthy state. Abnormal transitions should cause traps, after which the system's state may no longer be considered trustworthy or "secure." Accordingly, much of a security system's essential functionality is implemented inside trap handlers.

For security policies, *events* that correspond to the system's transitions between trusted states play a similar central role in the design and implementation of the policy mechanism. Namely, the policy mechanism is charged with allowing only "safe" transitions that preserve the desired security properties. While such mechanisms can be implemented purely in software, in practice they rely on hardware-supported traps whenever possible, to let application code execute at full speed between mediated events, and to provide additional assurance of separation between the more and less trusted parts of the system.

In practice, therefore, traps form a core mechanism upon which to implement security policy interpreters. As such, they directly or indirectly affect all aspects of the latter. In our opinion, the details of the trap system *defacto* shapes the capabilities and performance of the policy system.

1.2.2 Traps and Debugging

Informally speaking, the process of debugging an application has much in common with the process of enforcing a policy. Instead of "trustworthiness", a bug-hunter tries to ensure that the system behaves according to her mental model of what the code is supposed to do and catch the moment when it begins to deviate[2] from that model. That moment — more precisely, that event — is assumed to be the manifestation of the hunted bug.

The programmer's mental model of a program's intended behavior can be more complicated than that of its security properties. As a result, the set of events that the developer needs to monitor to debug the program can be harder to describe than the set of security-related events that require mediation. As a result, debugging likely needs much more flexible support than security policy enforcement. Yet, we note that the available hardware support on commodity platforms remains in its infancy (to put it somewhat bluntly, in an equivalent of the Stone Age).

No other area of computer science or computer engineering exhibits such an amazing paucity of efficient tools or the formal means to describe itself. Efficient debugging is somehow still perceived as

[2]We note that behaving as expected is one of the definitions of "trust." Thus a "bug" in the program, from the programmer's point of view, is exactly what breaks the trust.

an arcane skill. Intuition remains programmers' best bet in producing a *usable* machine-understandable description of intended behaviors. In contrast, having current programming languages specialists rely mostly on intuition to produce usable compilers is hardly imaginable. Needless to say, this state of affairs translates to major deficiencies in the trustworthiness of the majority of software.

We believe that the continued lack of a flexible means to describe events relevant to debugging is caused by the absence of a more comprehensive set of hardware primitives. Although software systems can provide this rich set of primitives (Pin [13] is a good example), real production software constantly pushes the limits of computing speed, making the use of software–level debugging unattractive for such systems. 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.

1.3 Problem Description

The key issue here is that the set of trapping conditions that can be described with hardware primitives is severely limited. On the `x86` architecture, traps can be triggered on reads, writes, or instruction fetches from fixed memory locations, 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 is rather *adhoc* and 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 both to store high–level state in hardware page table control bits and to manipulate existing trap handling mechanisms strongly suggest 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. Unfortunately, the process of debugging does not seem to scale with the size of new software systems or the rates of exploitable bugs. Although Agrawal [2] suggests an automated approach to debugging (using 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 paradigm for debugging support to both increase the expressive power of debugging as well as reduce the performance cost of intercepting execution. We begin by examining several motivating examples and some case studies of 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).

In order to understand what types of primitives such a system should supply, we analyze several recent software systems that provide "smarter" (*i.e.*, script and condition-driven), more granular memory trapping systems. We propose new directions in hardware design that can support these primitives. We explore the possibility of using the proposed hardware for debugging and analysis of software as well as a new class of security policies that are better suited for expressing certain integrity-related security goals.

2 Motivation

In this section, we consider several motivating examples that, although drawn from different areas, highlight the need for a faster and more expressive trap system. As Figure 1 shows, we advocate an approach where 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.

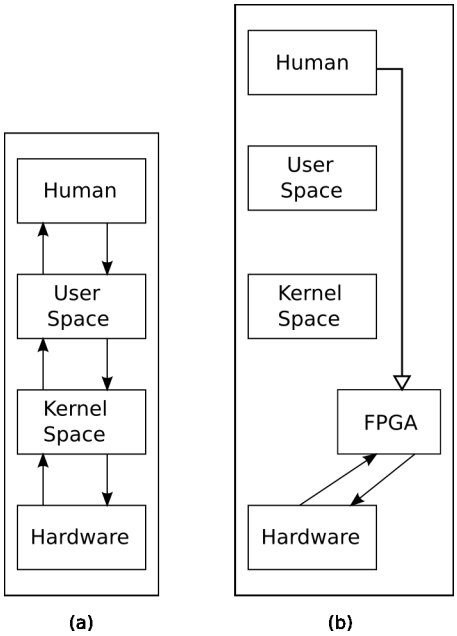

Figure 1: *Transforming the Memory Trap Service Path*. Currently, as shown in (a), servicing an interrupt for debugging purposes involves traversing the system stack twice, which incurs both human processing time as well as two context switches (between the traced and tracing process). We propose to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

2.1 Debugger Watchpoints

Debugger watchpoints [19], 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 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[3].

2.2 Costly State

Consider the following example of using a watchpoint. Suppose we want to break on the 200^{th} 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[4]. 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 in a

[3] *"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 [4]*

[4] 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 unique hardware features is a symptom of exactly the problem we try to address.

security policy regarding information flow to be efficiently enforced[5]. 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. In order to run, the malware is extracted by an "unpacker" or decoder preamble. Catching the moment 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 CPU emulator such as Bochs[6] or QEMU [3] to analyze the dynamic 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 slow down execution of the code in a virtual machine based environment 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 attempt to detect that it is being run in a VM environment (for example, by executing an instruction that is not virtualizable — several such instructions exist on `x86`) and abort execution.

Therefore, an analyst greatly benefits from a trapping mechanism that would allow the unpacker to execute at normal or close to normal hardware speed, yet trap close to 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.

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 because:

- 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.
- 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.

2.4 Malware Anti-tracing Tricks

In the game of malware authors vs. malware analysts, malware protection aims at wasting as much of the analysts' time as possible. Besides many tricks meant to interfere with `x86` hardware tracing support and various OS debugger support features[7], a simple but powerful alternative to defeat an analyst tracing the

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

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

[7] E.g., `http://www.securityfocus.com/infocus/1893`, many others in [21].

program for a particular behavior is to prepend it with a harmless long preamble. In the situation when the tracing options available to the analyst are all-or-nothing, such a preamble is sure to waste a lot of time executing when running at slower-than-hardware speeds. Analysts thus require more complex trapping primitives that allow them to describe conditions of interest in the malware's lifecycle.

2.5 Return-to-ELF-object

Controlling malware's access to objects in memory can be difficult: return-to-libc and return-to-PLT attacks are examples of malware or exploits using the system's own services (such as the linker's symbol search-and-load by name routine) to locate the target object and complete an attack.

We would like to efficiently monitor access to memory objects; since the linker and loader retain significant knowledge about many objects, we should be able to convey this knowledge to the hardware. In effect, this capability is similar to hardware–based runtime type enforcement. Policies controlling this type of access would examine memory events such as instructions that access particular portions of an ELF object. We see this capability as a way to generalize "RAM firewalls"[8] that operate on the basis of validating control flow transfers at the machine level. RAM firewalls seem analogous to packet filters, whereas the memory event architecture we propose is akin to an application–level firewall that is aware of the semantics of the events flowing through it.

2.6 Information Flow Control

Information flow control — tainted data flow analysis [16, 20, 9, 8] in particular — can benefit from hardware that supports the tainting and tagging of *objects* rather than memory addresses. At present, these mechanisms (with the exception of Suh *et al.* [20] and Nakka *et al.* [14], who propose hardware support) impose a relatively hefty performance penalty derived from the need to intercept each machine instruction in software. Also, most "tainting" systems are designed to follow a narrow set of threat–centric goals. In fact, tracing arbitrary tags is of more general interest and would require a general and efficient trapping mechanism.

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 [13], DTrace [7], and Kprobes and SystemTap [17]; we have covered some informal "hacker debugging" techniques such as those used in OllyBone, ProcessStalker, and RastaDebug in Section 2. 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

[8] Determina (now a part of VMWare) `http://www.determina.com`, commercialized the program shepherding research [11], calling it a "memory firewall."

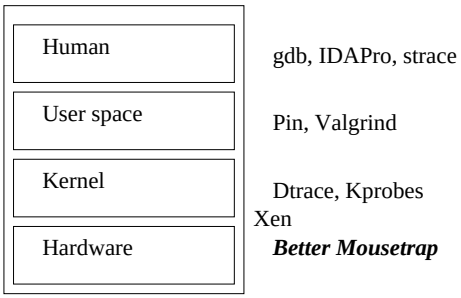

Figure 2: *Handler Hierarchy.* Forcing humans to manually evaluate a condition results in a slow, iterative, and offline debugging procedure. Although software systems can help alleviate the demand for more flexible debugging and program analysis procedures, this level of programability begins to rapidly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

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.

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 [13] 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 stated at the end of Section 1.4, where we seek to reduce the cost of servicing a trapped event.

3.2 Pin

Both Valgrind [15] and Pin [13] provide customizable runtime interception of program execution (*i.e.*, something more feature–rich than `ptrace(2)`). Pin is an IA-32 binary rewriting framework. 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. Pin tools are user–level code written in C++ and contain two basic types of functions: (1) instrumentation functions and (2) analysis functions.

When a Pin tool starts up, it uses the Pin API to register instrumentation functions that serve as callbacks for when Pin recognizes an event or portion of program execution that the tool is interested in (*e.g.*, instruction execution, basic block entrance or exit, *etc.*). The instrumentation functions then employ the Pin API to insert calls to the tool's analysis functions. Analysis functions are invoked every time Pin encounters the corresponding code or event; in contrast, instrumentation functions are executed only once.

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 [7], 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.

3.4 Kprobes

Kprobes is a dynamic instrumentation framework for Linux that allows the user to insert arbitrary breakpoints at the instruction level 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. This property is critical for system efficiency and one of the core design principles that we believe any tracing system should strive to accomplish.

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.

In 2005, the SystemTap project [17] by Redhat, IBM, and Intel attempted to address the limitations of Kprobes by building a user–level scripting language on top of it for tracing the Linux kernel. The semantics of this scripting language is similar to *AWK* and the language contains some features from C. One major benefit of SystemTap is that the scripting language removes the programmer from directly manipulating code that is inserted into the kernel. Hence, we can think of SytemTap as providing similar functionalities to DTrace, but on Linux.

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(see Figure 3), we hope to satisfy the twin demands of more flexible analysis and better trap handling performance.

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

Table 1: *Features of Instrumentation Tools.* Here we compare the feature set of existing approaches to debugging with our Better Mousetrap (BMT) 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.

4.1 MMU Modifications

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 necessary actions (such as `ALLOW`, `DENY` etc).

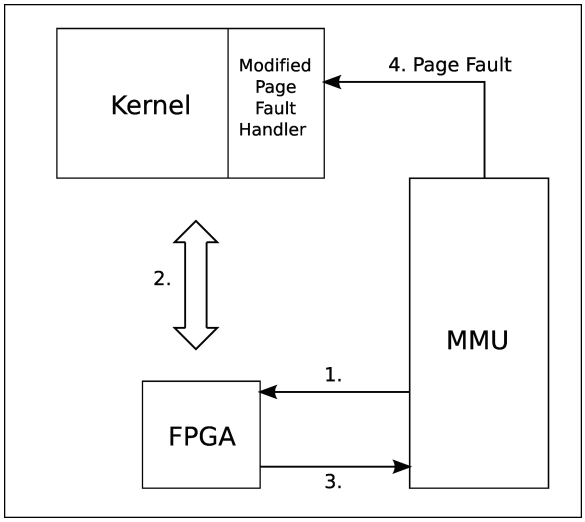

Figure 3: *A Block Diagram of the Better Mousetrap.*

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

1. When a process accesses 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 satisfies 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 occurred, and takes the necessary action, which might include logging the access, killing the process, panicking the system *etc.*, as dictated by the policy itself.

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

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 of `var2`.
lock_status(lck)	Returns `LOCK_HELD` or `LOCK_FREE`

Table 2: *Debugging Policy Language Primitives.*

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 built-in features, keywords, and predicates. This current, unfinished form of the policy language contains the primitives listed in Table 2. 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 to access the memory block. For example, a particular instance of this policy might allow only the kernel functions `list_add` and `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.

Such a policy can be written as:

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

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

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 unnecessarily 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 following pseudocode:

```
// 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();
}
```

While one might argue that using call-stacks would be more efficient in detecting calls to these mutex functions, many times it is not possible to do so since such functions are inlined[9] to improve efficiency, and hence 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. This is achievable from VMM systems, but 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 unsecure 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 the verification check. As such, enforcing this policy from hardware would assist in preventing such attacks.

```
define var bar = MEM[ 0xc3541ab0 ..
                      0xc4500130 ];

rule readonlyMem
{
  if (w(bar))
    ALERT();
}
```

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 this through ITLB flushing, the FPGA provides a simpler and faster solution to the same problem. Here we choose to detect instruction fetches that occur within one second of memory writes at that address. We could easily alter this time-interval to better capture the realities of existing malware.

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

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

[9] `http://lxr.linux.no/linux+v2.6.24.1/include/asm-x86/semaphore\_32.h`

Policy LinkProtect Protecting tables of links is also of concern. A policy to protect these types of data follows:

```
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();
  }
}
```

6 Related Work

The activities of debugging and testing software applications form a large part of the software engineering process. Research on increasing the power, efficiency, and ease of use of these mechanisms draws on a number of different fields including software engineering and software reliability. A closely related NSPW paper is the concept of policy–constrained speculative execution [12], in which branches of execution proceeding through the CPU pipeline are controlled by some higher–level policy (as opposed to simply evaluating the branch conditional).

The Spyder project at Purdue introduced several seminal techniques, among them program slicing and backtracking, to help provide an automated debugging tool [2]. MemSherlock [18] 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. They construct a write set via static analysis. MemSherlock employs TaintCheck [16] to extract a vulnerability path.

Program Instrumentation

The recent work of King *et al.* [10] 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 [15] and Pin [13] have recently emerged that enable a programmer or software tester to interweave complex programmatic instrumentation (written in C++ or a C dialect) 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 [5] and Dyninst [6].

Program shepherding [11] focuses on ensuring that control flow transfers of a process remain within the bounds of some policy. For example, the technique uses the Rio [5] 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. Data Flow Integrity, (DFI) [8] ensures that no tainted external data corrupts important control flow structures such as the program counter or instruction register.

Finally, we note that our advocacy of an "automated search of a constrained solution space" is related to model checking. Such activity differs from model checking in two fundamental ways. First, model checking is done before software is deployed (or a program run); we advocate automating similar kinds of checks on software that is running. Second, model checking tries to statically match the behavior of the program with a specification of good behavior; we advocate an approach that dynamically diagnoses *failures* — bad or aberrant behavior. Although design teams can certainly create models of "failure modes," programmers may find it difficult to simultaneously envision all the ways a program could be *made* to fail while engaged in the process of specifying how it should work.

7 Conclusion

Current hardware provides basic and limited inspection facilities on which entire debugging infrastructures must be built from the ground up. We wish to drastically increase the performance of debugging by hiding the performance impact of trapping details from the user while exposing a more flexible debugging policy language.

We are advocating a shift in which debugging changes from an offline process of test and manual analysis to a more online, automated debugging procedure of live production programs. In essence, where previous efforts to provide a "memory firewall" have worked at a level analogous to early "packet filtering" firewalls, we seek to build a memory firewall that performs at the "application" level. This sort of capability depends on having an efficient, expressive, and useful trap mechanism. These trap systems, backed by the logic loaded into an FPGA at the MMU (just as the FPGA in Sam King's recent work in LEET 2008) sits 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.

Software developers and vendors largely rely on proactive or reactive debugging activity to identify and correct security issues and flaws in their software. In addition, reverse engineering malcode samples

relies on the same basic set of techniques and hardware features that debuggers do. Both of these activities — particularly the diagnosis step — remain manual activities because the hardware does not provide comprehensive support for an automated, planned approach to debugging.

Hardware is designed to execute code, not support the analysis of it (except in an abstract sense, by running software programs specifically designed to analyze symptoms of faults and errors). While systems can easily run large numbers of test cases and identify potential issues, the system must stop there and create a report for a human to follow up on and actually diagnose the root cause of the problem. As a result, developers or reverse engineers must spend time processing "false positive" events and manually evaluating predicates on the state of the program in question. This paradigm limits debugging to a mostly *offline* process. If we are to achieve the vision of autonomic computing, we require a method of *online* debugging. Of course, fundamental limits of computability dictate that systems cannot completely diagnose themselves, but we hope to make possible the automation of a diagnosis plan that is far more sophisticated than simply running test cases.

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*. PhD thesis, August 1991.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[9] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of EuroSys 2006*, pages 29–41, April 2006.

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

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

[12] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Speculative Virtual Verification: Policy-Constrained Speculative Execution. In *Proceedings of the* 14^{th} *New Security Paradigms Workshop (NSPW)*, pages 119–124, September 2005.

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

[14] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 585, Washington, DC, USA, 2004. IEEE Computer Society.

[15] 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.

[16] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[17] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating System Problems Using Dynamic Instrumentation. In *Ottawa Linux Symposium (OLS)*, 2005.

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

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

[20] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[21] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.