Memory Safety for Low-Level Software/Hardware Interactions

John Criswell
University of Illinois
criswell@uiuc.edu

Nicolas Geoffray
Université Pierre et Marie Curie
INRIA/Regal
nicolas.geoffray@lip6.fr

Vikram Adve
University of Illinois
vadve@uiuc.edu

Abstract

Systems that enforce memory safety for today's operating system kernels and other system software do not account for the behavior of low-level software/hardware interactions such as memory-mapped I/O, MMU configuration, and context switching. Bugs in such low-level interactions can lead to violations of the memory safety guarantees provided by a safe execution environment and can lead to exploitable vulnerabilities in system software. In this work, we present a set of program analysis and run-time instrumentation techniques that ensure that errors in these low-level operations do not violate the assumptions made by a safety checking system. Our design introduces a small set of abstractions and interfaces for manipulating processor state, kernel stacks, memory mapped I/O objects, MMU mappings, and self modifying code to achieve this goal, without moving resource allocation and management decisions out of the kernel. We have added these techniques to a compiler-based virtual machine called Secure Virtual Architecture (SVA), to which the standard Linux kernel has been ported previously. Our design changes to SVA required only an additional 100 lines of code to be changed in this kernel. Our experimental results show that our techniques prevent reported memory safety violations due to low-level Linux operations and that *these violations are not prevented by SVA without our techniques*. Moreover, the new techniques in this paper introduce very little overhead over and above the existing overheads of SVA. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

1 Introduction

Most modern system software, including commodity operating systems and virtual machine monitors, are vulnerable to a wide range of security attacks because they are written in unsafe languages like C and C++. In fact, there has been an increase in recent years of attack methods against the operating system (OS) kernel. There are reported vulnerabilities for nearly all commodity OS kernels (e.g., [2, 28, 43]). One recent project went so far as to present one OS kernel bug every day for a month for several different open source and commercial kernels [26] (several of these bugs are exploitable vulnerabilities). Preventing these kinds of attacks *requires protecting the core kernel and not just device drivers*: many of the vulnerabilities are in core kernel components [19, 40, 41, 43, 46].

To counter these threats, there is a growing body of work on using language and compiler techniques to enforce *memory safety* (defined in Section 2) for OS code. These include new OS designs based on safe languages [4, 18, 22, 33], compiler techniques to enforce memory safety for commodity OSs in unsafe languages [10], and instrumentation techniques to isolate a kernel from extensions such as device drivers [45, 47, 51]. We use the term "*safe execution environment*" (again defined in Section 2) to refer to the guarantees provided by a system that enforces memory safety for operating system code. Singularity, SPIN, JX, JavaOS, SafeDrive, and SVA are examples of systems that enforce a safe execution environment.

Unfortunately, all these memory safety techniques (even implementations of safe programming languages) make assumptions that are routinely violated by low-level interactions between an OS kernel and hardware. Such assumptions include a static, one-to-one mapping between virtual and physical memory, an idealized processor whose state is modified only via visible program instructions, I/O operations that cannot overwrite standard memory objects except input I/O targets, and a protected stack modifiable only via load/store operations to local variables. For example, when performing type checking on a method, a safe language like Java or Modula-3 or compiler techniques like those in SVA assume that pointer values are only defined via visible program operations. In a kernel, however, a *buggy* kernel operation might overwrite program state while it is off-processor and that state might later be swapped in between the definition and the use of the pointer value, a *buggy* MMU mapping might remap the underlying physical memory to a different virtual page holding data of a different type, or a *buggy* I/O operation might bring corrupt pointer values into memory.

In fact, as described in Section 7.1, we have injected bugs into the Linux kernel ported to SVA that are capable of *disabling the safety checks that prevented 3 of the 4 exploits* in the experiments reported in the original SVA work [10]: the bugs modify the metadata used to track array bounds and thus allow buffer overruns to go undetected. Similar vulnerabilities can be introduced with other bugs in low-level operations. For example, there are reported MMU bugs [3, 39, 42] in previous versions of the Linux kernel that are logical errors in the MMU configuration and could lead to kernel exploits.

A particularly nasty and very recent example is an insidious error in the Linux 2.6 kernel (not a device driver) that led to severe (and sometimes permanent) corruption of the e1000e network card [9]. The kernel was overwriting I/O device memory with the x86 `cmpxchg` instruction, which led to corrupting the hardware. This bug was caused by a write through a dangling pointer to I/O device memory. This bug took weeks of debugging by multiple teams to isolate. A strong memory safety system should prevent or constrain such behavior, either of which would have prevented the bug.

All these problems can, in theory, be prevented by moving some of the kernel-hardware interactions into a virtual machine (VM) and providing a high-level interface for the OS to invoke those operations safely. If an OS is *co-designed* with a virtual machine implementing the underlying language, e.g., as in JX [18], then eliminating such operations from the kernel could be feasible. For commodity operating systems such as Linux, Mac OS X, and Windows, however, reorganizing the OS in such a way may be difficult or impossible, requiring, at a minimum, substantial changes to the OS design. For example, in the case of SVA, moving kernel-hardware interactions into the SVA VM would require extensive changes to any commodity system ported to SVA.

Virtual machine monitors (VMMs) such as VMWare or Xen [16] do not solve this problem. They provide sufficiently strong guarantees to enforce isolation and fair resource sharing between different OS instances (i.e., different “domains”) but do not enforce memory safety *within* a single instance of an OS. For example, a VMM prevents one OS instance from modifying memory mappings for a different instance but does not protect an OS instance from a bug that maps multiple pages of its own to the same physical page, thus violating necessary assumptions used to enforce memory safety. In fact, a VMM would not solve any of the reported real-world problems listed above.

In this paper, we present a set of novel techniques to prevent low-level kernel-hardware interactions from violating memory safety in an OS executing in a safe execution environment. There are two key aspects to our approach: (1) we define carefully a set of abstractions (an API) between the kernel and the hardware that enables a lightweight run-time checker to protect hardware resources and their behaviors; and (2) we leverage the existing safety checking mechanisms of the safe execution environment to *optimize* the extra checks that are needed for this monitoring. Some examples of the key resources that are protected by our API include processor state in CPU registers; processor state saved in memory on context-switches, interrupts, or system calls; kernel stacks; memory-mapped I/O locations; and MMU configurations. Our design also permits limited versions of self-modifying code that should suffice for most kernel uses of the feature. Most importantly, our design provides these assurances while leaving essentially all the *logical control* over hardware behavior in the hands of the kernel, i.e., no policy decisions or complex mechanisms are taken out of the kernel. Although we focus on preserving memory safety for commodity operating systems, these principles would enable any OS to reduce the likelihood and severity of failures due to bugs in low-level software-hardware interactions.

We have incorporated these techniques in the SVA prototype and correspondingly modified the Linux 2.4.22 kernel previously ported to SVA [10]. Our new techniques required a significant redesign of SVA-OS, which is the API provided by SVA to a kernel for controlling hardware and using privileged hardware operations. The changes to the Linux kernel were generally simple changes to use the new SVA-OS API, even though the new API provides much more powerful protection for the entire kernel. We had to change only about 100 lines in the SVA kernel to conform to the new SVA-OS API.

We have evaluated the ability of our system to prevent kernel bugs due to kernel-hardware interactions, both with real reported bugs and injected bugs. Our system prevents two MMU bugs in Linux 2.4.22 for which exploit code is available. Both bugs crash the kernel when run under the original SVA. Moreover, as explained in Section 7.1, we would also prevent the e1000e bug in Linux 2.6 if that kernel is run on our system. Finally, the system successfully prevents all the low-level kernel-hardware interaction errors we have tried to inject.

We also evaluated the performance overheads for two servers and three desktop applications (two of which perform substantial I/O). Compared with the original SVA, the new techniques in this paper add very low or negligible overheads. Combined with the ability to prevent real-world exploits that would be missed otherwise, it clearly seems worthwhile to add these techniques to an existing memory safety system.

To summarize, the key contributions of this work are:

- We have presented novel mechanisms to ensure that low-level kernel-hardware interactions (e.g., context switching, thread creation, MMU changes, and I/O operations) do not violate assumptions used to enforce a safe execution environment.

- We have prototyped these techniques and shown that they can be used to enforce the assumptions made by a memory safety checker for a commodity kernel such as Linux. To our knowledge, no previous safety enforcement technique provides such guarantees to commodity system software.

- We have evaluated this system experimentally and shown that it is effective at preventing exploits in the above operations in Linux while incurring little overhead over and above the overhead of the underlying safe execution environment of SVA.

2 Breaking Memory Safety with Low-Level Kernel Operations

Informally, *a program is type-safe* if all operations in the program respect the types of their operands. For the purposes of this work, we say *a program is memory safe* if every memory access uses a previously initialized pointer variable; accesses the same object to which the pointer pointed initially;[1] and the object has not been deallocated. Memory safety is necessary for type safety (conversely, type safety implies memory safety) because dereferencing an uninitialized pointer, accessing the target object out of bounds, or dereferencing a dangling pointer to a freed object, can all cause accesses to unpredictable values and hence allow illegal operations on those values.

A safe programming language guarantees type safety and memory safety for all legal programs [34]; these guarantees also imply a *sound operational semantics* for programs in the language. Language implementations enforce these guarantees through a combination of compile-time type checking, automatic memory management (e.g., garbage collection or region-based memory management) to prevent dangling pointer references, and run-time checks such as array bounds checks and null pointer checks.

Four recent compiler-based systems for C, namely, CCured [30], SafeDrive [51], SAFECode [15], and SVA [10] enforce similar, but weaker, guarantees for C code. Their guarantees are weaker in two ways: (a) they provide type safety for only a subset of objects, and (b) three of the four systems — SafeDrive, SAFECode and SVA — permit dangling pointer references (use-after-free) to avoid the need for garbage collection. Unlike SafeDrive, however, SAFECode and SVA *guarantee* that dangling pointer references do not invalidate any of the other safety properties, i.e., partial type safety, memory safety, or a sound operational semantics [14, 15]. We refer to all these systems – safe languages or safety checking compilers – as providing a *safe execution environment*.

All of the above systems make some fundamental assumptions regarding the run-time environment in enforcing their safety guarantees. In particular, these systems assume that the code segment is static; control flow can only be altered through explicit branch instructions, call instructions, and visible signal handling; and that data is stored either in a flat, unchanging address space or in processor registers. Furthermore, data can only be read and written by direct loads and stores to memory or direct changes to processor registers.

Low-level system code routinely violates these assumptions. Operating system kernels, virtual machine monitors, language virtual machines such as a JVM or CLR, and user-level thread libraries often perform operations such as context switching, direct stack manipulation, memory mapped I/O, and MMU configuration, that violate these assumptions. More importantly, as explained in the rest of this section, perfectly *type-safe* code can violate many of these assumptions (through logical errors), i.e., such errors will not be prevented by the language in the first place. This is unacceptable for safe language implementations and, at least, undesirable for system software because these violations can compromise safety and soundness and thus permit the vulnerabilities a safe language was designed to prevent, such as buffer overflows or the creation of illegal pointer values.

There are, in fact, a small number of root causes (or categories of root causes) of all these violations. This section enumerates these root causes, and the next section describes the design principles by which these root causes can be eliminated. We assume throughout this discussion that a safety checker (through some combination of static and run-time checking) enforces the language-level safety guarantees of a safe execution environment, described above, for the kernel.[2] This allows us to assume that the run-time checker itself is secure, and that static analysis can be used soundly on kernel code [15]. Our goal is to ensure the integrity of the *as-*

[1] Note that we permit a pointer to "leave" its target object and later return, as long as it is not accessed while it is out of bounds [32].

[2] This work focuses on enforcing memory safety for the kernel. The same techniques could be applied to protect user-space threads from these violations.

sumptions made by this safety checker. We refer to the extensions that enforce these assumptions as a *verifier*.

Briefly, the fundamental categories of violations are:

- corrupting processor state when held in registers or memory;
- corrupting stack values for kernel threads;
- corrupting memory mapped I/O locations;
- corrupting code pages in memory;
- other violations that can corrupt arbitrary memory locations, including those listed above.

Unlike the last category, the first four above are errors that are specific to individual categories of memory.

2.1 Corrupting Processor State

Corrupting processor state can corrupt both data and control flow. The verifier must first ensure that processor state cannot be corrupted while on the processor itself, i.e., preventing arbitrary changes to processor registers. In addition, however, standard kernels save processor state (i.e., data and control registers) in memory where it is accessible by standard (even type-safe) load and store instructions. Any (buggy) code that modifies this state before restoring the state to the processor can alter control flow (the program counter, stack pointer, return address register, or condition code registers) or data values. In safe systems that permit dangling pointer references, processor state can also be corrupted if the memory used to hold saved processor state (usually located on the heap [5]) is freed and reallocated for other purposes.

Note that there are cases where the kernel makes explicit, legal, changes to the interrupted state of user-space code. For example, during signal handler dispatch, the kernel modifies interrupted program state that has been saved to memory, including the interrupted program's program counter and stack pointer [5]. Also, returning from a signal handler requires undoing the modifications made by signal delivery. The verifier must be able to distinguish legal from illegal changes to saved state.

2.2 Corrupting Stack State

The kernel directly manages the stacks of both user and kernel threads; it allocates and deallocates memory to hold them, sets up initial stack frames for new threads and signal handlers, and switches between stacks during a context switch or interrupt/system call return.

Memory for the stack is obtained from some standard memory allocation. Several safety violations are possible through this allocated memory. First, the memory for the stack should only be used for stack frames created during normal function calls and not directly modified via arbitrary stores;[3] such stores could corrupt the stack frames and thus compromise safety. Second, the memory for the stack must not be deallocated and reused for other memory objects while the stack is still in use. Third, a context switch must switch to a stack and its corresponding saved processor state as a pair; a context switch should not load processor state with the wrong stack or with a stack that has been deallocated. Fourth, after a stack is deallocated, live pointers to local variables allocated on the stack must not be dereferenced (the exiting thread may have stored pointers to such objects into global variables or the heap where they are accessible by other threads).

2.3 Corrupting Memory-Mapped I/O

Most systems today use memory-mapped I/O for controlling I/O devices and either memory-mapped I/O or DMA for performing data transfers. Many hardware architectures treat regular memory and memory-mapped I/O device memory (hereafter called I/O memory) identically, allowing a single set of hardware instructions to access both. From a memory safety perspective, however, it is better to treat regular memory and I/O memory as disjoint types of memory that are accessed using distinct instructions. First, I/O memory is not semantically the same as regular memory in that a load may not return the value last stored into the location; program analysis algorithms (used to enforce and optimize memory safety [15]) are not sound when applied to such memory. Second, I/O memory creates side-effects that regular memory does not. While erroneously accessing I/O memory instead of regular memory may not be a memory safety violation per se, it is still an error with potentially dire consequences. For example, the e1000e bug [9] caused fatal damage to hardware when an instruction (`cmpxchg`) that was meant to write to memory erroneously accessed memory-mapped I/O registers, which has undefined behavior. Therefore, for soundness of regular memory safety and for protection against a serious class of programming errors, it is best to treat regular memory and I/O memory as disjoint.

2.4 Corrupting Code

Besides the general memory corruption violations described below, there are only two ways in which the contents of code pages can be (or appear to be) corrupted. One is through self-modifying code (SMC); the other is through incorrect program loading operations (for new code or loadable kernel modules).

[3] An exception is when Linux stores the process's task structure at the bottom of the stack.

Self-modifying code directly modifies the sequence of instructions executed by the program. This can modify program behavior in ways not predicted by the compiler and hence bypass any of its safety checking techniques. For these reasons, most type-safe languages prohibit self-modifying code (which is distinct from “self-extending” behaviors like dynamic class loading). However, modern kernels use limited forms of self-modifying code for operations like enabling and disabling instrumentation [9] or optimizing synchronization for a specific machine configuration [8]. To allow such optimizations, the verifier must define limited forms of self-modifying code that do not violate the assumptions of the safety checker.

Second, the verifier must ensure that any program loading operation is implemented correctly. For example, any such operation, including new code, self-modifying code, or self-extending code (e.g., loadable kernel modules) requires flushing the instruction cache. Otherwise, cached copies of the old instructions may be executed out of the I-cache (and processors with split instruction/data caches may even execute old instructions with fresh data). This may lead to arbitrary memory safety violations for the kernel or application code.

2.5 General Memory Corruption

Finally, there are three kinds of kernel functionality that can corrupt arbitrary memory pages: (1) MMU configuration; (2) page swapping; and (3) DMA. Note that errors in any of these actions are generally invisible to a safety checking compiler and can violate the assumptions made by the compiler, as follows.

First, the kernel can violate memory safety with direct operations on virtual memory. Fundamentally, most of these are caused by creating an incorrect virtual-to-physical page mapping. Such errors include modifying mappings in the range of kernel stack memory, mapping the same physical page into two virtual pages (unintentionally), and changing a virtual-to-physical mapping for a live virtual page. As before, any of these errors can occur even with a type-safe language.

A second source of errors is in page swapping. When a page of data is swapped in on a page fault, memory safety can be violated if the data swapped in is not identical to the data swapped out from that virtual page. For example, swapping in the wrong data can cause invalid data to appear in pointers that are stored in memory.

Finally, a third source of problems is DMA. DMA introduces two problems. First, a DMA configuration error, device driver error, or device firmware error can cause a DMA transfer to overwrite arbitrary physical memory, violating type-safety assumptions. Second, even a correct DMA transfer may bring in unknown data which cannot be used in a type-safe manner, unless special language support is added to enable that, e.g., to prevent such data being used as pointer values, as in the SPIN system [21].

3 Design Principles

We now describe the general design principles that a memory safe system can use to prevent the memory errors described in Section 2. As described earlier, we assume a safety checker already exists that creates a safe execution environment; the *verifier* is the set of extensions to the safety checker that enforces the underlying assumptions of the checker. Examples of safety checkers that could benefit directly from such extensions include SVA, SafeDrive, and XFI. We also assume that the kernel source code is available for modification.

Processor State: Preventing the corruption of processor state involves solving several issues. First, the verifier must ensure that the kernel does not make arbitrary changes to CPU registers. Most memory safe systems already do this by not providing instructions for such low-level modifications. Second, the verifier must ensure that processor state saved by a context switch, interrupt, trap, or system call is not accessed by memory load and store instructions. To do this, the verifier can allocate the memory used to store processor state within its own memory and allow the kernel to manipulate that state via special instructions that take an opaque handle (e.g., a unique integer) to identify which saved state buffer to use. For checkers like SVA and SafeDrive, the safety checker itself prevents the kernel from manufacturing and using pointers to these saved state buffers (e.g., via checks on accesses that use pointers cast from integers). Additionally, the verifier should ensure that the interface for context switching leaves the system in a known state, meaning that a context switch should either succeed completely or fail.

There are operations in which interrupted program state needs to be modified by the kernel (e.g., signal handler dispatch). The verifier must provide instructions for doing controlled modifications of interrupted program state; for example, it can provide an instruction to push function call frames on to an interrupted program’s stack [11]. Such instructions must ensure that either their modifications cannot break memory safety or that they only modify the saved state of interrupted user-space programs (modifying user-space state cannot violate the kernel’s memory safety).

Stack State: The memory for a kernel stack and for the processor state object (the in-memory representation of processor state) must be created in a single operation (instead of by separate operations), and the verifier should ensure that the kernel stack and processor state object are always used and deallocated together. To ease implementation, it may be desirable to move some low-level, error-prone stack and processor state object initialization code into the verifier. The verifier must also ensure that memory loads and stores do not modify the kernel stack (aside from accessing local variables) and that local variables stored on the stack can no longer be accessed when the kernel stack is destroyed.

Memory-mapped I/O: The verifier must require that all I/O object allocations be identifiable in the kernel code, (e.g., declared via a pseudo-allocator). It should also ensure that only special I/O read and write instructions can access I/O memory (these special instructions can still be translated into regular memory loads and stores for memory-mapped I/O machines) and that these special instructions cannot read or write regular memory objects. If the verifier uses type-safety analysis to optimize run-time checks, it should consider I/O objects (objects analogous to memory objects but that reside in memory-mapped I/O pages) to be *type-unsafe* as the device's firmware may use the I/O memory in a type-unsafe fashion. Since it is possible for a pointer to point to both I/O objects and memory objects, the verifier should place run-time checks on such pointers to ensure that they are accessing the correct type of object (memory or I/O), depending upon the operation in which the pointer is used.

Kernel Code: The verifier must not permit the kernel to modify its code segment. However, it can support a limited version of self-modifying code that is easy to implement and able to support the uses of self-modifying code found in commodity kernels. In our design, the kernel can specify regions of code that can be enabled and disabled. The verifier will be responsible for replacing native code with no-op instructions when the kernel requests that code be disabled and replacing the no-ops with the original code when the kernel requests the code to be re-enabled. When analyzing code that can be enabled and disabled, the verifier can use conservative analysis techniques to generate results that are correct regardless of whether the code is enabled or disabled. For example, our pointer analysis algorithm, like most other inter-procedural ones used in production compilers, computes a *may-points-to* result [24], which can be computed with the code enabled; it will still be correct, though perhaps conservative, if the code is disabled.

To ensure that the instruction cache is properly flushed, our design calls for the safety checker to handle all translation to native code. The safety checker already does this in JVMs, safe programming languages, and in the SVA system [10]. By performing all translation to native code, the verifier can ensure that all appropriate CPU caches are flushed when new code is loaded into the system.

General Memory Corruption: The verifier must implement several types of protection to handle the general memory corruption errors in Section 2.5.

MMU configuration: To prevent MMU misconfiguration errors, the verifier must be able to control access to hardware page tables or processor TLBs and vet changes to the MMU configuration before they are applied. Implementations can use para-virtualization techniques [16] to control the MMU. The verifier must prevent pages containing kernel memory objects from being made accessible to non-privileged code and ensure that pages containing kernel stack frames are not mapped to multiple virtual addresses (i.e., double mapped) or unmapped before the kernel stack is destroyed.[4] Verifiers optimizing memory access checks must also prohibit double mappings of pages containing *type known* objects; this will prevent data from being written into the page in a way that is not detected by compiler analysis techniques. Pages containing type-unknown memory objects can be mapped multiple times since run-time checks already ensure that the data within them does not violate any memory safety properties. The verifier must also ensure that MMU mappings do not violate any other analysis results upon which optimizations depend.

Page swapping: For page swapping, the kernel must notify the verifier before swapping a page out (if not, the verifier will detect the omission on a subsequent physical page remapping operation). The verifier can then record any metadata for the page as well as a checksum of the contents and use these when the page is swapped back in to verify that the page contents have not changed.

DMA: The verifier should prevent DMA transfers from overwriting critical memory such as the kernel's code segment, the verifier's code and data, kernel stacks (aside from local variables), and processor state objects. Implementation will require the use of IOMMU techniques like those in previous work [17, 36]. Additionally, if the verifier uses type information to optimize memory safety checks, it must consider the memory accessible via DMA as type-unsafe. This solution is strictly stronger than previous work (like that in SPIN [21]): it allows pointer values in input data whereas they do not (and they do not guarantee type safety for other input data).

Entry Points: To ensure control-flow integrity, the kernel should not be entered in the middle of a function. Therefore, the verifier must ensure that all interrupt, trap, and system call handlers registered by the kernel are the initial address of a valid function capable of servicing the interrupt, trap, or system call, respectively.

[4] We assume the kernel does not swap stack pages to disk, but the design can be extended easily to allow this.

4 Background: Secure Virtual Architecture

The Secure Virtual Architecture (SVA) system (Figure 1) places a compiler-based virtual machine between the processor and the traditional software stack [10, 11]. The virtual machine (VM) presents a virtual instruction set to the software stack and translates virtual instructions to the processor's native instruction set either statically (the default) or dynamically. The virtual instruction set is based on the LLVM code representation [23], which is designed to be low-level and language-independent, but still enables sophisticated compiler analysis and transformation techniques. This instruction set can be used for both user-space and kernel code [11].

SVA optionally provides strong safety guarantees for C/C++ programs compiled to its virtual instruction set, close to that of a safe language. The key guarantees are:

1. *Partial type safety*: Operations on a subset of data are type safe.

2. *Memory safety*: Loads and stores only access the object to which the dereferenced pointer initially pointed, and within the bounds of that object.

3. *Control flow integrity*: The kernel code only follows execution paths predicted by the compiler; this applies to both branches and function calls.

4. *Tolerating dangling pointers*: SVA does not detect uses of dangling pointers but *guarantees that they are harmless*, either via static analysis (for type-safe data) or by detecting violations through run-time checks (for non-type safe data).

5. *Sound operational semantics*: SVA defines a virtual instruction set with an operational semantics that is guaranteed not to be violated by the kernel code; sound program analysis or verification tools can be built on this semantics.

Briefly, SVA provides these safety guarantees as follows. First, it uses a pointer analysis called Data Structure Analysis (DSA) [24] to partition memory into logical partitions ("points to sets") and to check which partitions are always accessed or indexed with a single type. These partitions are called "type-known" (TK); the rest are "type-unknown" (TU). SVA then creates a run-time representation called a "metapool" for each partition. It maintains a lookup table in each metapool of memory objects and their bounds to support various run-time checks. Maintaining a table per metapool instead of a single global table greatly improves the performance of the run-time checks [14].

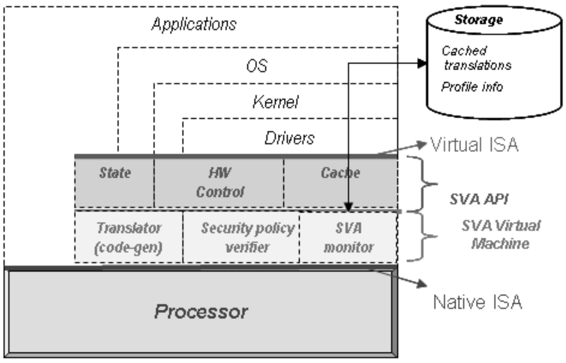

Figure 1: System Organization with SVA [10]

Compile-time analysis with DSA guarantees that all TK partitions are type-safe. Moreover, all uses of data and function pointers loaded *out of* TK partitions are type safe. SVA simply has to ensure that dangling pointer references to TK metapools cannot create a type violation by enforcing two constraints: (a) objects in TK metapools are aligned identically; and (b) freed memory from such a metapool is never used for a different metapool until the former is destroyed. These constraints are enforced by modifying the kernel allocators manually during the process of porting the kernel to SVA; this means that the allocators are effectively trusted and not checked. To enforce these constraints for stack objects belonging to TK metapools, SVA automatically modifies the kernel code to allocate such objects on the heap. Together, these guarantee that a pointer to a freed object and a new object (including array elements) access values of identical type [15].

At run-time, the SVA VM (thereafter called VM) performs a number of additional checks and operations. All globals and allocated objects are registered in the metapool to which they belong (derived from the target partition of the return pointer). Loads and stores that use pointers loaded from TU metapools are checked by looking up the target address in the metapool lookup table. Note that this works whether or not the pointer value is a dangling pointer, and even for pointers "manufactured" by casting arbitrary integers. Similarly, it checks function pointers obtained from TU metapools to ensure that they only access one of the target functions of that pointer predicted by DSA. Run-time checks also ensure that pointers to TK objects that are loaded from TU memory objects are checked since a TU object may have an invalid value for the TK pointer. All array indexing operations for TK or TU metapools are checked in the lookup table, which records the bounds for each object [14][5].

Note that the VM relies on the safe execution environ-

[5] Note that we permit a pointer to "leave" its target object and later return, as long as it is not accessed while it is out of bounds [32].

ment to protect the VM code and data memory instead of using the MMU and incurring the cost of switching page tables on every VM invocation. Since the environment prevents access to unregistered data objects or outside the bounds of legal objects, we can simply monitor all run-time kernel object registrations and ensure that they do not reside in VM code or data pages.

A subset of the SVA instruction set, SVA-OS, provides instructions designed to support an operating system's special interaction with the hardware [10, 11]. These include instructions for loading from/storing to I/O memory, configuring the MMU, and manipulating program state. An important property is that a kernel ported to SVA using the SVA-OS instructions *contains no assembly code*; this simplifies the compiler's task of safety checking within SVA. Nevertheless, these instructions provide low-level hardware interactions that can generate all the problems described in Section 2 if used incorrectly; it is very difficult for the compiler to check their correct use in the original design. In particular, the VM does not perform any special checks for processor state objects, direct stack manipulation, memory mapped I/O locations, MMU configuration changes, or DMA operations. Also, it disallows self-modifying code.

For example, we tested two [39, 42] of the three reported low-level errors we found for Linux 2.4.22, the kernel version ported to SVA (we could not try the third [3] for reasons explained in Section 7.1). Although both are memory safety violations, *neither of them was detected or prevented by the original SVA.*

5 Design

Our design is an extension of the original Secure Virtual Architecture (SVA) described in Section 4. SVA provides strong memory safety guarantees for kernel code and an abstraction of the hardware that is both low-level (e.g., context switching, I/O, and MMU configuration policies are still implemented in the kernel), yet easy to analyze (because the SVA-OS instructions for interacting with hardware are slightly higher level than typical processor instructions). Below, we describe our extensions to provide memory safety in the face of errors in kernel-hardware interactions.

5.1 Context Switching

Previously, the SVA system performed context switching using the `sva_load_integer` and `sva_save_integer` instructions [10], which saved from and loaded into the processor the processor state (named Integer State). These instructions stored processor state in a kernel allocated memory buffer which could be later modified by memory-safe store instructions or freed by the kernel deallocator. Our new design calls a single instruction named `sva_swap_integer` (see Table 1) that saves the old processor state and loads the new state in a single operation.

This design has all of the necessary features to preserve memory safety when context switching. The `sva_swap_integer` instruction allocates the memory buffer to hold processor state within the VM's memory and returns an opaque integer identifier which can be used to re-load the state in a subsequent call to `sva_swap_integer`. Combined with SVA's original protections against manufactured pointers, this prevents the kernel from modifying or deallocating the saved processor state buffer. The design also ensures correct deallocation of the memory buffer used to hold processor state. The VM tracks which identifiers are mapped to allocated state buffers created by `sva_swap_integer`; these memory buffer/identifier pairs are kept alive until the state is placed back on the processor by another call to `sva_swap_integer`. Once state is placed back on the processor, the memory buffer is deallocated, and the identifier invalidated to prevent the kernel from trying to restore state from a deallocated state buffer.

Finally, `sva_swap_integer` will either succeed to context switch and return an identifier for the saved processor state, or it will fail, save no processor state, and continue execution of the currently running thread. This ensures that the kernel stack and the saved processor state are always synchronized.

5.2 Thread Management

A thread of execution consists of a stack and a saved processor state that can be used to either initiate or continue execution of the thread. Thread creation is therefore comprised of three operations: allocating memory for the new thread's stack, initializing the new stack, and creating an initial state that can be loaded on to the processor using `sva_swap_integer`.

The VM needs to know where kernel stacks are located in order to prevent them from being written by load and store instructions. We introduce a new SVA instruction, `sva_declare_stack`, which a kernel uses to declare that a memory object will be used as a stack. During pointer analysis, any pointers passed to `sva_declare_stack` and pointers that alias with such pointers are marked with a special *DeclaredStack* flag; this flag indicates that run-time checks are needed on stores via such pointers to ensure that they are not writing into a kernel stack. The compiler, on seeing an `sva_declare_stack` instruction, will also verify, statically (via pointer analysis) if possible but at run-time if necessary, that the memory object used for the new stack is either a global or heap object; this will prevent

Name	Description
sva_swap_integer	Saves the current processor state into an internal memory buffer, loads previously saved state referenced by its ID, and returns the ID of the new saved state.
sva_declare_stack	Declares that a memory object is to be used as a new stack.
sva_release_stack	Declares that a memory object is no longer used as a stack.
sva_init_stack	Initializes a new stack.

Table 1: SVA Instructions for Context Switching and Thread Creation.

stacks from being embedded within other stacks. After this check is done, sva_declare_stack will unregister the memory object from the set of valid memory objects that can be accessed via loads and stores and record the stack's size and location within the VM's internal data structures as a valid kernel stack.

To initialize a stack and the initial processor state that will use the memory as a stack, we introduce sva_init_stack; this instruction will initialize the stack and create a new saved Integer State which can be used in sva_swap_integer to start executing the new thread. The sva_init_stack instruction verifies (either statically or at run-time) that its argument has previously been declared as a stack using sva_declare_stack. When the new thread wakes up, it will find itself running within the function specified by the call to sva_init_stack; when this function returns, it will return to user-space at the same location as the original thread entered.

Deleting a thread is composed of two operations. First, the memory object containing the stack must be deallocated. Second, any Integer State associated with the stack that was saved on a context switch must be invalidated. When the kernel wishes to destroy a thread, it must call the sva_release_stack instruction; this will mark the stack memory as a regular memory object so that it can be freed and invalidates any saved Integer State associated with the stack.

When a kernel stack is deallocated, there may be pointers in global or heap objects that point to memory (i.e., local variables) allocated on that stack. SVA must ensure that dereferencing such pointers does not violate memory safety. Type-unsafe stack allocated objects are subject to load/store checks and are registered with the SVA virtual machine [10]. In order for the sva_release_stack instruction to invalidate such objects when stack memory is reclaimed, the VM records information on stack object allocations and associates this information with the metadata about the stack in which the object is allocated. In this way, when a stack is deallocated, any live objects still registered with the virtual machine are automatically invalidated as well; run-time checks will no longer consider these stack allocated objects to be valid objects. Type-known stack allocated objects can never be pointed to by global or heap objects; SVA already transforms such stack allocations into heap allocations [15, 10] to make dangling pointer dereferencing to type-known stack allocated objects safe [15].

5.3 Memory Mapped I/O

To ensure safe use of I/O memory, our system must be able to identify where I/O memory is located and when the kernel is legitimately accessing it.

Identifying the location of I/O memory is straightforward. In most systems, I/O memory is located at (or mapped into) known, constant locations within the system's address space, similar to global variables. In some systems, a memory-allocator-like function may remap physical page frames corresponding to I/O memory to a virtual memory address [5]. The insight is that I/O memory is grouped into objects just like regular memory; in some systems, such I/O objects are even allocated and freed like heap objects (e.g., Linux's ioremap() function [5]). To let the VM know where I/O memory is located, we must modify the kernel to use a pseudo-allocator that informs the VM of global I/O objects; we can also modify the VM to recognize I/O "allocators" like ioremap() just like it recognizes heap allocators like Linux's kmalloc() [5].

Given this information, the VM needs to determine which pointers may point to I/O memory. To do so, we modified the SVA points-to analysis algorithm [24] to mark the target (i.e., the "points-to set") of a pointer holding the return address of the I/O allocator with a special *I/O flag*. This also flags other pointers aliased to such a pointer because any two aliased pointers point to a common target [24].

We also modified the points-to analysis to mark I/O memory as *type-unknown*. Even if the kernel accesses I/O memory in a type-consistent fashion, the firmware on the I/O device may not. *Type-unknown* memory incurs additional run-time checks but allows kernel code to safely use pointer values in such memory as pointers.

We also extended SVA to record the size and virtual address location of every I/O object allocation and deallocation by instrumenting every call to the I/O allocator and deallocator functions. At run-time, the VM records these I/O objects in a per-metapool data structure that is disjoint from the structure used to record the bounds of regular memory objects. The VM also uses new run-time checks for checking I/O load and store instructions.

Since I/O pointers can be indexed like memory pointers (an I/O device may have an array of control registers), the bounds checking code must check both regular memory objects and I/O memory objects. Load and store checks on regular memory pointers *without the I/O flag* remain unchanged; they only consider memory objects. New run-time checks are needed on both memory and I/O loads and stores for pointers that have both the I/O flag and one or more of the memory flags (heap, stack, global) to ensure that they only access regular or I/O memory objects, respectively.

5.4 Safe DMA

We assume the use of an IOMMU for preventing DMA operations from overflowing object bounds or writing to the wrong memory address altogether [13]. The SVA virtual machine simply has to ensure that the I/O MMU is configured so that DMA operations cannot write to the virtual machine's internal memory, kernel code pages, pages which contain type-safe objects, and stack objects.

We mark all memory objects that may be used for DMA operations as type-unsafe, similar to I/O memory that is accessed directly. We assume that any pointer that is *stored into* I/O memory is a potential memory buffer for DMA operations. We require alias analysis to identify such stores; it simply has to check that the target address is in I/O memory and the store value is of pointer type. We then mark the points-to set of the store value pointer as *type-unknown*.

5.5 Virtual Memory

Our system must control the MMU and vet changes to its configuration to prevent safety violations and preserve compiler-inferred analysis results. Below, we describe the mechanism by which our system monitors and controls MMU configuration and then discuss how we use this mechanism to enforce several safety properties.

5.5.1 Controlling MMU Configuration

SVA provides different MMU interfaces for hardware TLB processors and software TLB processors [11]. For brevity, we describe only the hardware TLB interface and how our design uses it to control MMU configuration.

The SVA interface for hardware TLB systems (given in Table 2) is similar to those used in VMMs like Xen [16] and is based off the `paravirtops` interface [50] found in Linux 2.6. The page table is a 3-level page table, and there are instructions for changing mappings at each level. In this design, the OS first tells the VM which memory pages will be used for the page table (it must specify at what level the page will appear in the table); the VM then takes control of these pages by zeroing them (to prevent stale mappings from being used) and marking them read-only to prevent the OS from accessing them directly. The OS must then use special SVA instructions to update the translations stored in these page table pages; these instructions allow SVA to first inspect and modify translations before accepting them and placing them into the page table. The `sva_load_pagetable` instruction selects which page table is in active use and ensures that only page tables controlled by SVA are ever used by the processor. This interface, combined with SVA's control-flow integrity guarantees [10], ensure that SVA maintains control of all page mappings on the system.

5.5.2 Memory Safe MMU Configuration

For preventing memory safety violations involving the MMU, the VM needs to track two pieces of information. First, the VM must know the purpose of various ranges of the virtual address space; the kernel must provide the virtual address ranges of user-space memory, kernel data memory, and I/O object memory. This information will be used to prevent physical pages from being mapped into the wrong virtual addresses (e.g., a memory mapped I/O device being mapped into a virtual address used by a kernel memory object). A special instruction permits the kernel to communicate this information to the VM.

Second, the VM must know how physical pages are used, how many times they are mapped into the virtual address space, and whether any MMU mapping makes them accessible to unprivileged (i.e., user-space) code. To track this information, the VM associates with each physical page a set of flags and counters. The first set of flags are mutually exclusive and indicate the purpose of the page; a page can be marked as: `L1` (Level-1 page table page), `L2` (Level-2 page table page), `L3` (Level-3 page table page), `RW` (a standard kernel page holding memory objects), `IO` (a memory mapped I/O page), `stack` (kernel stack), `code` (kernel or SVA code), or `svamem` (SVA data memory). A second flag, the `TK` flag, specifies whether a physical page contains *type-known* data. The VM also keeps a count of the number of virtual pages mapped to the physical page and a count of the number of mappings that make the page accessible to user-space code.

The flags are checked and updated by the VM whenever the kernel requests a change to the page tables or performs relevant memory or I/O object allocation. Calls to the memory allocator are instrumented to set the RW and, if appropriate, the TK flag on pages backing the newly allocated memory object. On system boot, the VM sets the `IO` flag on physical pages known to be memory-mapped I/O locations. The `stack`

Name	Description
`sva_end_mem_init`	End of the virtual memory boot initialization. Flags all page table pages, and mark them read-only.
`sva_declare_l1_page`	Zeroes the page and flags it read-only and L1.
`sva_declare_l2_page`	Zeroes the page and flags it read-only and L2.
`sva_declare_l3_page`	Puts the default mappings in the page and flags it read-only and L3.
`sva_remove_l1_page`	Unflags the page read-only and L1.
`sva_remove_l2_page`	Unflags the page read-only and L2.
`sva_remove_l3_page`	Unflags the page read-only and L3.
`sva_update_l1_mapping`	Updates the mapping if the mapping belongs to an L1 page and the page is not already mapped for a type known pool, sva page, code page, or stack page.
`sva_update_l2_mapping`	Updates the mapping if the mapping belongs to an L2 page and the new mapping is for an L1 page.
`sva_update_l3_mapping`	Updates the mapping if the mapping belongs to an L3 page and the new mapping is for an L2 page.
`sva_load_pagetable`	Check that the physical page is an L3 page and loads it in the page table register.

Table 2: MMU Interface for a Hardware TLB Processor.

flag is set and cleared by `sva_declare_stack` and `sva_release_stack`, respectively. Changes to the page table via the instructions in Table 2 update the counters and the `L1`, `L2`, and`L3` flags.

The VM uses all of the above information to detect, at run-time, violations of the safety requirements in Section 3. Before inserting a new page mapping, the VM can detect whether the new mapping will create multiple mappings to physical memory containing *type-known* objects, map a page into the virtual address space of the VM or kernel code segment, unmap or double map a page that is part of a kernel stack, make a physical page containing kernel memory accessible to user-space code, or map memory-mapped I/O pages into a kernel memory object (or vice-versa). Note that SVA currently trusts the kernel memory allocators to (i) return different virtual addresses for every allocation, and (ii) not to move virtual pages from one metapool to another until the original metapool is destroyed.

5.6 Self-modifying Code

The new SVA system supports the restricted version of self-modifying code described in Section 3: OS kernels can disable and re-enable pre-declared pieces of code. SVA will use compile-time analysis carefully to ensure that replacing the code with no-op instructions will not invalidate the analysis results.

We define four new instructions to support self-modifying code. The first two instructions, `sva_begin_alt` and `sva_end_alt` enclose the code regions that may be modified at runtime. They must be properly nested and must be given a unique identifier. The instructions are not emitted in the native code. The two other instructions, `sva_disable_code` and `sva_enable_code` execute at runtime. They take the identifier given to the `sva_begin_alt` and `sva_end_alt` instructions. `sva_disable_code` saves the previous code and inserts no-ops in the code, and `sva_enable_code` restores the previous code.

With this approach, SVA can support most uses of self-modifying code in operating systems. For instance, it supports the `alternatives`[6] framework in Linux 2.6 [8] and Linux's `ftrace` tracing support [9] which disables calls to logging functions at run-time.

5.7 Interrupted State

On an interrupt, trap, or system call, the original SVA system saves processor state within the VM's internal memory and permits the kernel to use specialized instructions to modify the state via an opaque handle called the interrupt context [10, 11]. These instructions, which are slightly higher-level than assembly code, are used by the kernel to implement operations like signal handler dispatch and starting execution of user programs. Since systems such as Linux can be interrupted while running kernel code [5], these instructions can violate the kernel's memory safety if used incorrectly on interrupted kernel state. To address these issues, we introduce several changes to the original SVA design.

First, we noticed that all of the instructions that manipulate interrupted program state are either memory safe (e.g., the instruction that unwinds stack frames for kernel exception handling [11]) or only need to modify the interrupted state of user-space programs. Hence, all instructions that are not intrinsically memory safe will verify that they are modifying interrupted user-space program state. Second, the opaque handle to the interrupt context will be made implicit so that no run-time checks are needed to validate it when it is used. We have observed that the Linux kernel only operates upon the most recently created interrupt context; we do not see a need for other operating systems of similar design to do so, either. Without an explicit handle to the interrupt context's location in memory, no validation code is needed, and the kernel cannot create a pointer to the saved program state (except for explicit integer to pointer casts, uses of which will be caught by SVA's existing checks) [10].

[6] Linux 2.6, file include/asm-x86/alternative.h

5.8 Miscellaneous

To ensure control-flow integrity requirements, the VM assumes control of the hardware interrupt descriptor table; the OS kernel must use special instructions to associate a function with a particular interrupt, trap, or system call [11, 29]. Similar to indirect function call checks, SVA can use static analysis and run-time checks to ensure that only valid functions are registered as interrupt, trap, or system call handlers.

SVA provides two sets of atomic memory instructions: `sva_fetch_and_phi` where phi is one of several integer operations (e.g., add), and `sva_compare_and_swap` which performs an atomic compare and swap. The static and run-time checks that protect regular memory loads and stores also protect these operations.

6 Modifications to the Linux Kernel

We implemented our design by improving and extending the original SVA prototype and the SVA port of the Linux 2.4.22 kernel [10]. The previous section described how we modified the SVA-OS instructions. Below, we describe how we modified the Linux kernel to use these new instructions accordingly. We modified less than 100 lines from the original SVA kernel to port our kernel to the new SVA-OS API; the original port of the i386 Linux kernel to SVA modified 300 lines of architecture-independent code and 4,800 lines of architecture-dependent code [10].

6.1 Changes to Baseline SVA

The baseline SVA system in our evaluation (Section 7) is an improved version of the original SVA system [10] that is suitable for determining the extra overhead incurred by the run-time checks necessitated by the design in Section 5. First, we fixed several bugs in the optimization of run-time checks. Second, while the original SVA system does not analyze and protect the whole kernel, there is no fundamental reason why it cannot. Therefore, we chose to disable optimizations which apply only to incomplete kernel code for the experiments in Section 7. Third, the new baseline SVA system recognizes `ioremap()` as an allocator function even though it does not add run-time checks for I/O loads and stores. Fourth, we replaced most uses of the `__get_free_pages()` page allocator with `kmalloc()` in code which uses the page allocator like a standard memory allocator; this ensures that most kernel allocations are performed in kernel pools (i.e., `kmem_cache_ts`) which fulfill the requirements for allocators as described in the original SVA work [10].

We also modified the SVA Linux kernel to use the new SVA-OS instruction set as described below. This ensured that the only difference between our baseline SVA system and our SVA system with the low-level safety protections was the addition of the run-time checks necessary to ensure safety for context switching, thread management, MMU, and I/O memory safety.

6.2 Context Switching/Thread Creation

The modifications needed for context switching were straightforward. We simply modified the `switch_to` macro in Linux [5] to use the `sva_swap_integer` instruction to perform context switching.

Some minor kernel modifications were needed to use the new thread creation instructions. The original i386 Linux kernel allocates a single memory object which holds both a thread's task structure and the kernel stack for the thread [5], but this cannot be done on our system because `sva_declare_stack` requires that a stack consumes an entire memory object. For our prototype, we simply modified the Linux kernel to perform separate allocations for the kernel stack and the task structure.

6.3 I/O

As noted earlier, our implementation enhances the pointer analysis algorithm in SVA (DSA [24]) to mark pointers that may point to I/O objects. It does this by finding calls to the Linux `__ioremap()` function. To make implementation easier, we modified `ioremap()` and `ioremap_nocache()` in the Linux source to be macros that call `__ioremap()`.

Our test system's devices do not use global I/O memory objects, so we did not implement a pseudo allocator for identifying them. Also, we did not modify DSA to mark memory stored into I/O device memory as type-unknown. The difficulty is that Linux casts pointers into integers before writing them into I/O device memory. The DSA implementation does not have solid support for tracking pointers through integers i.e., it does not consider the case where an integer may, in fact, be pointing to a memory object. Implementing these changes to provide DMA protection is left as future work.

6.4 Virtual Memory

We implemented the new MMU instructions and run-time checks described in Section 5.5 and ported the SVA Linux kernel to use the new instructions. Linux already contains macros to allocate, modify and free page table pages. We modified these macros to use our new API (which is based on the `paravirtops` interface from Linux 2.6). We implemented all of the run-time checks except for those that ensure that I/O device memory isn't mapped into kernel memory objects. These checks require that the kernel allocate all I/O memory objects within a predefined range of the virtual address space, which our Linux kernel does not currently do.

7 Evaluation and Analysis

Our evaluation has two goals. First, we wanted to determine whether our design for low-level software/hardware interaction was effective at stopping security vulnerabilities in commodity OS kernels. Second, we wanted to determine how much overhead our design would add to an already existing memory-safety system.

7.1 Exploit Detection

We performed three experiments to verify that our system catches low-level hardware/software errors: First, we tried two different exploits on our system that were reported on Linux 2.4.22, the Linux version that is ported to SVA. The exploits occur in the MMU subsystem; both give an attacker root privileges. Second, we studied the e1000e bug [9]. We could not duplicate the bug because it occurs in Linux 2.6, but we explain why our design would have caught the bug if Linux 2.6 had been ported to SVA. Third, we inserted many low-level operation errors inside the kernel to evaluate whether our design prevents the safety violations identified in Section 2.

Linux 2.4.22 exploits. We have identified three reported errors for Linux 2.4.22 caused by low-level kernel-hardware interactions [3, 39, 42]. Our experiment is limited to these errors because we needed hardware/software interaction bugs that were in Linux 2.4.22. Of these, we could not reproduce one bug due to a lack of information in the bug report [3]. The other two errors occur in the `mremap` system call but are distinct errors.

The first exploit [42] is due to an overflow in a count of the number of times a page is mapped. The exploit code overflows the counter by calling `fork`, `mmap`, and `mremap` a large number of times. It then releases the page, giving it back to the kernel. However, the exploit code still has a reference to the page; therefore, if the page is reallocated for kernel use, the exploit code can read and modify kernel data. Our system catches this error because it disallows allocating kernel objects in a physical page mapped in user space.

The second exploit [39] occurs because of a missing error check in `mremap` which causes the kernel to place page table pages with valid page table entries into the page table cache. However, the kernel assumes that page table pages in the page table cache do not contain any entries. The exploit uses this vulnerability by calling `mmap, mremap` and `munmap` to release a page table page with page entries that contain executable memory. Then, on an `exec` system call, the linker, which executes with root privileges, allocates a page table page, which happens to be the previously released page. The end result is that the linker jumps to the exploit's executable memory and executes the exploit code with root privileges. The SVA VM prevents this exploit by always zeroing page table pages when they are placed in a page directory so that no new, unintended, memory mappings are created for existing objects.

The e1000e bug. The fundamental cause of the e1000e bug is a memory load/store (the x86 `cmpxchg` instruction) on a dangling pointer, which happens to point to an I/O object. The `cmpxchg` instruction has non-deterministic behavior on I/O device memory and may corrupt the hardware. The instruction was executed by the `ftrace` subsystem, which uses self-modifying code to trace the kernel execution. It took many weeks for skilled engineers to track the problem. With our new safety checks, SVA would have detected the bug at its first occurrence. The self-modifying code interface of SVA-OS only allows enabling and disabling of code; writes to what the kernel (incorrectly) thought was its code is not possible. SVA actually has a second line of defense if (hypothetically) the self-modifying code interface did not detect it: SVA would have prevented the I/O memory from being mapped into code pages, and thus prevented this corruption. (And, hypothetically again, if a dangling pointer to a data object had caused the bug, SVA would have detected any ordinary reads and writes trying to write to I/O memory locations.)

Kernel error injection. To inject errors, we added new system calls into the kernel; each system call triggers a specific kind of kernel/hardware interaction error that either corrupts memory or alters control flow. We inserted four different errors. The first error modifies the saved Integer State of a process so that an invalid Integer State is loaded when the process is scheduled. The second error creates a new MMU mapping of a page containing type-known kernel memory objects and modifies the contents of the page. The third error modifies the MMU mappings of pages in the stack range. The fourth error modifies the internal metadata of SVA to set incorrect bounds for all objects. This last error shows that with the original design, we can *disable the SVA memory safety checks that prevent Linux exploits*; in fact, it would not be difficult to do so with this bug alone for three of the four kernel exploits otherwise prevented by SVA [10].

All of the injected errors were caught by the new SVA implementation. With the previous implementation, these errors either crash the kernel or create undefined behavior. This gives us confidence about the correctness of our new design and implementation of SVA. Note that we only injected errors that our design addresses because we believe that our design is "complete" in terms of the possible errors due to kernel-hardware interactions. Nevertheless, the injection experiments are useful because they *validate that the design and implementation actually solve these problems*.

7.2 Performance

To determine the impact of the additional run-time checks on system performance, we ran several experiments with applications typically used on server and end-user systems. We ran tests on the original Linux 2.4.22 kernel (marked i386 in the figures and tables), the same kernel with the original SVA safety checks [10] (marked SVA), and the SVA kernel with our safety checks for low-level software/hardware interactions (marked SVA-OS).

It is important to note that an underlying memory safety system like SVA can incur significant run-time overhead for C code, especially for a commodity kernel like Linux that was not designed for enforcement of memory safety. Such a system is not the focus of this paper. Although we present our results relative to the original (unmodified) Linux/i386 system for clarity, we focus the discussion on the excess overheads introduced by SVA-OS beyond those of SVA since the new techniques in SVA-OS are the subject of the current work.

We ran these experiments on a dual-processor AMD Athlon 2100+ at 1,733 MHz with 1 GB of RAM and a 1 Gb/s network card. We configured the kernel as an SMP kernel but ran it in on a single processor since the SVA implementation is not yet SMP safe. Network experiments used a dedicated 1 Gb/s switch. We ran our experiments in single-user mode to prevent standard system services from adding noise to our performance numbers.

We used several benchmarks in our experiments: the thttpd Web server, the OpenSSH sshd encrypted file transfer service, and three local applications – bzip2 for file compression, the lame MP3 encoder, and a perl interpreter. These programs have a range of different demands on kernel operations. Finally, to understand why some programs incur overhead while others do not, we used a set of microbenchmarks including the HBench-OS microbenchmark suite [6] and two new tests we wrote for the poll and select system calls.

Application Performance First, we used ApacheBench to measure the file-transfer bandwidth of the thttpd web server [31] serving static HTML pages. We configured ApacheBench to make 5000 requests using 25 simultaneous connections. Figure 2 shows the results of both the original SVA kernel and the SVA kernel with the new run-time checks described in Section 5. Each bar is the average bandwidth of 3 runs of the experiment; the results are normalized to the original i386 Linux kernel. For small files (1 KB - 32 KB) in which the original SVA system adds significant overhead, our new run-time checks incur a small amount of additional overhead (roughly a 9% decrease in bandwidth relative to the SVA kernel). However, for larger file sizes (64 KB or more), the SVA-OS checks add negligible overhead to the original SVA system.

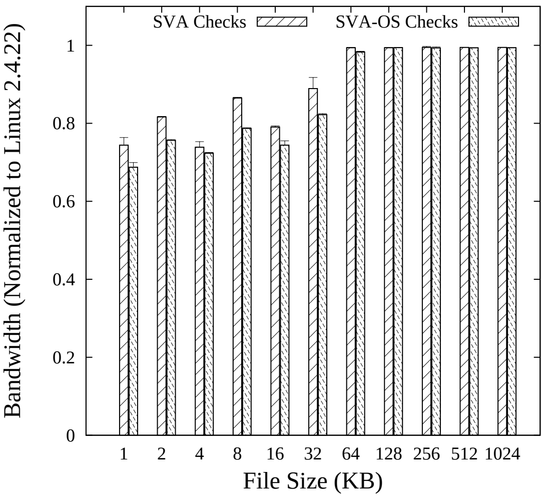

Figure 2: Web Server Bandwidth (Linux/i386 = 1.0)

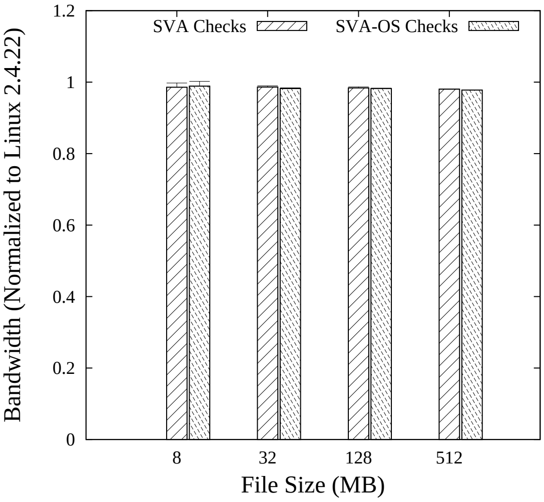

Figure 3: SSH Server Bandwidth (Linux/i386 = 1.0)

We also measured the performance of `sshd`, a login server offering encrypted file transfer. For this test, we measured the bandwidth of transferring several large files from the server to our test client; the results are shown in Figure 3. For each file size, we first did a priming run to bring file system data into the kernel's buffer cache; subsequently, we transfered the file three times. Figure 3 shows the mean of the receive bandwidth of the three runs normalized to the mean receive bandwidth mea-

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)	% Increase from i386 to SVA-OS	Description
bzip2	18.7 (0.47)	18.3 (0.47)	18.0 (0.00)	0.0%	Compressing 64 MB file
lame	133.3 (3.3)	132 (0.82)	126.0 (0.82)	-0.1%	Converting 206 MB WAV file to MP3
perl	22.3 (0.47)	22.3 (0.47)	22.3 (0.47)	0.0%	Interpreting scrabbl.pl from SPEC 2000

Table 3: Latency of Applications. Standard Deviation Shown in Parentheses.

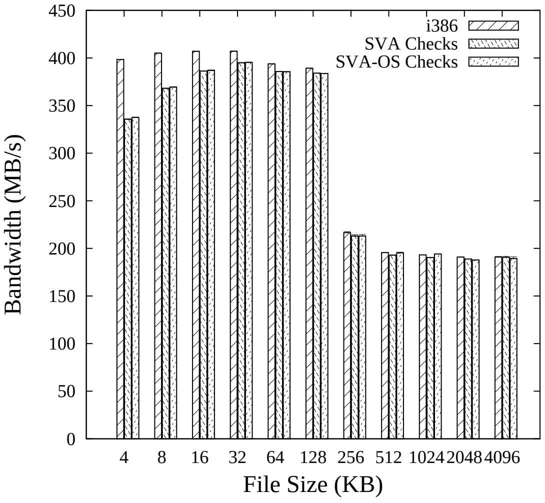

Figure 4: File System Bandwidth

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)
bzip2	41	40	40
lame	203	202	202
perl	24	23	23

Table 4: Latency of Applications During Priming Run.

sured on the original i386 kernel; note that the units on the X-axis are MB. Our results indicate that there is no significant decrease in bandwidth due to the extra run-time checks added by the original SVA system or the new run-time checks presented in this paper. This outcome is far better than thttpd, most likely due to the large file sizes we transfered via scp. For large file sizes, the network becomes the bottleneck: transferring an 8 MB file takes 62.5 ms on a Gigabit network, but the overheads for basic system calls (shown in Table 5) show overheads of only tens of microseconds.

To see what effect our system would have on end-user application performance, we ran experiments on the client-side programs listed in Table 3. We tested bzip2 compressing a 64 MB file, the LAME MP3 encoder converting a 206 MB file from WAV to MP3 format, and the perl interpreter running the training input from the SPEC 2000 benchmark suite. For each test, we ran the program once to prime any caches within the operating system and then ran each program three times. Table 3 shows the average of the execution times of the three runs and the percent overhead that the applications experienced executing on the SVA-OS kernel relative to the original i386 Linux kernel. The results show that our system adds virtually no overhead for these applications, even though some of the programs (bzip2 and lame) perform substantial amounts of I/O. Table 4 shows the latency of the applications during their priming runs; our kernel shows no overhead even when the kernel must initiate I/O to retrieve data off the physical disk.

Microbenchmark Performance To better understand the different performance behaviors of the applications, we used microbenchmarks to measure the overhead our system introduces for primitive kernel operations. For these experiments, we configured HBench-OS to run each test 50 times.

Our results for basic system calls (Table 5) indicate that the original SVA system adds significant overhead (on the order of tens of microseconds) to individual system calls. However, the results also show that our new safety checks only add a small amount of additional overhead (25% or less) to the original SVA system.

We also tested the file system bandwidth, shown in Figure 4. The results show that the original SVA system reduces file system bandwidth by about 5-20% for small files but that the overhead for larger files is negligible. Again, however, the additional checks for low-level kernel operations add no overhead.

The microbenchmark results provide a partial explanation for the application performance results. The applications in Table 3 experience no overhead because they perform most of their processing in user-space; the overhead of the kernel does not affect them much. In contrast, the sshd and thttpd servers spend most of their time executing in the kernel (primarily in the poll(), select(), and write() system calls). For the system calls that we tested, our new safety checks add less than several microseconds of overhead (as shown in Table 5). For a small network transfer of 1 KB (which takes less than 8 μs on a Gigabit network), such an overhead can affect performance. However, for larger files sizes (e.g., an 8 MB transfer that takes 62.5 ms), this overhead becomes negligible. This effect shows up in our results for networked applications (thttpd and sshd): smaller file transfers see significant overhead, but past a certain file size, the overhead from the run-time safety checks becomes negligible.

Benchmark	i386 (μs)	SVA (μs)	SVA-OS (μs)	% Increase from SVA to SVA-OS	Description
getpid	0.16 (0.001)	0.37 (0.000)	0.37 (0.006)	0.0%	Latency of getpid() syscall
openclose	1.10 (0.009)	11.1 (0.027)	12.1 (0.076)	9.0%	Latency of opening and closing a file
write	0.25 (0.001)	1.87 (0.012)	1.86 (0.010)	-0.4%	Latency of writing a single byte to /dev/null
signal handler	1.59 (0.006)	6.88 (0.044)	8.49 (0.074)	23%	Latency of calling a signal handler
signal install	0.34 (0.001)	1.56 (0.019)	1.95 (0.007)	25%	Latency of installing a signal handler
pipe latency	2.74 (0.014)	30.5 (0.188)	35.9 (0.267)	18%	Latency of ping-ponging one byte message between two processes
poll	1.16 (0.043)	6.47 (0.080)	7.03 (0.014)	8.7%	Latency of polling both ends of a pipe for reading and writing. Data is always available for reading.
select	1.00 (0.019)	8.18 (0.133)	8.81 (0.020)	7.7%	Latency of testing both ends of a pipe for reading and writing. Data is always available for reading.

Table 5: Latency of Kernel Operations. Standard Deviation Shown in Parentheses.

8 Related Work

Previous work has explored several approaches to providing greater safety and reliability for operating system kernels. Some require complete OS re-design, e.g., capability-based operating systems [37, 38] and microkernels [1, 25]. Others use isolation (or "sandboxing") techniques, including device driver isolation within the OS [35, 44, 45, 51] or the hypervisor [17]. While effective at increasing system reliability, none of these approaches provide the memory safety guarantees provided by our system, e.g., none of these prevent corruption of memory mapped I/O devices, unsafe context switching, or improper configuration of the MMU by either kernel or device driver code. In fact, none of these approaches could protect against the Linux exploits or device corruption cases described in Section 7. In contrast, our system offers protection from all of these problems for both driver code and core kernel code.

The EROS [38] and Coyotos [37] systems provide a form of safe (dynamic) typing for abstractions, e.g., capabilities, at their higher-level OS ("node and page") layer. This type safety is preserved throughout the design, even across I/O operations. The lower-level layer, which implements these abstractions, is written in C/C++ and is theoretically vulnerable to memory safety errors but is designed carefully to minimize them. The design techniques used here are extremely valuable but difficult to retrofit to commodity systems.

Some OSs written in type-safe languages, including JX [18], SPIN [21], Singularity [22], and others [20] provide abstractions that guarantee that loads and stores to I/O devices do not access main memory, and main memory accesses do not access I/O device memory. However, these systems either place context switching and MMU management within the virtual machine run-time (JX) or provide no guarantee that errors in these operations cannot compromise the safety guarantees of the language in which they are written.

Another approach that could provide some of the guarantees of our work is to add annotations to the C language. For example, SafeDrive's annotation system [51] could be extended to provide our I/O memory protections and perhaps some of our other safety guarantees. Such an approach, however, would likely require changes to every driver and kernel module, whereas our approach only requires a one-time port to the SVA instruction set and very minor changes to machine-independent parts of the kernel.

The Devil project [27] defines a safe interface to hardware devices that enforces safety properties. Devil could ensure that writes to the device's memory did not access kernel memory, but not vice versa. Our SVA extensions also protect I/O memory from kernel memory and provide comprehensive protection for other low-level hardware interactions, such as MMU changes, context switching, and thread management.

Mondrix [49] provides isolation between memory spaces within a kernel using a word-granularity memory isolation scheme implemented in hardware [48]. Because Mondrix enables much more fine-grained isolation (with acceptable overhead) than the software supported isolation schemes discussed earlier, it may be able to prevent some or all of the memory-related exploits we discuss. Nevertheless, it cannot protect against other errors such as control flow violations or stack manipulation.

A number of systems provide Dynamic Information Flow Tracking or "taint tracking" to enforce a wide range of security policies, including memory safety, but most of these have only reported results for user-space applications. Raksha [12] employed fine-grain information flow policies, supported by special hardware, to prevent buffer overflow attacks on the Linux kernel by ensuring that injected values weren't dereferenced as pointers. Unlike our work, it does not protect against attacks that inject non-pointer data nor does it prevent use-after-free errors of kernel stacks and other state buffers used in low-level kernel/hardware interaction. Furthermore, this system does not work on commodity hardware.

The CacheKernel [7] partitions its functionality into an application-specific OS layer and a common "cache kernel" that handles context-switching, memory mappings, etc. The CacheKernel does not aim to provide memory safety, but its two layers are conceptually similar to the commodity OS and the virtual machine in our approach. A key design difference, however, is that our interface also attempts to make kernel code easier to analyze. For example, state manipulation for interrupted programs is no longer an arbitrary set of loads/stores to memory but a single instruction with a semantic meaning.

Our system employs techniques from VMMs. The API provided by SVA for configuring the MMU securely is similar to that presented by para-virtualized hypervisors [16, 50]. However, unlike VMMs, our use of these mechanisms is to provide fine-grain protection internal to a single domain, including isolation between user and kernel space and protection of type-safe main memory, saved processor state, and the kernel stack. For example, hypervisors would not be able to guard against [42], which our system does prevent, even though it is an MMU error. Also, a hypervisor that uses binary rewriting internally, e.g., for instrumenting itself, could be vulnerable to [9], just as the Linux kernel was. We believe VMMs could be a useful *target* for our work.

SecVisor [36] is a hypervisor that ensures that only approved code is executed in the processor's privileged mode. In contrast, our system does not ensure that kernel code meets a set of requirements other than being memory safe. Unlike SVA, SecVisor does not ensure that the approved kernel code is memory safe.

9 Conclusion

In this paper, we have presented new mechanisms to ensure that low-level kernel operations such as processor state manipulation, stack management, memory mapped I/O, MMU updates, and self-modifying code do not violate the assumptions made by memory safety checkers. We implemented our design in the Secure Virtual Architecture (SVA) system, a safe execution environment for commodity operating systems, and its corresponding port of Linux 2.4.22. Only around 100 lines of code were added or changed to the SVA-ported Linux kernel for the new techniques. To our knowledge, this is the first paper that (i) describes a design to prevent bugs in low-level kernel operations from compromising memory safe operating systems, including operating systems written in safe or unsafe languages; and (ii) implements and evaluates a system that guards against such errors.

Our experiments show that the additional runtime checks add little overhead to the original SVA prototype and were able to catch multiple real-world exploits that would otherwise bypass the memory safety guarantees provided by the original SVA system. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

Acknowledgments

We wish to thank our shepherd, Trent Jaeger, and the anonymous reviewers for their helpful and insightful feedback.

References

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for unix development. In *Proc. USENIX Annual Technical Conference* (Atlanta, GA, USA, July 1986), pp. 93–113.

[2] APPLE COMPUTER, INC. Apple Mac OS X kernel semop local stack-based buffer overflow vulnerability, April 2005. http://www.securityfocus.com/bid/13225.

[3] ARCANGELI, A. Linux kernel mremap local privilege escalation vulnerability, May 2006. http://www.securityfocus.com/bid/18177.

[4] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Copper Mountain, CO, USA, December 1995), pp. 267–284.

[5] BOVET, D. P., AND CESATI, M. *Understanding the LINUX Kernel*, 2^{nd} ed. O'Reilly, Sebastopol, CA, 2003.

[6] BROWN, A. *A Decompositional Approach to Computer System Performance*. PhD thesis, Harvard College, April 1997.

[7] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.* (Monterey, CA, USA, November 1994), pp. 179–193.

[8] CORBET. SMP alternatives, December 2005. http://lwn.net/Articles/164121.

[9] CORBET, J. The source of the e1000e corruption bug, October 2008. http://lwn.net/Articles/304105.

[10] CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Stevenson, WA, USA, October 2007), pp. 351–366.

[11] CRISWELL, J., MONROE, B., AND ADVE, V. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture* (Boston, MA, USA, June 2006), pp. 26–33.

[12] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the USENIX Security Symposium* (San Jose, CA, USA, July 2008), pp. 395–410.

[13] DEVICES, A. M. AMD64 architecture programmer's manual volume 2: System programming, September 2006.

[14] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the Int'l Conf. on Software Engineering* (Shanghai, China, May 2006), pp. 162–171.

[15] DHURJATI, D., KOWSHIK, S., AND ADVE, V. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Ottawa, Canada, June 2006), pp. 144–157.

[16] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Bolton Landing, NY, USA, October 2003), pp. 164–177.

[17] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (Boston, MA, USA, October 2004).

[18] GOLM, M., FELSER, M., WAWERSICH, C., AND KLEINODER, J. The JX Operating System. In *Proc. USENIX Annual Technical Conference* (Monterey, CA, USA, June 2002), pp. 45–58.

[19] GUNINSKI, G. Linux kernel multiple local vulnerabilities, 2005. http://www.securityfocus.com/bid/11956.

[20] HALLGREN, T., JONES, M. P., LESLIE, R., AND TOLMACH, A. A principled approach to operating system construction in haskell. In *Proc. ACM SIGPLAN Int'l Conf. on Functional Programming* (Tallin, Estonia, September 2005), pp. 116–128.

[21] HSIEH, W., FIUCZYNSKI, M., GARRETT, C., SAVAGE, S., BECKER, D., AND BERSHAD, B. Language support for extensible operating systems. In *Workshop on Compiler Support for System Software* (Arizona, USA, February 1996).

[22] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FHNDRICH, M., HODSON, C. H. O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, October 2005.

[23] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization* (San Jose, CA, USA, March 2004), pp. 75–88.

[24] LATTNER, C., LENHARTH, A. D., AND ADVE, V. S. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA, June 2007), pp. 278–289.

[25] LIEDTKE, J. On micro-kernel construction. *SIGOPS Oper. Syst. Rev. 29*, 5 (1995), 237–250.

[26] LMH. Month of kernel bugs (MoKB) archive, 2006. http://projects.info-pull.com/mokb/.

[27] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: an IDL for hardware programming. In *USENIX Symposium on Operating System Deisgn and Implementation* (San Diego, CA, USA, October 2000), pp. 17–30.

[28] MICROSYSTEMS, S. Sun solaris sysinfo system call kernel memory reading vulnerability, October 2003. http://www.securityfocus.com/bid/8831.

[29] MONROE, B. M. Measuring and improving the performance of Linux on a virtual instruction set architecture. Master's thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Urbana, IL, Dec 2005.

[30] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* (2005).

[31] POSKANZE, J. thttpd - tiny/turbo/throttling http server, 2000. http://www.acme.com/software/thttpd.

[32] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium* (San Diego, CA, USA, February 2004), pp. 159–169.

[33] SAULPAUGH, T., AND MIRHO, C. *Inside the JavaOS Operating System.* Addison-Wesley, Reading, MA, USA, 1999.

[34] SCOTT, M. L. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2001.

[35] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating System Deisgn and Implementation* (Seattle, WA, October 1996), pp. 213–227.

[36] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev. 41*, 6 (2007), 335–350.

[37] SHAPIRO, J., DOERRIE, M. S., NORTHUP, E., SRIDHAR, S., AND MILLER, M. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification* (Sydney, Australia, October 2004).

[38] SHAPIRO, J. S., AND ADAMS, J. Design evolution of the EROS single-level store. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, June 2002), pp. 59–72.

[39] STARSETZ, P. Linux kernel do_mremap function vma limit local privilege escalation vulnerability, February 2004. http://www.securityfocus.com/bid/9686.

[40] STARZETZ, P. Linux kernel elf core dump local buffer overflow vulnerability. http://www.securityfocus.com/bid/13589.

[41] STARZETZ, P. Linux kernel IGMP multiple vulnerabilities, 2004. http://www.securityfocus.com/bid/11917.

[42] STARZETZ, P., AND PURCZYNSKI, W. Linux kernel do_mremap function boundary condition vulnerability, January 2004. http://www.securityfocus.com/bid/9356.

[43] STARZETZ, P., AND PURCZYNSKI, W. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. http://www.securityfocus.com/bid/10179.

[44] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst 23*, 1 (2005), 77–110.

[45] ÚLFAR ERLINGSSON, ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating System Deisgn and Implementation* (Seattle, WA, USA, November 2006), pp. 75–88.

[46] VAN SPRUNDEL, I. Linux kernel bluetooth signed buffer index vulnerability. http://www.securityfocus.com/bid/12911.

[47] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (1993), 203–216.

[48] WITCHEL, E., CATES, J., AND ASANOVIC., K. Mondrian memory protection. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, October 2002), pp. 304–316.

[49] WITCHEL, E., RHEE, J., AND ASANOVIC, K. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Brighton, UK, October 2005), pp. 31–44.

[50] WRIGHT, C. Para-virtualization interfaces, 2006. http://lwn.net/Articles/194340.

[51] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating System Deisgn and Implementation* (Seattle, WA, USA, November 2006), pp. 45–60.