Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems

John Criswell
University of Illinois at Urbana-Champaign
criswell@uiuc.edu

Andrew Lenharth
University of Illinois at Urbana-Champaign
alenhar2@uiuc.edu

Dinakar Dhurjati
DoCoMo Communications Laboratories, USA
dhurjati@docomolabs-usa.com

Vikram Adve
University of Illinois at Urbana-Champaign
vadve@uiuc.edu

ABSTRACT

This paper describes an efficient and robust approach to provide a *safe execution environment* for an entire operating system, such as Linux, and all its applications. The approach, which we call *Secure Virtual Architecture* (SVA), defines a virtual, low-level, typed instruction set suitable for executing *all* code on a system, including kernel and application code. SVA code is translated for execution by a virtual machine transparently, offline or online. SVA aims to enforce *fine-grained (object level) memory safety*, *control-flow integrity*, *type safety for a subset of objects*, and *sound analysis*. A virtual machine implementing SVA achieves these goals by using a novel approach that exploits properties of existing memory pools in the kernel and by preserving the kernel's explicit control over memory, including custom allocators and explicit deallocation. Furthermore, the safety properties can be encoded compactly as extensions to the SVA type system, allowing the (complex) safety checking compiler to be outside the trusted computing base. SVA also defines a set of OS interface operations that abstract all privileged hardware instructions, allowing the virtual machine to monitor all privileged operations and control the physical resources on a given hardware platform. We have ported the Linux kernel to SVA, treating it as a new architecture, and made only minimal code changes (less than 300 lines of code) to the machine-independent parts of the kernel and device drivers. SVA is able to prevent 4 out of 5 memory safety exploits previously reported for the Linux 2.4.22 kernel for which exploit code is available, and would prevent the fifth one simply by compiling an additional kernel library.

Categories and Subject Descriptors: D.4.6 [*Operating Systems*] Security and Protection; D.4.7 [*Operating Systems*] Organization and Design; D.3.4 [*Programming Languages*] Processors;

General Terms: Design, reliability, security

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010 ...$5.00.

Keywords: Operating systems, virtual machine, compiler, security, memory safety, type safety, typed assembly language

1. INTRODUCTION

Despite many advances in system security, most major operating systems remain plagued by security vulnerabilities. One major class of such vulnerabilities is *memory safety errors*, such as buffer overruns, double frees, and format string errors. Rapidly spreading malware like Internet worms, especially those that use "self-activation" [49], exploit such errors because such attacks can be carried out in large numbers of systems quickly and fully automatically. Weaver et al. estimate that buffer overflow and other attacks due to the use of C/C++ represent roughly 50% of all attacks in the last 20 years [49].

Safe programming languages, such as Java and C#, guarantee that such errors do not occur, but "commodity" operating systems like Linux, FreeBSD, Mac OS, Solaris and Windows (as well as security-sensitive software such as OpenSSH and Apache) are all written using C and C++, and hence enjoy none of the safety guarantees of safe languages.

Furthermore, a safe execution environment can *foster new avenues for innovation* in commodity systems. Several previous research projects have studied novel OS design strategies that exploit a safe execution environment, by using safe languages like Modula 3 (e.g.,. SPIN [5]), Java (e.g., JX, JavaOS, KaffeOS, and others [19, 36, 23, 9, 4]), C# (e.g., Singularity [24]), ML (ML/OS [17]), and Haskell (several experimental kernels [22]). For example, SPIN allows user programs to create untrusted, safe kernel extensions for higher performance. Singularity allows a single address space to be shared by user processes and the kernel and uses typed communication channels for interprocess communication [16]. Many such design innovations are fundamentally impractical without a safe execution environment.

A less obvious (but, in the long term, perhaps even more important) issue is that many higher-level security problems could be addressed effectively by a combination of compiler and run-time techniques (Section 9 lists some examples). The *compiler-based virtual machines*[1] underlying systems

[1] Throughout this paper, we use the term "*virtual machine*" to refer to a run-time system implementing a virtual instruction set architecture, such as the Java Virtual Machine. This differs from the literature on hypervisors, in which the term *virtual machine* refers to a single instance of a guest OS.

like JavaOS, JX, or Singularity make it possible to apply sophisticated compiler techniques on the OS and its applications. Most practical security research, however, focuses on systems in widespread use rather than novel experimental operating systems.

All three of the above goals – memory safety, novel design opportunities, and more powerful security solutions – can be enabled with a single new (but very ambitious) strategy: *using a safe, compiler-based virtual machine capable of hosting a complete commodity operating system and security-sensitive system software*. Such a VM would provide a safe execution environment for most or all of the kernel and system software, largely eliminating memory safety errors and enabling some of the novel OS design ideas to be incorporated. Also, if a practical, compiler-based virtual machine is available to host a commodity OS, security researchers could begin to explore novel approaches for a wide range of challenging security problems, combining the compiler capabilities with run-time monitoring in the OS or the VM.

There have been several previous approaches for enforcing various safety guarantees for commodity OS code, as discussed in Section 8. On the one hand, approaches like Software Fault Isolation [47] or XFI [45] enforce isolation between coarse-grained components but do not provide "fine-grained safety guarantees" (e.g., for individual buffers, pointers, or heap objects) needed for a safe execution environment. At the other extreme, language-based approaches such as Cyclone [20, 6] and Deputy [50] provide a safe execution environment but have only been applied to specific components, appear difficult to extend to a complete kernel, and require significant changes or annotations to existing kernel code.

In this paper, we describe a *Secure Virtual Architecture* and its implementation, both designed to support modern operating systems efficiently and with relatively little change to the guest OS. SVA provides a safe execution environment for a kernel and (selected) application code. Porting a kernel to SVA requires no major design changes; it is similar to, but significantly simpler than, porting to a new hardware architecture because SVA provides simple abstractions for the privileged operations performed by a kernel. Furthermore, the compilation and execution process is largely transparent: most application or kernel developers would see no change when running SVA code (unless they inspect object code).

The safety guarantees SVA provides, listed in detail in Section 4.9, include *memory safety*, *control-flow integrity*, *type safety for a subset of objects*, and support for *sound program analysis*. These guarantees are close to, but slightly weaker than, the safety guarantees provided by a safe language like Java, C#, or Modula-3. There are two essential weaknesses which occur because we preserve the low-level memory model of C: (a) dangling pointer references can occur but are rendered harmless, i.e., they cannot violate the safety guarantees (although they may still represent potential logical errors); and (b) arbitrary, explicit casts are permitted, as in C. (In fact, we support the full generality of C code with no required source changes.) These compromises allow us to achieve a fairly strong degree of safety for commodity OSs while minimizing kernel porting effort.

SVA is also designed to ensure that the (relatively complex) safety checking compiler does not need to be a part of the trusted computing base. The compiler can generate code in the SVA virtual instruction set, and a simple type checker can ensure that the code meets the safety requirements of SVA. This provides a *robust* implementation strategy for enforcing the safety properties of SVA. Furthermore, higher-level security properties (e.g., information flow [35] or security automata [48]) expressible as types can be encoded compactly in SVA code, enabling robust implementations of sophisticated security-checking strategies.

The next section gives an overview of the SVA design. Section 3 describes the virtual instruction set and the kernel boot and execution process. Section 4 explains the approach to enforcing safety in SVA. Section 5 explains the SVA type system and type checker, which help minimize the SVA trusted computing base. Section 6 describes how we ported the Linux kernel to SVA. Section 7 presents our experimental evaluation, including performance overheads, effectiveness at catching previously reported kernel exploits, and static metrics on the effectiveness of the safety checking compiler. Finally, Section 8 compares SVA with previous work, and Section 9 concludes with a brief summary and our goals for ongoing and future work.

2. OVERVIEW OF THE SVA APPROACH

The broad goals of the SVA project are to provide a safe execution environment for commodity operating systems and the security-critical programs that run on them, and (in future work) to incorporate novel solutions to higher-level security problems that combine OS, virtual machine, and compiler techniques. This paper focuses on the first goal. We aim to achieve this goal under two major constraints: (a) have only a small impact on performance, and (b) require small porting effort and few OS design changes on the part of OS developers.

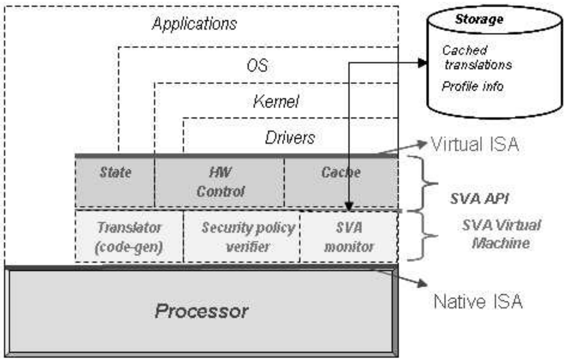

Figure 1: System Organization with SVA

Figure 1 shows the system organization with SVA. Briefly, SVA defines a virtual, low-level, typed instruction set suitable for executing *all* code on a system, including kernel and application code. Part of the instruction set is a set of operations (collectively called SVA-OS) that encapsulate all the privileged hardware operations needed by a kernel. SVA also defines a few constraints that kernel memory allocators must satisfy and a few operations they must implement. Safety properties such as memory safety and type safety (and in the future, also higher-level security properties, such as some of those listed in Section 9) are encoded as extensions of this instruction set and its type system. The system implementing the virtual instruction set is responsible for verifying that an executable "*bytecode*" (i.e., virtual object code) file satisfies the desired safety properties. We refer to this system as a Secure Virtual Machine (SVM) and the virtual object code that it executes as "bytecode."

The SVA design builds on two key technologies from our previous research: the *LLVM virtual instruction set definition* (which supports a sophisticated and widely-used compiler infrastructure) [27, 1], and the *SAFECode principles for efficient safety enforcement for unmodified C code* [11, 12]. These are described briefly in Sections 3.2 and 4.1. SVA extends the LLVM virtual instruction set to add OS support operations, as described in Section 3. SVA extends the safety strategy in SAFECode to work in a full-fledged commodity kernel, as described in Section 4.

Porting an existing kernel to SVA includes three steps. First, the target-dependent layer of the kernel is ported to use the SVA-OS interface. This is similar to porting to a new architecture, but potentially simpler because the SVA-OS operations are slightly higher-level and more abstract. At the end of this step, there will be *no explicit assembly code* in the kernel. Second, a few well-defined changes are required in kernel memory allocators, to implement two simple constraints on object alignment and memory reuse. SVA requires no other changes to the allocation strategies used within these allocators. More importantly, except for these allocator changes, the target-independent code of a kernel needs no other general class of changes for an initial correct port to SVA (there can be system-specific design inconsistencies that must be fixed, as we found for the signal-handler dispatch mechanism in Linux). Finally, some optional improvements to kernel code can significantly improve the analysis precision and hence the safety guarantees provided by SVA. Sections 3.3, 4.4 and 6.3 describe these three kinds of changes in general terms, and Section 6 describes the specific changes we made to port the Linux kernel to SVA.

The steps by which kernel or secure application code is compiled and executed on SVA are as follows. A front-end compiler translates source code to SVA bytecode. A *safety checking compiler* (which may be part of the front end or separate) transforms kernel or application bytecode to implement the safety properties we provide. This code is then shipped to end-user systems. At install time or load time, a *bytecode verifier* checks the safety properties of the bytecode and a *translator* transparently converts the bytecode into native code for a particular processor. The verification and translation process can happen offline (to maximize performance) or online (to enable additional functionality). When translation is done offline, the translated native code is cached on disk together with the bytecode, and the pair is digitally signed *together* to ensure integrity and safety of the native code. In either case, kernel modules and device drivers can be dynamically loaded and unloaded (if the kernel supports it) because both the bytecode verifier and translator are intraprocedural and hence modular (unlike the safety-checking compiler). Moreover, modules can be dynamically loaded whether they were compiled with the safety checking compiler or were left as external "unknown" code; obviously, including them improves the SVA safety guarantees as described in Section 4.

An important issue that we do not address in this paper is kernel recovery after a safety error is detected at runtime. Several systems, including Vino [37], Nooks [42, 44] and SafeDrive [50], provide strategies for recovering a commodity kernel from a fault due to a device driver or other extension. SVA does not yet include any specific mechanisms for recovery, and investigating recovery mechanisms (including these existing techniques, which should be compatible with SVA) is a subject of future work.

3. THE SVA INSTRUCTION SET AND EXECUTION

The SVA virtual instruction set has two parts: the core computational instructions (SVA-Core), which are used by all software running on SVA for computation, control flow, and memory access, and the OS support operations (SVA-OS). Below, we describe the virtual instruction set and the execution strategy for a kernel running on the SVA virtual machine (SVM).

3.1 Instruction Set Characteristics

The SVA-Core instructions and the SVA object file format are inherited directly from the LLVM compiler infrastructure [27]. LLVM (and hence, SVM) defines a single, compact, typed instruction set that is used both as the in-memory compiler internal representation (IR) and the external, on-disk "bytecode" representation. This is a simple, RISC-like, load-store instruction set on which detailed optimizations can be expressed directly, like a machine language, but unlike a machine language, it also enables sophisticated analysis and transformation. There are four major differences from native hardware instruction sets. First, memory is partitioned into code (a set of functions), globals, stack, and other memory. Second, every function has an explicit control flow graph with no computed branches. Third, the instruction set uses an "infinite" virtual register set in Static Single Assignment (SSA) form, making many dataflow analyses simple and powerful. Fourth, the instruction set is typed, and all instructions are type-checked. The type system enables advanced techniques for pointer analysis and array dependence analysis. Unsafe languages like C and C++ are supported via explicit cast instructions, a detailed low-level memory model, and explicit control over memory allocation and deallocation.

An SVA object file (called a *Module*) includes functions, global variables, type and external function declarations, and symbol table entries. Because this code representation can be analyzed and transformed directly, it simplifies the use of sophisticated compiler techniques at compile-time, link-time, load-time, run-time, or "idle-time" between program runs [27]. In fact, both the safety checking compiler and the bytecode verifier operate on the same code representation. Furthermore, because the bytecode language is also designed for efficient just-in-time code generation,[2] the bytecode verification and translation steps can easily be done at load-time for dynamically loaded modules.

3.2 SVA-Core Instructions

The SVA-Core instruction set includes instructions for ordinary arithmetic and logical operations, comparisons that produce a boolean value, explicit conditional and unconditional branches, typed indexing into structures and arrays, loads and stores, function calls, and both heap and stack memory allocation and deallocation. Heap objects can be

[2] The underlying LLVM optimizer and JIT are robust and efficient enough to be used in production systems like Mac OS X.

Name	Description
sva.save.integer(void * buffer)	Save the Integer State of the native processor in to the memory pointed to by *buffer*.
sva.load.integer(void * buffer)	Load the Integer State stored in *buffer* back on to the processor. Execution resumes at the instruction immediately following the sva.save.integer() instruction that saved the state.
sva.save.fp(void * buffer, int always)	Save the FP State of the native processor or FPU to the memory pointed to by *buffer*. If *always* is 0, state is only saved if it has changed since the last sva.load.fp(). Otherwise, save state unconditionally.
sva.load.fp(void * buffer)	Load the FP State of the native processor (or FPU) from a memory buffer previously used by sva.save.fp().

Table 1: Functions for Saving and Restoring Native Processor State

Name	Description
sva.icontext.save(void* icp, void* isp)	Save Interrupt Context *icp* into memory pointed to by *isp* as Integer State.
sva.icontext.load(void* icp, void* isp)	Load Integer State *isp* into Interrupt Context pointed to by *icp*.
sva.icontext.commit(void* icp)	Commit the entire Interrupt Context *icp* to memory.
sva.ipush.func(void* icp, int (*f)(...), ...)	Modify the state in Interrupt Context *icp* so that function *f* has been called with the given arguments. Used in signal handler dispatch.
sva.was.privileged(void* icp)	Return 1 if Interrupt Context *icp* was running in privileged mode; else, return 0.

Table 2: Functions for Manipulating the Interrupt Context

allocated using explicit `malloc` and `free` instructions that are (typically) lowered to call the corresponding standard library functions; however, some heap objects may be allocated via other functions (e.g., custom allocators) that may not be obvious to the compiler. The instruction set also supports fundamental mechanisms for implementing exceptions (including C++ exceptions and C's `setjmp/longjmp`), and multithreaded code.

We extended the LLVM instruction set with atomic memory access instructions (atomic load-increment-store and compare-and-swap) and a memory write barrier instruction for the SVA-Core instruction set. These instructions can support an OS kernel, multi-threaded userspace applications, and libraries.

3.3 SVA-OS Instructions

We developed a set of extensions (SVA-OS) to the core instruction set (exposed as an API) to provide low-level mechanisms that are normally implemented in a kernel using hand-written assembly code [8]. An important design choice is that SVA-OS provides only *mechanisms*, not policies; the policies are left to the OS kernel. These operations allow the SVM to monitor and control privileged OS operations as *all such operations* are performed by the SVM. These instructions must provide two critical services to the OS. First, the OS needs mechanisms to manipulate privileged hardware resources such as the page and interrupt vector tables. Second, the OS requires instructions to efficiently save, restore, and manipulate the processor state.

Privileged hardware operations, such as I/O functions, MMU configuration functions, and the registration of interrupt and system call handlers, are straightforward (and are not shown here). For every such operation, we added a new SVA-OS function to provide the necessary functionality.

The state manipulation instructions require some novel techniques. Ideally, the OS and other external tools should save, restore and manipulate state in terms of objects visible at the virtual instruction set level (such as the virtual registers). Supporting such functionality is necessary (e.g., for debugging) but would be very inefficient and is unnecessary for critical path operations like context-switching or signal delivery. Our solution is to allow the OS to save and restore native processor state directly (but opaquely) using a set of high level, well defined functions (as shown in Table 1).

We divided the opaque native state into two components. The *control state* consists of a program's control registers (including privileged registers) and general-purpose registers. The *floating point (FP) state* includes all floating pointer registers. There are separate instructions for saving and loading control state and FP state. Since many OS kernels and programs do not use FP state, it can be saved lazily so that the critical paths (e.g., context switching) need not be lengthened by always saving the floating point state of the processor.

On an interrupt or system call, the SVM must save native state to memory before handing control over to the kernel. SVM is able to take advantage of processor features for low latency interrupts (e.g. shadow registers) by creating an *interrupt context*. This abstraction represents the interrupted control state of the processor. On entry to the kernel, the SVM saves the subset of the control state that will be overwritten by the OS kernel on the kernel stack; all other control state is left on the processor. The OS kernel is then given a handle to the interrupt context which it can manipulate using the instructions in Table 2. In most cases, the OS kernel need not commit the entire interrupt context to memory; however, it can use the `sva.icontext.save` or `sva.icontext.commit` instructions for those cases in which it does (e.g., context switching, signal delivery).

3.4 The SVA Boot and Execution Strategy

The Secure Virtual Machine (SVM) implements SVA by performing bytecode verification, translation, native code caching and authentication, and implementing the SVA-OS instructions on a particular hardware architecture. Section 2 briefly discussed the high level steps by which a SVA kernel is loaded and executed. Here, we describe some additional details of the process.

On system boot, a native code boot loader will load both the SVM and OS kernel bytecode and then transfer control to the SVM. The SVM will then begin execution of the operating system, either by interpreting its bytecode or translating its bytecode to native code. For handling application code during normal operation, the SVM can use callbacks into the operating system to retrieve cached native code translations. These translations can be cryptographically signed to ensure that they have not been modified.

Since the SVM translates all code on the system (including the OS kernel), it is able to exercise a great degree of control over software execution. It can inline reference monitors [15] during code generation; it can choose to execute software in less privileged processor modes (similar to hypervisors like Xen [13]). Using either or both of these techniques allows the SVM to mediate access to privileged processor resources (like the MMU) and to enforce a wide range of policies.

In our design, we aim to run both SVM code and kernel code at the highest privilege level (e.g., level 0 on x86). Our system is designed assuming the safety checking techniques in Section 4 are used to prevent the kernel from corrupting SVM data. Our current work does this for the SVA-Core instructions; we will address the SVA-OS instructions in future work. Designers of a security-critical system may choose to add an extra layer of defense and run kernel code at a lower privilege level than SVM, similar to hypervisors like VMWare or Xen. The required use of a trap to change privilege level when making a "hypercall" to SVM would add some additional overhead but should not require any other significant change to the approach.

The SVM uses two methods to obtain memory for its own execution. First, it reserves a portion of physical memory for its initial use ("bootstrap"); this memory is used for code and internal data needed during system boot before the OS kernel has become active. The amount of memory needed for bootstrap is fixed and statically determinable; the current version of SVM reserves about 20KB. Second, during normal system operation, the SVM uses callback functions provided by the OS to obtain memory from and release memory to the OS. Since the SVM mediates all memory mappings, it can ensure that the memory pages given to it by the OS kernel are not accessible from the kernel or any other program on the system.

4. ENFORCING SAFETY FOR KERNEL CODE

In this section, we first give some brief background on SAFECode, our previous work on enforcing fine grained safety properties for stand-alone C programs. (SVA uses SAFECode directly to provide a safe execution environment for stand-alone programs.) We then identify three major challenges that arise when we apply the safety principles in SAFECode to a commodity kernel. In the subsequent sections, we present our solution to each of the three challenges. We end by summarizing concretely the specific safety guarantees SVA provides for kernel code.

4.1 Background: The SAFECode Approach for Enforcing Safety for C Programs

The SAFECode compiler and run-time system together enforce the following safety properties for a *complete, standalone C program with no manufactured addresses* [11, 12, 10]:

(T1) *Control-flow integrity*: A program will never execute an instruction sequence that violates the compiler-computed control flow graphs and call graph.

(T2) *Type safety for a subset of objects*: All objects in type-homogeneous partitions (defined below) are accessed or indexed according to their compiler-inferred type, or arrays of that type.

(T3) *Array bounds safety*: All accesses to array objects (including strings) fall within the array bounds.

(T4) *No uninitialized pointer dereferences*: All successful loads, stores, or calls use correctly initialized pointers.

(T5) *No double or illegal frees*: Dynamic deallocation operations will only be performed for *live* (not previously deallocated) objects with a legal pointer to the start of the allocated object.

(T6) *Sound analysis*: A sound operational semantics [11] is enforced, incorporating a flow-insensitive points-to graph, call graph, and a subset of type information, usable by compilers and static analysis tools to implement *sound* higher-level analyses.

Note that dangling pointer *dereferences* (i.e., *read or write after free*) are *not* prevented. Nevertheless, SAFECode ensures that the other guarantees are not violated.

SAFECode enforces these properties through a combination of program analysis and run-time checks, with *no* changes to source code of programs and without preventing the explicit deallocation of objects. Briefly, the SAFECode principles are as follows.

The compiler is given (or computes) a call graph, a "points-to graph" representing a static *partition* of all the memory objects in the available part of the program [29], and type information for a subset of the partitions as explained below. It automatically transforms the program (using an algorithm called Automatic Pool Allocation [28]) so that memory objects in distinct partitions (nodes in the points-to graph) are allocated in different logical "pools" of memory. Individual object allocation and deallocation operations occur at exactly the same locations as the original program but use the appropriate pool. Stack and global objects are registered with the pool to which they were assigned by the pointer analysis, and stack objects are deregistered when returning from the parent function. A key property SAFECode exploits is that, with a suitable pointer analysis, many partitions (and therefore the corresponding run-time pools) are *type-homogeneous* (TH), i.e., all objects allocated in the pool are of a single (known) type or are arrays of that type.

The fundamental sources of memory safety violations in C programs include *uninitialized variable references*, *array bounds violations* (including format string errors), *dangling pointer references*, and a variety of *illegal type casts* (including improper arguments to function calls). SAFECode prevents these violations and enforces the guarantees above as follows:

- It prevents *uninitialized variable references* via dataflow analysis (for local variables) and via initialization of allocated memory to ensure a hardware-detected fault on dereferences (for all other pointer variables).

- It prevents *array bounds violations* using an extension [10] of the Jones-Kelly approach for detecting array bounds violations [26]. This extension uses a separate run-time search tree (a splay tree) in each pool to record all array objects at run-time, and looks up pointer values in this table to check for bounds violations. This strategy allows SAFECode to avoid using "fat pointers" for tracking array bounds at run-time; fat pointers, when used inside structures or variables that are accessible from library functions, are known to cause significant compatibility problems [32, 25, 3].

- Simple compile-time type-checking is sufficient to ensure *type safety of objects in type-homogeneous pools*. Dangling pointers to these objects cannot compromise type safety *as long as the run-time allocator does not release memory of one pool to be used by any other pool, until the first pool is "dead"*; in practice, SAFECode releases memory of a pool only when the pool is unreachable [12]. For pointers to objects in non-TH pools, run-time "pool bounds checks" at dereference ensure the pointer target lies within the pool.[3] Dangling pointers to stack frames are prevented by promoting stack objects into heap objects where needed.

- SAFECode enforces *control-flow integrity* by generating native code itself (preventing illegal branches to data areas), preventing writes to code pages, and using run-time checks to ensure that indirect function calls match the call targets computed by the compiler's call graph analysis.

- Finally, the *soundness of the operational semantics*, which requires correctness of the given analysis information (i.e., the call graph, points-to graph, and type information for TH partitions) follows directly from the previous safety properties[11].

Partitioning memory into logical pools corresponding to the pointer analysis has several critical benefits for the SAFECode approach:

- Type-homogeneity directly gives type safety for some objects.

- Type-homogeneous pools and run-time checks for non-TH pools together make dangling pointers harmless.

- No run-time checks are needed on dereferences of a pointer between TH pools.

- Using a separate splay tree per pool and eliminating scalar objects from the splay tree in TH pools make array bounds checking orders-of-magnitude faster than the original Jones-Kelly method. In fact, it is enough to make this approach practical [10].

Note that the SAFECode guarantees are weaker than a safe language like Java or C#. In particular, SAFECode does not prevent or detect dangling pointers to freed memory (safe languages prevent these by using automatic memory management) and permits flexible pointer casts at the cost of type safety for a subset of memory. The SAFECode approach provides a useful middle ground between completely safe languages and unsafe ones like C/C++ as it does not impose much performance overhead, does not require automatic memory management, is able to detect all types of memory safety errors other than dangling pointer uses, and is applicable to the large amount of legacy C/C++ code.

4.2 SAFECode for a Kernel: Challenges

We use the SAFECode compiler directly in SVA for standalone programs. Extending the SAFECode safety principles from standalone programs to a commodity kernel, however, raises several major challenges:

- The custom allocators used by kernels to manage both correctness and performance during memory allocation are incompatible with the pool allocation strategy that is fundamental to SAFECode.

- Unlike standalone programs, a kernel contains many entry and exit points from or to external code, and many of these bring pointers into and out of the kernel.

- Unlike most applications, a kernel typically uses a number of "manufactured addresses," i.e., where a predefined integer value is used as an address, and these can be difficult to distinguish from illegal addresses.

In the following subsections, we present our solution to these challenges. Preserving custom allocators is, by far, the most difficult issue and is a major focus of this section.

4.3 Integrating Safety Checking with Kernel Allocators

The SAFECode approach critically depends on a specific custom allocation strategy, namely, pool allocation with pools partitioned according to a pointer analysis. Kernels, however, make extensive use of custom allocators, e.g., `_alloc_bootmem`, `kmem_cache_alloc`, `kmalloc` and `vmalloc` in Linux, or the `zalloc`, `kalloc` and `IOMalloc` families of allocators in the Darwin kernel of Mac OS X. Running SAFECode directly on a kernel (even by identifying the kernel allocators to the compiler) is impractical because the compiler-generated allocation strategy would not work: the kernel allocators ensure many complex *correctness* requirements, e.g., pinning memory, alignment requirements, virtual memory mappings, and other issues.

The key insight underlying our solution is that *a kernel typically already uses pool allocation and furthermore, many of the "pools" are type-homogeneous*: distinct pools are created for different kinds of heavily used kernel objects. For example, a Darwin reference gives a partial list of 27 different data types (or classes of data types) for which a separate "zone" per data type is created using the zone allocator. In Linux, at least 37 different "caches" are created using the `kmem_cache_create` operation (not including the non-type-homogeneous caches used within the more generic `kmalloc` allocator). This suggests that if we can map existing kernel pools with pointer-analysis partitions, we may be able to achieve the safety benefits we seek without changing the kernel's custom allocators. We do so as follows.

First, as part of the porting process, kernel developers must identify the allocation routines to the compiler and specify which ones should be considered "pool allocators";

[3] We could instead perform more precise "object bounds" checks, using the same search trees as for array bounds violations. That is what we do in SVA.

```
1   MetaPool MP1, MP2;
2
3   struct fib_info * fib_create_info(
4     const struct rtmsg *r, struct kern_rta *rta,
5     const struct nlmsghdr *nlh, int *errp) {
6   ...
7     //look up object bounds and then check the access
8     getBounds(MP1, &fib_props, &s, &e);
9     boundscheck(s, &fib_props[r->rtm_type].scope, e)
10    if (fib_props[r->rtm_type].scope > r->rtm_scope)
11       goto err_inval;
12  ...
13    fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct
14                    fib_nh), GFP_KERNEL);
15    pchk_reg_obj(MP2, fi, 96, NULL, SVA_KMALLOC);
16  ...
17    //check bounds for memset without lookup since
18    //we know the start and size from the kmalloc
19    boundscheck(fi, (char*)fi + 95, (char*)fi + 96);
20    memset(fi,0,sizeof(*fi)+nhs*sizeof(...));
21
22    //check that rta is a valid object
23    lscheck(MP1, rta);
24    if (rta->rta_priority) {
25       //check that rta->rta_priority is valid
26       temp = rta->priority;
27       lscheck(MP2, temp);
28       fi->fib_priority = *temp;
29    }
30  ...
31  }
```

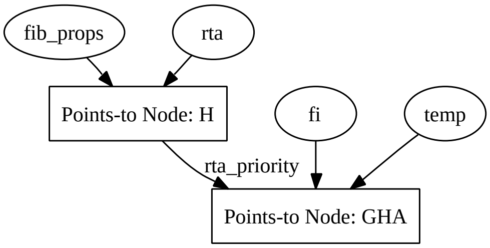

Figure 2: Example code fragment from the Linux kernel, and a part of the points-to graph for it. Square boxes are nodes representing memory objects. Ovals represent virtual registers.

the rest are treated as ordinary allocators. The existing interface to both kinds of allocators are not modified. For each allocator, routines must be specified for object allocation and deallocation and, for pool allocators, the routines for creation, initialization and destruction. The specific requirements for porting kernel allocators are described in Section 4.4.

The primary goal of the analysis, then, is to correlate kernel pools and other kernel objects with the static partitions of memory objects (i.e., nodes in the points-to graph) computed by pointer analysis. The compiler does *not* use the Automatic Pool Allocation transformation to partition memory into pools.

The output of pointer analysis is a points-to graph in which each node represents a distinct partition of objects in the analyzed part of the program [29]. An assumption in both SAFECode and SVA is that the pointer analysis is a "unification-style" algorithm [41], which essentially implies that every pointer variable in the program points to a unique node in the points-to graph. A single node may represent objects of multiple memory classes including *Heap*, *Stack*, *Global*, *Function* or *Unknown*. Figure 2 shows a part of a points-to graph for a code fragment from the Linux kernel. The `G` and `H` flags on the node pointed to by `fi` indicate that the node includes global objects as well as heap objects such as the one allocated with `kmalloc` at line 13.

Given the points-to graph, the compiler creates a global *metapool* variable for each node in the graph, e.g., `MP1` and `MP2` in the example. A metapool is simply a set of data objects that map to the same points-to node and so must be treated as one logical pool by the safety checking algorithm. Metapool variables, like the pool descriptors in SAFECode [11], are used as run-time representations of each points-to graph partition, recording run-time information ("metadata") about objects to enable run-time checks. They are also represented as types on pointer variables to be checked by the bytecode verifier. Using a global variable for each metapool avoids the need to pass metapool variables between functions.

At every heap allocation point, identified using the kernel allocator functions specified during porting, we insert a `pchk.reg.obj` operation to register the allocated object in the appropriate metapool (the one mapped to the pointer variable in which the allocation result is stored), e.g., at line 15 of Figure 2. Similarly, the compiler inserts a `pchk.drop.obj` operation at every deallocation point. These two operation signatures are shown in Table 3.

The compiler also inserts `pchk.reg.obj` operations to register all global and stack-allocated objects, and `pchk.drop.obj` to drop stack objects at function exits. Global object registrations are inserted in the kernel "entry" function, where control first enters during the boot process. Stack-allocated objects that may have reachable pointers after the parent function returns (which can be identified directly from the points-to graph) are converted to be heap allocated using a kernel-specified allocator, and deallocated at function return. This tolerates dangling pointers in the same way as heap objects: by controlling the reuse within TH pools and performing run-time checks within non-TH pools.

One key requirement for achieving our guarantees is that all memory managed as a single pool of memory (i.e., with internal reuse) must be registered in a single metapool: if it was spread across multiple metapools, a dangling pointer from one metapool could point to a different metapool, which could violate both type-safety (if one metapool is TH and any of the other metapools is of a different type) and sound pointer analysis (since each metapool represents a distinct node in the points-to graph). If a single kernel pool (e.g., a single `kmem_cache_t` object in Linux) maps to two or more partitions, we merge the points-to graph nodes for those partitions into a single graph node (effectively making the points-to analysis less precise, but still correct). Note that the converse – multiple kernel pools in a single metapool – needs no special handling for achieving our guarantees. (We only need to deregister all "remaining" objects that are in a kernel pool when a pool is destroyed.)

For the same reason, for ordinary allocators (e.g., `kmalloc` in Linux), all the memory managed by the allocator has to be treated as a single metapool because it may have full internal reuse. We effectively must merge all metapools (and hence all points-to graph nodes) that represent objects with a particular ordinary allocator. In some cases, however, an ordinary allocator is internally implemented as a default version of a pool allocator, e.g., `kmalloc` internally just uses `kmem_cache_alloc`. By exposing that relationship, as explained in Section 6, the kernel developer can reduce the need for unnecessary merging.

At this point, the safety-checking compiler has successfully mapped kernel pools and all pointer variables (including pointers to globals and stack allocated objects) to metapool variables. The compiler finally encodes the list of metapools and these mappings as type attributes on the SVA bytecode. This bytecode will later be presented to the verifier, which type-checks the program (as described in Section 5), and then inserts the necessary run-time checks, as described below. The specific ways in which we applied the overall strategy above to the Linux kernel is described in Section 6.

4.4 Kernel Allocator Changes

The kernel's allocators must fulfill several responsibilities in order to support the run-time checks needed for our memory safety strategy:

- The kernel source code must identify which allocators can be treated as pool allocators, and declare the allocation and deallocation functions used for each distinct allocator. The kernel must also provide an ordinary (non-pool) allocation interface that is available *throughout* a kernel's lifetime for stack-to-heap promotion. Internally, this interface could be implemented to use distinct allocators at different times (e.g., during boot vs. normal operation).

- Each allocator must provide a function that returns the size of an allocation given the arguments to the allocation function. This allows the compiler to insert `pchk.reg.obj` operations with the correct size.

- A type-homogeneous pool allocator must allocate objects aligned at the type size (or integer multiples thereof) to ensure that references using a dangling pointer do not cause a type conversion when accessing a newly allocated object.

- A kernel pool allocator must not release freed memory back for use by other pools, though it can reuse memory internally (technically, it can also release memory to other pools within the same metapool, though we do not currently provide such a mechanism).

- Kernel allocators must initialize memory to zero on each allocation (the first page of virtual memory should be unmapped to generate a fault if accessed).

Except for the third and fourth restrictions – on object alignment and memory release – there are no other changes to the allocation strategy or to the internal metadata used by the allocator. Note that these porting requirements are difficult to verify, i.e., SVA trusts the kernel developer to perform these changes correctly. However, performing all object registration and deregistration under compiler control and avoiding adding any metadata to the kernel allocators reduces the level of trust placed on the kernel developer.

4.5 Run-time Checks

The SVM verifier is responsible for inserting run-time checks into the kernel bytecode, logically as a part of type-checking the code. The run-time checks work as follows.

Each metapool maintains a splay tree to record the ranges of all registered objects. The run-time checks use these splay trees to identify legal objects and their bounds. The run-time checks SVM performs are:

1. *Bounds Check*: A bounds check must be performed for any array indexing operation that cannot be proven safe at compile-time, e.g., the checks at lines 9 and 19 in Figure 2. In the SVA instruction set, all indexing calculations are performed by the `getelementptr` instruction which takes a source pointer and a list of indexes and calculates a new address based upon the index list and the source pointer's type. A `boundscheck` operation verifies that the source and destination pointer belong to the same object within the correct metapool.

   If the SVM verifier can determine the bounds expressions for the target object of the source pointer, those bounds can be used directly, as at line 19 in the example. Otherwise, the verifier must insert a `getBounds` operation to verify that the object is in the correct metapool and then use the fetched bounds of that object for the check.

   Bounds checks should be performed on structure indexing operations. However, since these operations are usually safe and the performance overheads often outweigh any safety benefits gained, we have chosen not to perform structure indexing checks in our prototype.

2. *Load-store check*: A check must be performed on any load or store through a pointer obtained from a non-type-homogeneous metapool since such pointer values may come from arbitrary type casts instead of through legal, verified pointer operations. The `lscheck` operation is used to verify that the pointer points to a legal object within the correct metapool. Lines 23 and 27 show two examples of such checks.

3. *Indirect Call Check*: An indirect call check verifies that the actual callee is one of the functions predicted by the call graph by checking against the set of such functions. As with load-store checks, this check is not needed if the function pointer is obtained from a type-homogeneous pool because *all* writes to such pools have been detected (including any possible writes through dangling pointers).

4. *Illegal Free Check*: An illegal free check verifies that the argument to a kernel deallocator was allocated by the corresponding kernel allocator. This check is integrated with the `pchk.drop.obj` operation.

One complication is that some partitions may be exposed to external code that is not compiled by the safety checking compiler; the pointer analysis marks these partitions "Incomplete" [29]. Incomplete partitions may include objects not allocated within the available kernel code and, therefore, not registered with the corresponding metapool. This forces the SVM to be conservative in performing run-time checks. First, load-store, indirect call, and illegal free checks using a pointer to an incomplete partition are useless and must be turned off: even if they detect an address that is not part of the metapool, they cannot tell if it is an illegal address or simply an unregistered, but legal, object or function target. Second, checks on array indexing operations look up the operand and result of the `getelementptr` instruction. If either pointer points to an object in the splay tree, then the check can be performed, failing if both pointers are not within the same object. If the target object is not in the

Name	Description
pchk.reg.obj (MetaPool * MP, void * address, unsigned length, void * pool, int alloctrID)	Register an object starting at address of length bytes with the MetaPool MP. The pool and alloctrID arguments track which allocator and kernel pool (if applicable) were used to allocate the object.
pchk.drop.obj (MetaPool * MP, void * address, void * pool, int alloctrID)	Remove the object starting at address from the MetaPool MP. If MP is complete, perform an invalid free check.

Table 3: Instructions for Registering Memory Allocations

splay, nothing can be said. Overall, this means that "incomplete" partitions only have bounds checks on registered objects. We refer to this situation as "*reduced checks.*" Reduced checks are the sole source of false negatives in SVM, i.e., cases where a memory safety violation is not detected.

4.6 Multiple Entry Points

A kernel contains many entry points, including system calls, interrupts, and traps. System calls, in particular, bring pointers to objects allocated outside the kernel in via their parameters. SVA must ensure safety of kernel objects, while still allowing access to these objects even though they are not registered. We observe that pointer arguments in system calls may have three valid destinations. They may point to userspace objects, they may point to kernel objects when the kernel issues a system call internally, or they may point to objects returned by the kernel to userspace.

Objects in userspace are handled by registering all of userspace as a single object with every metapool reachable from system call arguments. Thus accesses to them are checked, and the checks pass since they find valid objects. This mechanism also ensures that userspace pointers stay in userspace; if an attacker tries to pass a buffer that starts in userspace but ends in kernel space in an attempt to read kernel memory, this will be detected as a bounds violation.

Internal system calls, those issued from within kernel space, are analyzed like any other function call; thus, the metapools reachable from the arguments already contain the objects being passed in.

Finally, we simply assume that the the last case – where user programs pass in pointers to kernel objects through system call arguments – does not occur. If a kernel wanted to allow this, SVA could support this via minor changes to the pointer analysis to allow checking on these objects. However, this is a very poor design for safety and stability reasons, so it is not supported.

4.7 Manufactured Addresses

Unlike most applications, a kernel typically uses a number of "manufactured addresses." These are most often used for accessing BIOS objects that exist on boot at certain addresses. These are, in effect, memory objects which are allocated prior to the start of the kernel. The kernel developer simply must register these objects prior to first use (using the SVA function pseudo_alloc), which the compiler then replaces with pchk.reg.obj, thus treating them like any other allocation. For example, in Linux's ACPI module, we insert pseudo_alloc(0xE0000, 0xFFFFF) before a statement that scans this range of memory (for a particular byte signature).

There are also C programming idioms that can *appear* to manufacture an address out of integer values even if they do not; most can be handled simply by tracking (pointer-sized) integer values as potential pointers during the pointer analysis, which is generally a necessary requirement for C compilers [18]. Some additional cases, however, may be too expensive to analyze completely, e.g., when bit or byte operations on small integers are used. For example, the Linux kernel performs bit operations on the stack pointer to obtain a pointer to the current task structure. Such operations have the problem that the resulting pointer can point to *any partition* in the points-to graph, complete or incomplete, since the address could be any absolute address.

One solution is simply to reject the code in the safety-checking compiler, requiring that such operations be rewritten in a more analyzable form. We take this approach in the current system, modifying the Linux source to eliminate such operations as explained in Section 6.3. Alternatively, we could ignore such operations to reduce initial porting effort, essentially trusting that they do not cause any violations of our safety guarantees.

4.8 Analysis Improvements

We have added several features to the pointer analysis to improve precision when analyzing kernel code. The Linux kernel often uses small constant values (e.g., 1 and -1) as return values from functions returning a pointer to indicate errors (bugs caused by this have been noted before [14]). These values appear as integer-to-pointer casts and would cause partitions to be marked unknown. We extended the analysis to treat such values (in a pointer context) simply as null. We also added limited tracking of integer dataflow to find all sources of these small constants for a cast site.

Since the kernel may issue system calls internally using the same dispatch mechanism as userspace, namely a trap with the system call (and hence kernel function) specified by numeric identifier, we had to be able to map from syscall number to kernel function in the analysis. This information can be obtained by inspecting all calls to the SVA-OS operation, sva_register_syscall, which is used by the kernel to register all system call handlers. With this information, we were able to analyze internal system calls simply as direct calls to the appropriate function.

We use a new heuristic to minimize the extent to which userspace pointers alias kernel memory. The original pointer analysis recognized and handled memcpy and memmove operations (and their in-kernel equivalents) as copy operations, but simply handled them by merging the source and target object nodes (i.e., handling the copy like p = q instead of *p = *q). However, for copy operations to or from userspace pointers (which are easily identifiable), we want to minimize the merging of kernel and userspace objects. Our new heuristic merges only the target nodes of outgoing edges of the objects being copied, but it requires precise type information for the source and target objects. If that type information is not available, the analysis *collapses* each node individually (sacrificing type safety) while preventing merging of the nodes themselves.

We also made two improvements that are not specific to kernel code. First, we can reduce spurious merging of objects by judiciously cloning functions. For example, different objects passed into the same function parameter from different call sites appear aliased and are therefore merged into a single partition by the pointer analysis. Cloning the function so that different copies are called for the different call sites eliminates this merging. Of course, cloning must be done carefully to avoid a large code blowup. We used several heuristics to choose when to create a clone of an existing function. The heuristics have been chosen intuitively and more experience and experiments are needed to tune them. Nevertheless, we saw significant improvements in the points-to graph and some improvements in the type information due to reduced merging, and the total size of the SVA bytecode file increased by less than 10%.

Second, the largest source of imprecision in the analysis results comes from certain hard-to-analyze indirect calls, especially through function pointers loaded from global tables attached to objects. Because the tables may mix functions with many different signatures, type safety is completely lost at an indirect call site with such a function pointer, even though the code itself never invokes incompatible functions at the call site. We introduce an annotation mechanism that kernel programmers can use at a call site to assert that the function signatures of all possible callees match the call. In some cases, this can reduce the number of valid targets at an annotated call site by two orders of magnitude. For example, for 7 call sites in Linux, the number of callees went down from 1189 each (they all get their call targets from the same points-to graph node) to a range of 3-61 callees per call site. This improves analysis precision, since fewer behaviors of callees are considered; safety, since fewer control flow paths exist and the programmer is making an assertion about which are valid; and speed of indirect call target checks, since the check is against a much smaller set of possible functions. In fact, with a small enough target set, it is profitable to "devirtualize" the call, i.e., to replace the indirect function call with an explicit `switch` or branch, which also allows the called functions be inlined if the inliner chooses. The current system only performs devirtualization at the indirect call sites where the function signature assertion was added.

4.9 Summary of Safety Guarantees

The SVA guarantees provided to a kernel vary for different (compiler-computed) partitions of data, or equivalently, different metapools. The strongest guarantees are for partitions that are proven *type-homogeneous* and *complete.* For partitions that lack one or both of these properties, the guarantees are correspondingly weakened.

Type-homogeneous, complete partitions:

For such partitions, the guarantees SVA provides are exactly [T1-T6] listed in Section 4.1. Note that the type safety guarantee (T2) applies to all objects in these partitions.

Non-type-homogeneous, complete partitions:

For such partitions, all the above guarantees hold *except*:

(N1) *No type safety*: Memory references may access or index objects in ways inconsistent with their type.

Note that *array indexing* and loads and stores are still checked and enforced; pointer arithmetic or bad casts cannot be used to compute and then use a pointer outside the bounds of an object. This is valuable because it prevents buffer overruns due to common programming errors like incorrect loop bounds, incorrect pointer arithmetic, illegal format strings, and too-small allocations due to integer overflow/underflow. Even if an illegal pointer-type cast or a dangling pointer use converts an arbitrary value to a pointer type, it can only be used to access a legal object within the correct partition for that pointer variable (this is stronger than CCured and SafeDrive, which only guarantee that a pointer dereference on a wild pointer will access *some* pointer value, but it can be to an arbitrary object).

Incomplete partitions:

Incomplete partitions must be treated as non-type homogeneous because the analysis for them was fundamentally incomplete: operations on such a partition in unanalyzed code may use a different, incompatible type. In addition, some run-time checks had to be relaxed. The guarantees, therefore, are the same as non-TH partitions above, except:

(I1) *No array bounds safety for external objects*: Array bounds are not enforced on "external" objects, even though the objects logically belong to a partition.

(I2) *Loads and stores to incomplete partitions may access arbitrary memory.* As explained in Section 4.5, no *load-store checks* are possible on such partitions.

(I3) *Deallocators may be called with illegal addresses.* No *double/illegal free checks* are done on the partition.

A note on component isolation:

We say a particular kernel component or extension is *isolated* from the rest of the kernel if it cannot perform any illegal writes to (or reads from) objects allocated in the rest of the kernel, i.e., except using legal accesses via function arguments or global variables.

With SVA, many partitions do get shared between different kernel components because (for example) objects allocated in one component are explicitly passed to others. SVA guarantees component isolation if (a) the component only accesses complete TH partitions (completeness can be achieved by compiling the complete kernel); and (b) dangling pointers are ignored (since SVA only guarantees "metapool-level" isolation and not fine-grained object-level isolation on such errors).

Even if these conditions are not met, SVA *improves but does not guarantee* isolation of a component from the rest of the kernel: many important memory errors that are common causes of memory corruption (e.g., buffer overruns, uninitialized pointers) cannot occur for kernel objects themselves.

5. MINIMIZING THE TRUSTED COMPUTING BASE

The approach described thus far uses pointer analysis results to compute the static partitioning of the memory objects in the kernel. This pointer analysis is interprocedural and relatively complex. (In our SAFECode compiler for standalone programs, Automatic Pool Allocation is also a complex interprocedural transformation.) Any bugs in the implementation of such complex passes may result in undetected security vulnerabilities. For this reason, it is important that such a complex piece of software be kept out of the

Trusted Computing Base (TCB) of SVA. Furthermore, we would also like to have a formal assurance that the strategy described thus far is sound, i.e., it provides the claimed guarantees such as memory safety, type safety for some subset of objects, control flow integrity, and analysis integrity.

The SAFECode safety approach has been formalized as a type system, and the type system together with an operational semantics encoding the run-time checks have been proved sound for a relevant subset of C [11]. We adapted that type system in the context of SVA, using metapools instead of the pool descriptors in SAFECode. The type system allows us to use a simple intraprocedural type checker to check that the complex pointer analysis is correct. Furthermore, it also provides a soundness guarantee on the principles used in our work. An ancillary benefit is that the type system *effectively encodes a sophisticated pointer analysis result directly within the bytecode*, potentially simplifying translation to native code since it does not have to repeat such an analysis.

More specifically, the SVA type system [11] essentially encodes each pointer with its associated metapool. For example, for a pointer `int *Q` pointing to a metapool M1, the type system defines the pointer's type to be `int *M1 Q`. If another pointer P has type `int *M2 *M3 P`, then it indicates that `P` points to objects in metapool `M3` which in turn contains pointers that point to objects in metapool `M2`. Effectively, this metapool information has been added as an extension of the underlying SVA type system. This encoded information in the form of types is analogous to the encoding of a "proof" in Proof Carrying Code [31]. The proof producer uses the results of pointer analysis and the metapool inference algorithm (described in Section 4.3) to infer the type qualifiers `M1`, `M2`, and `M3`. It then encodes this information as annotations on each pointer type.

The typing rules in this type system check that annotated types are never violated. For example, if pointer `int *M1 Q` is assigned a value `*P` where P is of type `int *M2 *M3`, then the typing rules flag this as an error. This will check that type annotations inferred by the proof producer are actually correct. This is essentially the same as checking that objects in `M3` point to objects in `M2` and not objects in `M1`.

The type checker implements the typing rules to check that the "encoded" proof is correct. The type checker is analogous to the proof checker in [31]. Because the typing rules only require local information (in fact, just the operands of each instruction), they are very simple and very fast, both attractive features for use within the virtual machine. Thus only the type checker (and not the complex compiler) is a part of the TCB.

Once the type checker checks that the metapool annotations are correct, it inserts all the run-time checks described in Section 4.5.

The type checker implementation should be carefully tested because it is part of the TCB. Evaluating its correctness experimentally can also increase our confidence that there are no errors in the manual proof of soundness [11]. For this evaluation, we injected 20 different bugs (5 instances each of 4 different kinds) in the pointer analysis results. The four kinds of bugs were incorrect variable aliasing, incorrect inter-node edges, incorrect claims of type homogeneity, and insufficient merging of points-to graph nodes. The verifier was able to detect all 20 bugs.

6. PORTING LINUX TO SVA

Porting an operating system to SVA requires three steps, as noted in Section 2: porting to SVA-OS, changes to memory allocators, and optional code changes to improve analysis quality. The specific changes we made to port the Linux 2.4.22 kernel [4] to SVA are described below.

One of our goals was to minimize the changes needed to the architecture-independent portions of a kernel. Table 4 summarizes the number of changes that were needed. Column "Total LOC" shows the total number of lines of source code in the original kernel, for each major component. The next three columns show the changes (total number of non-comment lines modified, deleted or added) for the three kinds of porting changes listed in Section 2. The last column shows the total number of lines of code changed. As the table shows, the number of changes required are quite small for the improved safety benefits they provide. Section 7 suggests some additional kernel changes that can improve the analysis effectiveness, but we expect those additional changes to be on the same order of magnitude as the ones shown here.

6.1 Porting to SVA-OS

Linux, like most OSes, use abstractions for interacting with hardware. Porting the Linux kernel to use SVA was a relatively simple matter of rewriting the architecture-dependent functions and macros to use SVA-OS instructions instead of assembly code, resulting in a kernel with *no inline assembly code*. The total number of architecture-dependent changes from `arch/i386` is shown as the last line in table 4. In some cases, the interface between the architecture independent and dependent code changed. For example, system call handlers rely upon the architecture-dependent code to directly modify saved program state to restart an interrupted system call. On SVA, either the system call handler code or the C library need to restart system calls; the SVA instruction set does not provide a mechanism for the architecture dependent code to use to modify saved program state directly.

The Linux kernel needed only a small number of changes to the architecture-independent code and to device drivers for porting to SVA-OS. The drivers needed changes in order to use the instructions for I/O. The core kernel needed some changes in the signal-handler dispatch code to save state on the kernel stack instead of the user stack, because the SVA-OS instructions provide no mechanism to inspect the state and ensure that the state has not been tampered with. In fact, many of the changes in Table 4 were due to changing the name of a structure. A cleaner port may yield a kernel with even fewer changes to its architecture independent code.

6.2 Memory Allocator Changes

As detailed in Section 4.4, a kernel's memory allocators require several modifications in order to take advantage of the memory safety properties of our virtual machine.

First, we identified Linux's `kmem_cache_alloc` allocator as the only pool allocator; the rest were treated as ordinary allocators. We identified the `kmem_cache_alloc` and `kmem_cache_free` functions as allocation and deallocation routines so that the safety checking compiler inserts object registrations after those operations. We similarly identified the allocation/deallocation routines for the other allocators.

[4] Linux 2.4.22 was a standard kernel in use when we began this project. We plan to move to Linux 2.6 in future work.

Section	Total LOC	SVA-OS	Allocators	Analysis	Total Modified
Arch-indep core	9,822	41	76	3	120
Net Drivers	399,872	12	0	6	18
Net Protocols	169,832	23	0	29	52
Core Filesys.	18,499	78	0	19	97
Ext3 Filesys.	5,207	0	0	1	1
Total indep	603,232	154	76	58	288
Arch-dep. core	29,237	4,777	0	1	4,778

Table 4: Number of lines modified in the Linux kernel

Second, we added a routine to support dynamic allocation throughout the kernel's lifetime (for stack objects promoted to the heap). This uses `_alloc_bootmem` early in the boot stages and then uses `kmalloc`.

Third, all kernel allocators must refrain from returning their physical page frames back to the system until the SVM indicates that it is safe to do so (the SVM will do this when the metapool is destroyed). For Linux, we modified `kmem_cache_create` to mark all pools with the `SLAB_NO_REAP` flag to prevent the buddy allocator from reclaiming unused memory from the pools when it is low on memory. We are still working on providing similar functionality for memory allocated by `vmalloc`.

Fourth, all kernel allocators must properly space objects to prevent type conversions when accessing dangling pointers. The Linux memory allocators already do this, so no changes were necessary.

Fifth, kernel allocators must zero memory before returning it. We modified `kmem_cache_alloc` to zero memory and call the pool's constructor function, if present. We are investigating what needs to be done for other kernel allocators.

Finally, as explained in Section 4.3, we exposed the relationship between `kmalloc` and `kmem_cache_alloc`. The former is simply implemented as a collection of caches for different sizes. By exposing this, the compiler only needs to merge metapools that correspond to each cache instead of all those corresponding to any `kmalloc`.

The number of changes required for the Linux allocators are shown in the fourth column of Table 4. The required changes are localized to a single file in the core kernel and are trivial to add.

6.3 Changes to Improve Analysis

We made several changes to the kernel source code to help improve the precision of the analysis, including the changes to eliminate unanalyzable int-to-pointer casts as explained in Section 4.7. First, we rewrote function signatures to reflect more accurately the type of the arguments. Second, we rewrote some structures declared as unions to use explicit structures. Last, we rewrote hard-to-analyze stack usage for accessing task structures.

Several functions, most notably the *sys_ioctl* related ones, have parameters that are interpreted as both ints and pointers, depending on the values of other arguments. In the case of *sys_ioctl*, in fact, the argument is almost always used as a pointer into userspace.We changed the parameter (and a few other similar locations) to all related functions in the kernel to be a pointer; those few locations that treat it as an integer cast from pointer to integer. This is sufficient because casts from pointer to integer do not affect the pointer analysis whereas casts from integers to pointers look like unanalyzable manufactured pointers.

Some important structures in the kernel were declared in a way that obscured the type of the structure. The most notable example is the initial task structure. This structure is declared as a union of a task structure and an array which acts as the initial kernel stack. Changing this to a struct with the task structure and a smaller array, which reserves the same amount of memory, makes the analysis better able understand the type, increasing precision in the analysis.

The last major change was in changing how the current task structure is accessed. The kernel performed masking operations of the stack pointer to find the structure. This is very hard to analyze and was changed into an easier to analyze global variable.

7. EXPERIMENTAL RESULTS

Our experiments aim to evaluate three aspects of SVA: the performance overhead of SVA due to the SVA-OS instructions and the run-time safety checks; the effectiveness of our approach in catching exploits in different kernel subsystems; and an understanding of what fraction of the kernel obtains the strongest (type-safe, complete) and weakest (incomplete) safety guarantees.

7.1 Performance Overheads

We evaluated the performance overheads of SVA using the HBench-OS microbenchmark suite [7] and a set of standard user-level applications. We report performance metrics for four versions of the Linux 2.4.22 kernel:

1. *Linux-native*: the original kernel compiled directly to native code with GCC 3.3 [5].

2. *Linux-SVA-GCC*: the SVA-ported kernel (see Section 6) compiled with GCC 3.3.

3. *Linux-SVA-LLVM*: the SVA-ported kernel compiled with the LLVM C compiler.

4. *Linux-SVA-Safe*: the SVA-ported kernel compiled with the LLVM C compiler plus the safety-checking passes.

Because SVA-OS is implemented simply as a C library that can be linked with the SVA-ported Linux kernel, that kernel can be compiled with any C compiler. Thus, Linux-SVA-GCC and Linux-SVA-LLVM are simply that kernel compiled with the GCC and LLVM C compilers, respectively. The Linux-SVA-Safe version is nearly the same as the Linux-SVA-LLVM one with the additional run-time checks inserted. The only other difference is that Linux-SVA-Safe also has the two compiler *transformations* described in Section 4.8 to make alias analysis results more accurate: function cloning and function devirtualization. We expect the performance impact of these to be relatively small (although the precision improvement in pointer analysis can be significant, these pointer analysis results are only used for safety checking and are *not* used for any optimizations). To generate native code for the Linux-SVA-LLVM and Linux-SVA-Safe kernels, we translate LLVM bytecode to C code and compile the output with GCC4 -O2 (this factors out the difference between the GCC and LLVM native back-ends).

[5] We found that GCC4 miscompiled the native kernel

All four kernels are configured identically, in SMP mode with TCP/IPv4, Ext2, Ext3 and ISO 9660 filesystems, and a few drivers. For Linux-SVA-Safe, some portions of the kernel were not processed by the safety checking passes because of errors encountered in booting the kernel. These were the memory subsystem (`mm/mm.o` and `arch/llvm/mm/mm.o`), two sets of utility libraries (`lib/lib.a` and `arch/llvm/lib/lib.a`), and the character drivers.

We ran all of our experiments on an 800 MHz Pentium III machine with a 256KB L2 cache and 256MB of RAM. While this is an old machine, we expect little change in relative performance overhead on newer x86 processors.

7.1.1 Application Performance Impact

The results for application tests are shown in Table 5. We used four "local" applications (top half of Table 5 showing the mean of 10 runs): two SPEC benchmarks, the MP3 encoder `lame`, and `ldd` (a kernel-intensive utility that prints the libraries needed by an executable). Column 2 shows that these programs spend widely varying percentages of time executing in the kernel, with `ldd` being an extreme case for local programs. We also used two servers: OpenSSH's `sshd` and `thttpd` (bottom half of Table 5 showing the median of 3 runs). For the local applications, we measured the elapsed time using a remote machine's clock so as to not rely on the experimental kernel's time keeping facilities. For `sshd`, we used the `scp` command from a remote machine to measure the time needed to transfer a 42 megabyte file. For `thttpd`, we used ApacheBench to measure the total time to transfer various files over 25 simultaneous connections. All server tests were executed over an isolated 100 Mb Ethernet network.

Table 5 shows the execution time with the Linux-native kernel and the overheads when using each of the other three kernels as a percentage of the Linux-native kernel's execution time (i.e., $100 \times (T_{other} - T_{native})/T_{native}\%$). Using the numbers for each of the latter three kernels and subtracting the preceding column (using 0% for Linux-native) isolates the impact of the SVA-OS instructions alone, of using the LLVM C compiler instead of GCC, and of introducing the safety checks.

Three of the local applications show little overhead, including `bzip2`, which spends 16% of its time in the kernel. Only `ldd` shows significant overhead, most likely due to its heavy use of `open/close` (see below). The network benchmarks, generally, show greater overhead. The worst case, with a 62% slowdown, is `thttpd` serving a small file to many simultaneous connections. When serving larger files, the overhead drops considerably, to 4.6%. Note that the slowdown due to the SVA-OS instructions alone (Linux-SVA-GCC vs. Linux-native) is very small for `thttpd`; most of the penalty comes from the safety checking overheads. The `sshd` server shows no overhead at all (the apparent speedup from safety checks is within the range of measurement error).

Table 6 shows that the total impact on bandwidth for `thttpd` is reasonable, with reductions below 34%. Most of the overhead stems from our safety checks.

7.1.2 Kernel Microbenchmarks

To investigate the impact of SVA on kernel performance in more detail, we used several latency and bandwidth benchmarks from the HBench-OS suite [7]. We configured HBench-OS to run each test 50 times and measured time using the processor's cycle counters. Tables 7 and 8 show the mean of the 50 runs.

Test	% System Time	Native (s)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
bzip2 (8.6MB)	16.4	11.1	1.8	0.9	1.8
lame (42MB)	0.91	12.7	0.8	0.0	1.6
gcc (-O3 58klog)	4.07	24.3	0.4	1.2	2.1
ldd (all system libs)	55.9	1.8	44.4	11.1	66.7
scp (42MB)	-	9.2	0.00	0.00	-1.09
thttpd (311B)	-	1.69	10.1	13.6	61.5
thttpd (85K)	-	36.1	0.00	-0.03	4.57
thttpd (cgi)	-	19.4	8.16	9.40	37.2

Table 5: Application latency increase as a percentage of Linux native performance.

`thttpd` request	# Requests	Native (KB/s)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
311 B	5k	1482	3.10	4.59	33.3
85 KB	5k	11414	0.21	-0.26	2.33
cgi	1k	28.3	-0.32	-0.46	21.8

Table 6: `thttpd` Bandwidth reduction as a percentage of Linux native performance (25 concurrent connections)

Overall file bandwidth has small overhead (8%) while pipe bandwidth overhead is higher (67%). The latency results in Table 7 show many moderate overheads of 20-56%, but there are a few severe cases of overheads reaching 2x-4x.

The difference between the LLVM and GCC code generators creates at most a 13% overhead. Most of the overhead comes from the use of the SVA-OS instructions and the run-time safety checks. For system calls that do little processing, the SVA-OS instructions cause the most overhead. Perhaps not surprisingly, run-time checks tend to add the most overhead to system calls that perform substantially more computation, e.g., open/close, pipe, fork and fork/exec.

7.1.3 Future Performance Improvements

The performance experiments above only provide a snapshot showing the current performance of the SVA prototype. The overall system design effort has been quite large, and therefore, we have only had time to do preliminary performance tuning of the safety checking compiler and run-time system in SVA. There are still at least three major improvements that we expect to make to reduce the overheads of the run-time checks:

1. using "fat pointers" instead of splay tree lookups for pointer variables in *complete* partitions, which are completely internal to the kernel being compiled (avoiding the compatibility problems fat pointers can cause when being passed to or from external code);

2. better compile-time optimizations on bounds-checks, especially hoisting checks out of loops with monotonic index ranges (a common case); and

3. performing static array bounds checking to reduce run-time checks on `getelementptr` operations.

Test	Native (μs)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
getpid	0.38	21.1	21.1	28.9
getrusage	0.63	39.7	27.0	42.9
gettimeofday	0.61	47.5	52.5	55.7
open/close	2.97	14.8	27.3	386
sbrk	0.53	20.8	26.4	26.4
sigaction	0.86	14.0	14.0	123
write	0.71	39.4	38.0	54.9
pipe	7.25	62.8	62.2	280
fork	106	24.9	23.3	74.5
fork/exec	676	17.7	20.6	54.2

Table 7: Latency increase for raw kernel operations as a percentage of Linux native performance

Test	Native (MB/s)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
file read (32k)	407	0.80	1.07	1.01
file read (64k)	410	0.69	0.99	0.80
file read (128k)	350	5.15	6.10	8.36
pipe (32k)	567	29.4	31.2	66.4
pipe (64k)	574	29.1	31.0	66.5
pipe (128k)	315	12.5	17.4	51.4

Table 8: Bandwidth reduction for raw kernel operations as a percentage of Linux native performance

7.2 Exploit Detection

To see how well our system detects exploits that use memory error vulnerabilities, we tried five different exploits on our system that were previously reported for this version of the Linux kernel, and which occur in different subsystems of the kernel. We were limited to five because we had to choose ones that were memory error exploits and for which working proof of concept code existed. Only one exploit was in a device driver; the rest were in the IPv4 network module (two exploits), the Bluetooth protocol module, and the ELF loader in the core filesystem module.

The SVA checks caught four out of the five exploits. Two of these were integer overflow errors in which too small a heap object was allocated, causing an array overrun [21, 40]. A third was a simple out of bounds error when indexing into a global array [46]. Finally, SVA caught a buffer overrun caused by decrementing a length byte to a negative value and then using that value as an unsigned array length (making it a large positive number) [39].

SVA did not catch the exploit in the ELF loader [38]. This one caused the kernel's user-to-kernel copying routine to overflow a kernel object by using an unchecked negative value (interpreted as a large positive value) as the object length. SVA failed to catch this because the implementation of the user-to-kernel copying function was in a kernel library that was not included when running the safety-checking compiler on the kernel. We anticipate that including that library will allow SVA to catch this exploit.

7.3 Analysis Results

To get a sense of how many accesses receive the different levels of security, we examined the static number of instructions that access type safe metapools (the highest level) and the number that access incomplete metapools (the lowest level). These represent the two extremes of our safety guarantees. However, not all accesses are equally easy to exploit. Therefore we considered four cases: loads, stores, structure indexing (*struct.field*), and array indexing (*array*[*index*]). Buffer overflows fall into this last category. The first kernel presented in Table 9 is the kernel used in the performance and safety experiments; the second one includes the complete kernel. No sources of incompleteness remain in the second kernel because all entry points are known to the analysis, userspace is considered a valid object for syscall parameters, and all SVA operations are understood. Also, in both cases, no unanalyzable casts to pointers remained.

	Allocation Sites Seen	Access Type	Incom-plete	Type Safe
Kernel As Tested	99.3%	Loads	80%	29%
		Stores	75%	32%
		Structure Indexing	91%	16%
		Array Indexing	71%	41%
Entire Kernel	100%	Loads	0%	26%
		Stores	0%	34%
		Structure Indexing	0%	12%
		Array Indexing	0%	39%

Table 9: Static metrics of the effectiveness of the safety-checking compiler

Column 4 in Table 9 shows that in the first kernel, 71%–91% of different kinds of accesses are to incomplete nodes, i.e., may be accessing unregistered objects. The figure is 71% for array indexing operations, the most common cause of memory errors. The second column in the table also shows, however, that over 99% of dynamic allocation sites in the kernel were instrumented; most of the rest are objects used internally by the allocators (which are within the memory subsystem). This means that almost all objects will be checked. (The fraction of accesses to incomplete *nodes* may nevertheless be high because many points-to graph nodes are likely to be exposed to the memory system and low-level libraries.) In the complete kernel, there are no unregistered objects, so all objects will be checked.

Table 9 also shows that much of the kernel is not type safe. This reduces the ability to eliminate load-store checks and increases the cost of the splay tree lookups because the pointer analysis may not be able to generate fine-grained metapools. The level of type safety did not vary much between the complete and incomplete kernels. We believe two types of changes can greatly improve these numbers: additional porting effort (to reduce non-type-safe C idioms) and several key refinements to the type merging properties in our pointer analysis. Again, these are both tuning exercises that have not yet been done for SVA or the ported kernel.

8. RELATED WORK

Since the early work on system virtualization on the IBM S/370, there have been numerous systems (called hypervisors or Virtual Machine Monitors) that allow multiple, complete OS instances on a single hardware system. These include recent systems like VMware, Connectix, Denali, Xen, VirtualPC, Qemu, and others. The SVA approach is orthogonal to the hypervisor based virtualization approach. The two approaches could be combined to achieve new properties such as memory safety for the hypervisor itself. Also, some of the privileged operations in SVA-OS could instead use standardized mechanisms that are being defined for hypervisor support, such as VMI for Linux [2].

As discussed in Section 1, there have been several experimental operating systems written using safe languages [5, 19, 36, 23, 9, 4, 24, 22]. It is difficult to rewrite today's commodity systems in such languages. Furthermore, all these systems rely on garbage collection to manage memory. Retrofitting garbage collection into commodity systems is quite difficult since many reachable heap-allocated objects would get leaked. In contrast, SVA provides a version of a safe execution environment to commodity kernels directly, requiring no language changes and preserving the low-level, explicit memory management techniques of the kernel.

Numerous projects have focused on isolated execution of application-specific extensions or specific kernel components (especially device drivers) within a commodity kernel [30, 6, 47, 33, 43, 45, 50]. As explained in Section 4.9, SVA achieves isolation between a kernel component (or extension) and the rest of the kernel under certain conditions, and improves but does not guarantee isolation otherwise. Besides isolation, SVA also aims to enforce memory safety for the *entire operating system* and security-sensitive software running on it. Such a comprehensive approach is valuable because, as shown in our experimental results, there can be exploitable vulnerabilities in core kernel code as well as in device drivers.

While some of the above systems for isolation use interpreters [30] or coarse-grained compartmentalization of memory [47, 43, 45], a few others enforce fine-grained safety via language or compiler techniques applicable to commodity kernels [33, 6, 50].

Proof-carrying code [33] provides an efficient and safe way to write kernel extensions and can be used with a variety of proof formalisms for encoding safety or security properties of programs [31]. The approach, however, relies on type-safety (e.g., the PCC compiler for C was limited to a type-safe subset of C with garbage collection [34]) and appears difficult to extend to the full generality of (non-type-safe) C used in commodity kernels.

The Open Kernel Environment (OKE) allows custom extensions written in the Cyclone language[25] to be run in the kernel [6]. This approach is difficult to extend to a complete commodity kernel because Cyclone has many syntactic differences from C that are required for enforcing safety, and (as the OKE authors demonstrate), Cyclone introduces some significant implementation challenges within a kernel, including the need for custom garbage collection [6].

The SafeDrive project provides fine-grained memory and type safety within system extensions (like Cyclone), although their only reported experience is with device drivers [50]. Deputy requires annotations (at external entry points, and potentially other interfaces) to identify the bounds of incoming pointer variables. In contrast to both Cyclone and Deputy, SVA provides fine-grained safety guarantees both for the core kernel as well as device drivers and avoids the need for annotations.

Some static analysis techniques, such as Engler et. al.'s work [14], have targeted bugs in OS kernels. These techniques are able to find a variety of programing mistakes, from missing null pointer checks to lock misuse. However, these techniques are not able to provide guarantees of memory safety, as SVA aims to do. These techniques are complementary to runtime memory safety enforcement because they can be used to *eliminate* some of the errors that SVA would only discover at run-time, and they can address broader classes of errors beyond memory and type safety.

TALx86 is a typed assembly language (TAL) for the x86 instruction set that can express type information from rich high-level languages and can encode safety properties on native x86 code. On the other hand, TALx86 assumes garbage collection for encoding any type system with safety guarantees. Furthermore, any type information in TALx86 has to correctly handle many low-level features such as callee-saved registers, many details of the call stack, computed branch addresses (which require typing preconditions on branch target labels), and general-purpose registers (GPRs) that can hold multiple types of values. *None of these features arise with SVA* which greatly simplifies the tasks of defining and implementing the encoding of security properties. Overall, we believe SVA provides a more attractive foundation for encoding multiple security properties in (virtual) object code and verifying them at the end-user's system. It could be combined with a lower-level layer like TALx86 to verify these properties on the generated native code as well, taking the translator out of the trusted computing base.

9. SUMMARY AND FUTURE WORK

Secure Virtual Architecture (SVA) defines a virtual instruction set, implemented using a compiler-based virtual machine, suitable for a commodity kernel and ordinary applications. SVA uses a novel strategy to enforce a safe execution environment for both kernel and application code. The approach provides many of the benefits of a safe language like Modula-3, Java, or C#, without sacrificing the low-level control over memory layout and memory allocation/deallocation enabled by C code in commodity kernels. Our experiments with 5 previously reported memory safety exploits for the Linux 2.4.22 kernel (for which exploit code is available) show that SVA is able to prevent 4 out of the 5 exploits and would prevent the fifth one simply by compiling an additional kernel library.

In our ongoing work, we are implementing the three major optimizations described in Section 7.1.3 (and several minor ones) which should greatly improve the performance overheads for kernel operations. We are also investigating techniques for recovering from kernel memory access errors, not just in kernel extensions but also in the core kernel.

In future work, we plan to investigate higher-level security problems that could be addressed in novel ways by using the compiler-based virtual machine in cooperation with the OS. Some examples include enforcing information flow between programs; enforcing privilege separation and minimization at load time for programs that are today run via "setuid"; analyzing memory consumption to protect against denial-of-service attacks via memory-exhaustion; and encoding various security policies as type systems within the typed bytecode language in SVA.

Acknowledgements

We would like to thank our shepherd, Adrian Perrig, and the anonymous reviewers for their extensive and helpful feedback on the paper. We would also like to thank Sam King for his feedback on the paper and the work, and Pierre Salverda, David Raila, and Roy Campbell for numerous insightful discussions about the design of SVA-OS.

10. REFERENCES

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-Level Virtual Instruction Set Architecture. In *Int'l Symp. on Microarchitecture*, Dec. 2003.

[2] Z. Amsden. Transparent paravirtualization for linux. In *Linux Symposium*, Ottawa, Canada, Jul 2006.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 1994.

[4] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. on Prog. Lang. and Sys.*, 27(4):583–630, 2005.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM Symp. on Op. Sys. Principles*, pages 267–284, Copper Mountain, CO, 1995.

[6] H. Bos and B. Samwel. Safe kernel programming in the oke. In *Proceedings of IEEE OPENARCH*, 2002.

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

[8] J. Criswell, B. Monroe, and V. Adve. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, Boston, June 2006.

[9] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, 1998.

[10] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the Int'l Conf. on Software Engineering*, May 2006.

[11] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 2006.

[12] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Trans. on Embedded Computing Systems*, Feb. 2005.

[13] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. ACM Symp. on Op. Sys. Principles*, October 2003.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. ACM Symp. on Op. Sys. Principles*, 2001.

[15] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 2003. Technical Report 2003-1916.

[16] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys*, 2006.

[17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: a substrate for kernel and language research. In *Proc. ACM Symp. on Op. Sys. Principles*, 1997.

[18] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2001.

[19] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder. The JX Operating System. In *Proc. USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 2002.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 2002.

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

[22] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. In *Proc. ACM SIGPLAN Int'l Conf. on Functional Programming*, 2005.

[23] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, June 1998.

[24] G. C. Hunt and J. R. Larus. Singularity Design Motivation (Singularity Technical Report 1). Technical Report MSR-TR-2004-105, Microsoft Research, Dec 2004.

[25] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, 2002.

[26] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[27] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization*, Mar 2004.

[28] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, Chicago, IL, Jun 2005.

[29] C. Lattner, A. D. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, San Diego, USA, Jun 2007.

[30] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference*, pages 259–270, 1993.

[31] G. C. Necula. Proof-carrying code. In *Proc. ACM SIGACT Symp. on Principles of Prog. Lang.*, Jan. 1997.

[32] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. on Prog. Lang. and Sys.*, 2005.

[33] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Symp. on Op. Sys. Design and Impl.*, 1996.

[34] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 333–344, 1998.

[35] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Comm.*, 2003.

[36] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System.* Addison-Wesley, Reading, MA, USA, 1999.

[37] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symp. on Op. Sys. Design and Impl.*, pages 213–227, Seattle, WA, 1996.

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

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

[40] P. Starzetz and W. Purczynski. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. http://www.securityfocus.com/bid/10179.

[41] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. ACM SIGACT Symp. on Principles of Prog. Lang.*, 1996.

[42] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Symp. on Op. Sys. Design and Impl.*, Dec. 2004.

[43] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM Symp. on Op. Sys. Principles*, 2003.

[44] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst*, 23(1):77–110, 2005.

[45] Úlfar Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Op. Sys. Design and Impl.*, pages 75–88. USENIX, Nov. 2006.

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

[47] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[48] D. Walker. A type system for expressive security policies. In *Proc. ACM SIGACT Symp. on Principles of Prog. Lang.*, pages 254–267, 2000.

[49] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *Proc. of the ACM workshop on Rapid malcode*, 2003.

[50] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Symp. on Op. Sys. Design and Impl.*, pages 45–60. USENIX, Nov. 2006.