Enforcing Least Privilege Memory Views for Multithreaded Applications

Terry Ching-Hsiang Hsu
Purdue University
terryhsu@purdue.edu

Kevin Hoffman
eFolder Inc.
khoffman@efolder.net

Patrick Eugster
Purdue University and TU Darmstadt
peugster@cs.purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

Abstract

Failing to properly isolate components in the same address space has resulted in a substantial amount of vulnerabilities. Enforcing the least privilege principle for memory accesses can selectively isolate software components to restrict attack surface and prevent unintended cross-component memory corruption. However, the boundaries and interactions between software components are hard to reason about and existing approaches have failed to stop attackers from exploiting vulnerabilities caused by poor isolation.

We present the *secure memory views* (SMV) model: a practical and efficient model for secure and selective memory isolation in monolithic multithreaded applications. SMV is a third generation privilege separation technique that offers explicit access control of memory and allows concurrent threads within the same process to partially share or fully isolate their memory space in a controlled and parallel manner following application requirements. An evaluation of our prototype in the Linux kernel (TCB $<$ 1,800 LOC) shows negligible runtime performance overhead in real-world applications including Cherokee web server ($<$ 0.69%), Apache `httpd` web server ($<$ 0.93%), and Mozilla Firefox web browser ($<$ 1.89%) with at most 12 LOC changes.

1. INTRODUCTION

Ideally, software components are separated logically into small fault compartments, so that a defect in one component cannot compromise the others. This concept of *privilege separation* [35, 36] protects confidentiality and integrity of data (and code) that should only be accessible from small trusted components. However, most applications use a *single address space*, shared among all components and threads. Redesigning all legacy multithreaded applications to use processes for isolation is impractical. Today's software, such as web servers and browsers, enhances its functionality through libraries, modules, and plugins that are developed independently by various *third-parties*. Failing to properly separate privileges in applications and confine software components in terms of their memory spaces leaves a system vulnerable to attacks such as privilege escalation, denial-of-service, buffer overflows, and control-flow hijacking, jeopardizing both the stability and the security of the system.

Many proposals exist for privilege separation in a monolithic application. The first generation privilege separation techniques focus on splitting a process into different single-process compartments. Provos et al. [33] presented an intra-process privilege separation case study by manually partitioning OpenSSH components into privileged master processes and unprivileged slave processes. Privtrans [9] automated the partitioning procedure. Wedge [8] then introduced capabilities to privilege separation and Salus [40] enabled a dynamic security policy. Unfortunately, all these techniques cannot support multithreaded compartments.

Second generation privilege separation techniques like Arbiter [47] aimed to support multithreaded applications by allowing concurrent thread execution. However, Arbiter's implementation for separating memory space and its serialized user-level memory management impose prohibitive runtime overhead (200% – 400% for memory operations). As a result, the thread execution is not fully concurrent since all threads must wait on a global barrier to tag memory pages for capabilities. In addition, the required retrofitting efforts for legacy software are non-trivial, as the case studies showed that at least hundreds of lines of code (LOC) changes are required to separate software components even for applications that have small code base sizes (8K LOC).

We postulate that a third generation privilege separation technique for achieving intra-process isolation in monolithic multithreaded applications such as the Cherokee web server, Apache `httpd`, and the Mozilla Firefox web browser, needs to fulfill the following requirements (which are only partially addressed by existing solutions) for wide adoption:

- *Genericity and flexibility* (GF): Implementing privilege separation in different types of applications requires programmers to employ completely different abstractions and concepts. A general model with a universal interface and isolation concept that supports *both* client- and server-side multithreaded applications is needed (e.g., compartments in the Firefox browser, worker buffers in Cherokee web server, worker pools in Apache `httpd` web server).

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978327

	1st gen technique				2nd gen technique	3rd gen technique
Problem tackled	Non-parallel privilege separation				Concurrent execution and dynamic security policy	Concurrent memory operations and high performance
Issue vs solution	**OpenSSH** [33]	**Privtrans** [9]	**Wedge** [8]	**Salus** [40]	**Arbiter** [47]	**SMV** (This paper)
Security principal	Process	Process	Single thread	Single thread	Multiple threads	Multiple threads
Parallel execution	Not handled	Not handled	Not handled	Not handled	Partially handled	Yes
Parallel tagging	Not handled	Not handled	Not handled	Not handled	Not handled	Yes
Security policy	Static	Static	Static	Dynamic	Dynamic	Dynamic
TCB	OS	Compiler, OS	OS	Library, OS	Library, OS	OS (<1800 LOC)
Refactoring efforts	Fully manual	Annotations	Tool assisted	Tool assisted	Fully manual(>100 Δ LOC)	Library assisted(<20 Δ LOC)
Use cases	OpenSSH	OpenSSH etc.	OpenSSH etc.	PolarSSL	FUSE (8K LOC) etc.	Firefox (13M LOC) etc.

Table 1: Issues and solutions for intra-process privilege separation techniques.

- *Ease of use* (EU): Programmers prefer to realize their desired security policy in a model with a high-level API rather than through low-level error-prone memory management tools without intra-process capabilities (e.g., `mmap`, `shmem`.). In particular, porting legacy software to a new model has to be easy despite the complexity of component interweaving and the underlying assumption of shared memory (e.g., Firefox, which contains 13M LOC), and should be possible with minimal code refactoring efforts.
- *No hardware modifications* (NH): Over-privileged multithreaded applications are pervasive. A model that is ready to run on today's commodity hardware (even regardless of the CPU brands/models) is necessary for wide deployment.
- *Low runtime overhead* (LO): Monitoring application memory accesses at high frequency is unrealistic for practical systems. A practical model must be implemented in a way that incurs only negligible runtime overheads. In particular, enhanced security should not sacrifice the parallelism in multithreaded applications. A model has to support selective memory isolation for multiple computing entities (i.e., multiple threads can exercise the same privilege to parallelize a given workload and perform highly parallel memory operations).

To address the above challenges, we propose a third generation privilege separation solution for monolithic applications: *secure memory views* (SMVs) – a model and architecture that efficiently enforces differential security and fault isolation policies in monolithic multithreaded applications at negligible overheads. SMV protects applications from negligent or malicious memory accesses between software components. In short, the intrinsically shared process address space is divided into a dynamic set of *memory protection domains*. A thread container SMV maintains a collection of memory protection domains that define the *memory view* for its associated threads. Access privileges to the memory protection domains are explicitly defined in the SMVs and the associated `SMVthread`s must strictly follow the defined security policies. The SMV model provides a well-defined interface for programmers to exercise the least privilege principle for arbitrary software objects "inside" a multithreaded process. For example, a server's worker thread can be configured to allow access to its thread stack and part of the global server configuration but not to the private key that resides within the same process address space.

With the SMV model, the programmer can enforce different access permissions for different components in a single address space (GF). New software can leverage the full API and can be designed to only share data along a well-defined API, and existing software can be retrofitted (with minimal code changes) by instrumenting calls across component boundaries to change the underlying memory view (EU). Moreover, the privilege enforcement relies on OS kernel level page table manipulation and standard hardware virtual memory protection mechanisms (NH). Therefore, the SMV model does not suffer from the performance overheads (LO) imposed by IPC (vs in-memory communication), user level memory management (vs kernel level), or per-instruction reference monitors (vs hardware trap). The SMV model's programmability and efficient privilege enforcement mechanism allow it to protect *both* client- and server-side multithreaded applications with low overhead.

We implemented a prototype of the SMV model in the Linux kernel. Our evaluation demonstrates (a) its negligible runtime overhead in the presence of high concurrency using multithreaded benchmarks that employ the general producer-consumer pattern, and (b) the immediate benefit of efficient software component isolation by compartmentalizing client connections for the popular Cherokee and Apache `httpd` web servers and the compartments in the Firefox web browser. SMVs incur only around 2% runtime overhead overall with 2 LOC changes for the multithreaded benchmark PARSEC, 0.69% throughput overhead with 2 LOC changes for Cherokee, 0.93% throughput overhead with 2 LOC changes for Apache `httpd`, and 1.89% runtime overhead with only 12 LOC changes for the Firefox web browser. Note that SMV focuses on restricting memory views for individual threads, access permissions for kernel APIs is an orthogonal problem that is well covered by, e.g., AppArmor [4], SELinux [39], or the seccomp framework [38].

In summary, this paper makes the following contributions:

Design of the SMV model which provides threads with fine-grained control over privileges for a shared address space.

Specification of an SMV API for programmers that facilitates porting existing `pthread` applications.

Implementation of the SMV model that consists of a trusted Linux kernel component (implementing enforcement) and the corresponding untrusted user-space library that implement the SMV API, which is publicly available along with our benchmarks and test suite[1].

Evaluation of our prototype implementation showing that SMVs achieve all four desired requirements as a practical and efficient model for enforcing least privilege memory views for multithreaded applications in practice.

[1] https://github.com/terry-hsu/smv

2. THREAT MODEL AND OBJECTIVES

Threat model. We assume that the attacker, an unprivileged user without root permissions, can control a thread in a vulnerable multithreaded program, allocate memory, and fork more threads up to resource limits on a trusted kernel with sound hardware. The adversary will try to escalate privileges through the attacker-controlled threads or gain control of another thread, e.g., by reading or writing data of another module or executing code of another module. In this model, the adversary may read or write any data that the controlled thread has access to. The adversary may also attempt to bypass protection domains by exploiting race conditions between threads or by leveraging confused deputy attacks, e.g., through the API exported by other threads. We assume that the OS kernel is not compromised (OS kernel security is an orthogonal topic [24]) and user-space libraries installed by root users are trusted. We assume that the access permissions both of the memory views (enforced through SMV) and for the kernel (enforced through AppArmor, SELinux, or seccomp) are set correctly.

Objectives. The key objective of the SMV model is to efficiently protect memory references of threads to prevent unintentional or malicious accesses to privileged memory areas during the lifetime of a program. Threads may communicate with other threads through mutually shared memory areas set up by the programmer through SMVs. The SMV model restricts the memory boundaries and memory access permissions for each thread. Without SMVs, an untrusted thread (e.g., a compromised worker thread) may access *arbitrary* software objects (e.g., the private key) within its process (e.g., a web server). Existing programs assume a shared memory space for threads and SMVs must therefore validate that all threads follow the memory rules defined by the programmer (cf. Section 3.2). Threads that deviate from these memory reference rules are killed by the system.

The SMV model aims to strictly confine the memory access boundaries for multithreaded programs while *preserving all four desired requirements* for intra-process isolation. We argue in Section 3 that the SMV model along with the memory access enforcement can constrain threads within the programmer-defined memory boundaries.

3. SMV MODEL DESIGN

The SMV model consists of three abstractions: *memory protection domains*, *secure memory views*, and `SMVthread`s. The SMV model uses user-defined security policies to enforce the threads' privileges in accessing the shared memory space. The flexibility and programmability of the model allows a programmer to specify the protection domains using high-level abstractions while enforcing the security policy at the lowest level of the software stack (page tables) with acceptable runtime overhead.

3.1 Memory Protection Domains

We define a *memory protection domain* as a contiguous range of virtual memory. Any memory address can only belong to one memory protection domain. In this way, a large shared memory space such as the heap can be divided into several distinct sets of memory protection domains. For example, a process can create a private memory protection domain that is only accessible by *one* thread, or a *partially* shared memory protection domain such that only threads with explicit privileges can access it. In addition, an in-memory communication domain can be allocated with global access privileges so that all threads can exchange data without relying on expensive IPC. In general, an unprivileged thread cannot tamper with a memory protection domain even if there exists a defect in the code of the thread. We use the term *memory domain* to refer to the memory protection domain in the rest of the paper.

3.2 Secure Memory Views

We define a *secure memory view* (SMV) to be a thread container with a collection of memory domains. The memory blocks covered by a memory view can only be accessed by threads explicitly given permission to run in the corresponding privileged SMV. Therefore, we consider a memory view to be *secure*.

We define three abstract operations for defining the composition of an SMV:

- *Register(SMV,MD)*: registers memory domain *MD* as part of *SMV*'s memory view.
- *Grant(SMV,MD,P)*: grants *SMV* the capability to access memory domain *MD* with access privilege *P*.
- *Revoke(SMV,MD)*: revokes *SMV*'s capabilities to access memory domain *MD*.

We categorize the privileges P of an SMV to access a memory domain into four operations:

- *Read*: An SMV can read from the memory domain.
- *Write*: An SMV can write to the memory domain.
- *Execute*: An SMV can execute in the memory domain.
- *Allocate*: An SMV can allocate/deallocate memory space in the memory domain.

The access privileges to each of the memory domains for an SMV can be different. Two SMVs can reference the same memory domain but the access privileges can differ. The programmer can set up the SMV's privileges to access memory domains in the way needed for the application at hand. For example, multiple threads sharing the same security context can be assigned to the same SMV to parallelize the workload (LO). To minimize an application's attack surface, the programmer can assume the main parent thread to be the master thread of a program. All the permission modifications must be done by the master thread and are immutable by child threads. The SMV model considers any access to a memory domain without proper privileges to be an *SMV invalid* memory access. We implemented the privilege enforcement at the OS kernel level and detail the design in Section 4.5.

3.3 `SMVthread`

An `SMVthread` is a thread that strictly follows the privileges defined by an SMV to access memory domains. `SMVthread`s run in the memory view defined by an SMV and cannot change to other SMVs. While the popular `pthread`s have to trust all `pthread`s running in the same memory space, `SMVthread`s distrust other `SMVthread`s by default. `SMVthread`s – unlike `pthread`s – must *explicitly* share access to the intrinsically shared memory space with other `SMVthread`s. We designed `SMVthread`s to partially share the memory space with

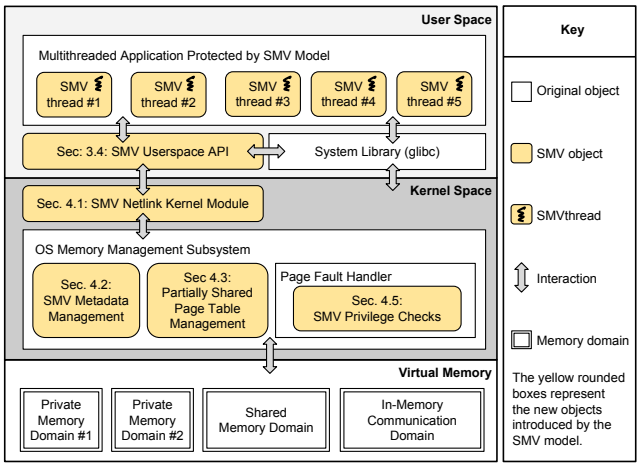

Figure 1: SMV architecture.

other `SMVthread`s according to the policy specified by the programmer through the API. Section 4.3 explains the implementation of the partially shared page tables for `SMVthread`s. `SMVthread`s are `glibc`-compatible, meaning that our `SMVthread`s can directly invoke the library functions in `glibc`. `SMVthread`s can cooperate with `pthread`s through all the synchronization primitives defined by the `pthread`s API. For SMV management, privileged `SMVthread`s have to invoke the SMV API to set up the memory boundaries for least privilege enforcement. `pthread`s can access the whole process address space. Changing such accesses would hamper the correctness of legacy programs that do not require any memory segregation (backward compatibility). While possible, programmers are advised *against* mixing `SMVthread`s and `pthread`s in one process when an application requires isolation as `pthread`s will have unrestricted access to all memory of the process.

3.4 SMV API: User Space Library

We implemented our SMV model as a user-space library that offers an API to support partially shared memory multithreading programming in C and C++. Table 2 summarizes the primary SMV API with descriptions of the main functions. For instance, a programmer can use `memdom_create` to create a memory domain and `memdom_alloc` to allocate memory blocks that are only accessible by `SMVthread`s running in the privileged SMVs. Each memory domain and SMV has a unique ID assigned by the SMV model in the system. `SMVthread`s are integrated with `pthread`s for easier synchronization and every `SMVthread` thus also has an associated `pthread_t` identifier. Note that casting an `SMVthread` to a `pthread` does not bypass the privilege checks. The SMV interface allows programmers to structure the process memory space into distinct memory domains with different privileges for `SMVthread`s and to manage the desired security policy. Furthermore, our library provides options for programmers to automatically override related function calls to significantly reduce the porting efforts. For example, `pthread_create` can be automatically replaced by `smvthread_create`, which internally allocates a private memory domain for the newly created `SMVthread`. Similarly, when an `SMVthread` calls `malloc`, the library allocates memory in the calling thread's private memory domain.

3.5 SMV Architecture

The SMV architecture consists of two parts: a user space programming interface and a kernel space privilege enforcement mechanism. Figure 1 gives an overview. In short, a user space application can call the SMV API to use the SMV model. In the OS kernel, the SMV kernel module is responsible for exchanging the messages between the user space component and the kernel memory management subsystem. We added SMV metadata management to the OS memory management subsystem to record the memory access privileges for the SMVs. We modified the page table management logic to support partially shared page tables and added the SMV privilege checks to the page fault handler that enforces the memory access control.

With the user space interface and the support from the OS kernel, applications can explicitly structure the intrinsically shared process memory space into distinct memory domains with different access privileges without any hardware modifications. Therefore, our approach can be run directly on today's commodity hardware (NH).

3.6 Application Examples

The SMV model allows privilege separation of individual components and data regions in an application. We present one example of the popular design model in general multithreaded applications and two concrete application examples of how the SMV model can protect applications by organizing the process address space with different privileges for threads (GF and EU).

3.6.1 Producer-Consumer Model

First, the SMV model can support the common producer-consumer model with strict memory isolation while maintaining efficient data sharing. Figure 2 illustrates how the SMV model can secure interacting components according to a *generic* producer-consumer model that is employed by all the applications in PARSEC we evaluated. In this example, the `SMVthread`s run in the process address space that contains four SMVs and six memory domains. The SMVs confine the memory access privileges of `SMVthread`s according to the security policy. In this case, the queue domain is the shared memory domain for *all* `SMVthread`s to cooperate with each

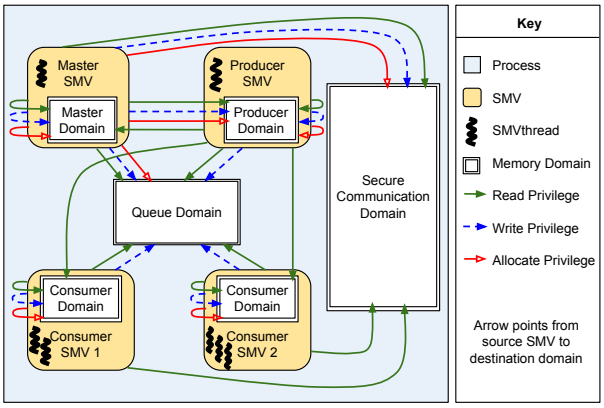

Figure 2: Security-enhanced producer/consumer model with fine-grained memory protection domains.

Table 2: List of primary SMV API.

SMV API	Description
`int smv_main_init (bool allow_global)`	Initialize the main process to use the SMV model. If `allow_global` is true, allow child threads to access global memory domains. Otherwise distrust all threads by default.
`int memdom_create (void)`	Creates a new memory domain, initializes the memory region, and returns the kernel-assigned memory domain ID.
`int smv_create (void)`	Creates a new SMV and returns the kernel-assigned `smv_id`.
`pthread_t smvthread_create (int smv_id, (void*)func_ptr, struct smv_data* args)`	Creates an `SMVthread` to run in the SMV specified by `smv_id` and returns a `glibc`-compatible `pthread_t` identifier.
`void* memdom_alloc (int memdom_id, unsigned long size)`	Allocates a memory block of `size` bytes in memory domain `memdom_id`.
`void memdom_free (void* data)`	Deallocates a memory block previously allocated by `memdom_alloc`.
`int memdom_priv_grant (int memdom_id, int smv_id, int privs)`	Grants the privileges `privs` to access memory domain `memdom_id` for SMV `svm_id` and returns new privileges.
`int memdom_priv_revoke (int memdom_id, int smv_id, int privs)`	Revokes the privileges `privs` to access memory domain `memdom_id` from SMV `svm_id` and returns new privileges.
`int memdom_kill (int memdom_id)`	Deletes the memory domain with `memdom_id` from the process.
`int smv_kill (int smv_id)`	Deletes the SMV with `smv_id` from the process.

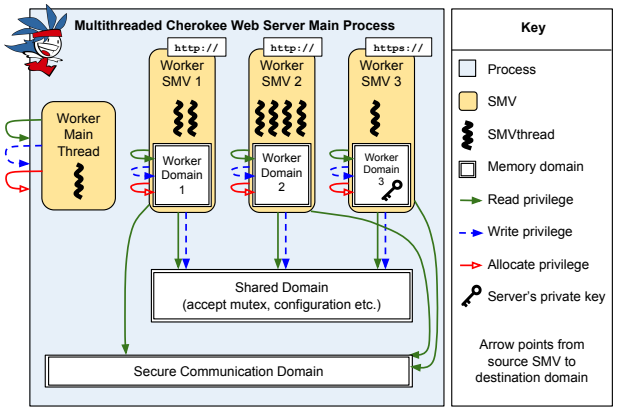

Figure 3: Security-enhanced Cherokee web server.

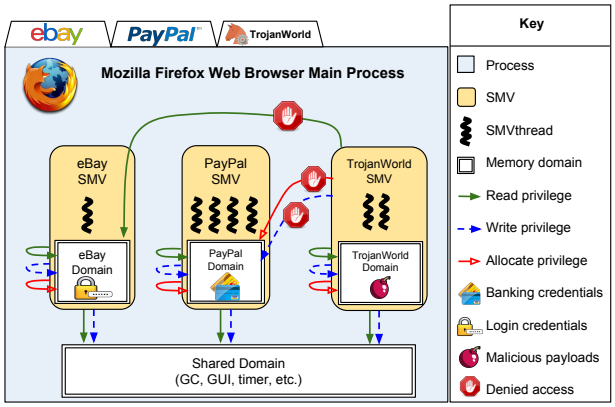

Figure 4: Security-enhanced Firefox.

other, but any write or allocate request from the producer SMV to the consumer domains is prohibited; only reads are permitted. The secure communication domain works as a one-way communication channel for the master `SMVthread` to transmit data to the consumer `SMVthreads` and is inaccessible to the producer `SMVthread` due to the restricted privileges of the producer SMV. In this case, the SMV model strictly enforces memory access boundaries, constraining memory safety bugs to the current component's memory view.

3.6.2 *Case Study: Cherokee Web Server*

Cherokee [12] is a high-performance and light-weight multithreaded web server. To isolate connections, Cherokee uses worker threads to handle incoming requests stored in per-thread connection queues. One worker thread handles all the requests coming from the same connection. However, only one worker thread on the server needs to be compromised to leak sensitive information. To provide an alternative for isolating server workers in different processes, we show how the SMV model can compartmentalize the process memory into memory domains and provide reasonable isolation for the multithreaded Cherokee web server. As shown in Figure 3, the SMV model defines the memory boundaries for worker `SMVthreads` and enforces the memory access privileges to protect the server. The SSL connections are handled only by the `SMVthreads` running in SMV 3 that have the privilege to access worker domain 3, which contains the server's private key. If `SMVthreads` in SMV 2 (handling only HTTP requests) make any attempt to access the private key, the SMV model will reject such invalid memory accesses because of insufficient privileges. In this way, when an exploited worker thread attempts to access memory in an invalid domain, the SMV model detects such invalid accesses and stops further attacks triggered by the memory bugs (e.g., CVE-2004-1097). The original `pthread` Cherokee server does not have this security guarantee since all the threads can access the complete process address space (with unanimously shared permission). We show how accessing invalid memory domains is prevented by the SMV model in Section 5.4.

3.6.3 *Case Study: Mozilla Firefox Web Browser*

The SMV model allows multithreaded web browsers such as Firefox and its JavaScript engine SpiderMonkey to achieve strict compartment isolation enforced by hardware protec-

tion, preventing one malicious origin from accessing sensitive data such as bank accounts hosted by another origin. Figure 4 presents an example of how the SMV model can isolate browser tabs in SMVs based on the same-origin policy [37]. With SMVs, the malicious origin TrojanWorld cannot escape from its compartment to access the PayPal banking account (add recipient account by allocating memory or transfer money to attacker's account by writing to memory) or read the user credentials hosted by eBay. Such strong isolation guarantees inspired Google to design Chrome to use process isolation for its rendering process.

4. IMPLEMENTATION

This section details the OS kernel level implementation of the SMV model and discusses its security guarantees. We modified the Linux kernel version 4.4.5 for the x64 architectures to support the SMV model. Table 3 summarizes the component sizes in our prototype.

4.1 SMV Communication Channel

We developed an SMV loadable kernel module (LKM) that allows the user-space SMV API to communicate with our kernel using the Netlink socket family. Once loaded, the SMV LKM is effectively part of the kernel. The SMV LKM works as a dispatcher in the SMV model that sanitizes the messages from the user space SMV API and invokes SMV-related kernel functions.

Security guarantee. The attacker cannot replace our SMV LKM with a malicious SMV LKM to perform a man-in-the-middle attack and escalate permissions for a given `SMVthread`. Such a system-wide change requires the attacker to have root privilege on the system.

4.2 Metadata Management

To efficiently maintain the state of the processes that have `SMVthreads`, we added two major objects to the OS kernel. (1) `memdom_struct`: memory domain metadata for tracing the virtual memory area and the memory domains mappings. (2) `SMV_struct`: the SMV privilege metadata for accessing memory domains. These kernel objects cooperate with each other to maintain the fine-grained privilege information of each SMV in a process.

Security guarantee. The metadata is allocated in kernel memory space and is not mutable by any user space programs without proper privileges through our API. Memory bugs in user space programs cannot affect the integrity of the metadata stored in kernel memory. One of the main sources of kernel 0-day attacks is the use of uninitialized bytes in kernel memory (e.g., CVE-2010-4158) that allows local users to read sensitive information. The SMV model sanitizes the metadata by initializing objects to avoid any potential information leakage from this added attack surface. Our kernel inherits the original kernel's garbage collection system using reference counting to ensure that the additional metadata does not create any dangling pointers.

Table 3: Summary of component sizes.

	LOC†	Source files	Protection level
SMV API	781	6	user space
SMV LKM	443	2	kernel
SMV MM‡	1,717	24	kernel

† Lines of code computed by `cloc`.
‡ SMV MM stands for SMV memory management, which is integrated into the OS memory management subsystem as we show in Figure 1.

4.3 Partially Shared Memory Space

In the SMV model, `SMVthreads` can be perceived as *untrusted* tasks by default. Therefore, our kernel has to *partially* separate the kernel objects; it also maintains the consistent process address space for the SMV model. Overall, our kernel: (1) separates the memory space of SMVs by using a page global directory (`pgd_t`) for each SMV; (2) frees memory for all SMVs when one `SMVthread` frees the process memory; (3) loads thread-private `pgd_t` into the CR3 register during a context switch.

All `SMVthreads` in a process share the *same* `mm_struct` that describes the process address space. Our kernel allocates one `pgd_t` for each SMV in a process and stores all `pgd_ts` in a process's `mm_struct`. `SMVthreads` use their private page tables to locate memory pages, yet their permissions to the same page might differ. Note that we designed SMVs to protect thread stacks as well. To ensure the integrity of the process memory space, the page tables of all SMVs need to be updated when the kernel frees the process page tables or when `kswapd` reclaims page frames. The original kernel avoids reloading page tables during a context switch if two tasks belong to the same process (thus using the same `mm_struct`). We modified our kernel to reload page tables and flush all TLB entries if one of the switching threads is an `SMVthread`. Note that processors equipped with tagged TLBs could mitigate the flushing overhead. However, SMVs do not rely on this hardware optimization feature in order to function correctly (NH).

Based on our extensive experiments, we found that using different `mm_structs` to separate the address space for threads is overkill and could significantly impact the performance for practical applications (LO). This is because all the memory operations related to `mm_struct` need to be synchronized in an aggressive manner in order to maintain the consistent process address space for all threads (e.g., rotating the `vm_area_struct` red-black tree). Using the `clone` syscall without `CLONE_VM` flag to isolate a thread's address space from its parent is another approach. However, this approach has two main drawbacks. First, the kernel creates a new `mm_struct` for the new thread if `CLONE_VM` is not set. This leads to frequent synchronization and imposes overhead. Second, debugging (e.g., GDB [15]) and tracing memory activity (e.g., Valgrind [43]) become extremely difficult: GDB has to be constantly detached from one process and then attached to another in order to debug a parallel program; Valgrind does not support programs with `clone` calls. In contrast, using the same `mm_struct` preserves the system-wide process address space assumption and allows the kernel to separate process address space for threads efficiently.

Security guarantee. The security features of the partially shared memory space rely on the protection guaranteed by the original kernel. The memory management subsystem in the kernel space is completely unknown to user space programs. The attacker has to exploit the permission bits of the page table entries (PTEs) for a thread to break the security features provided by our kernel. We argue that this kind of exploit is highly unlikely without serious DRAM bugs such as rowhammer [22].

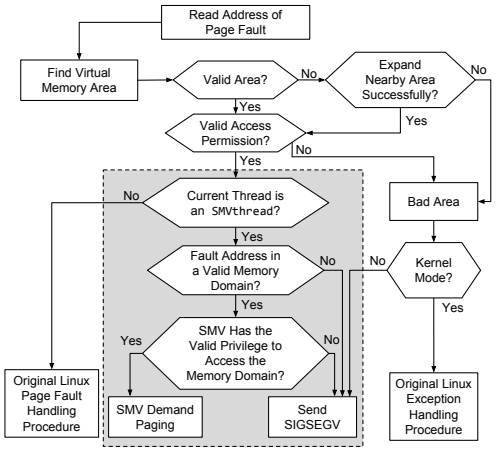

Figure 5: Page fault handler flow chart. The SMV kernel performs additional privilege checks (marked in the gray box).

4.4 Forking SMVthreads

The SMV API uses `pthread_create` to create a regular `pthread` and signals the kernel to convert the `pthread` to an `SMVthread` before the `SMVthread` starts execution. The kernel instructs the `SMVthread` to use the private page tables defined by the SMV that the `SMVthread` runs in. Once an `SMVthread` is created, the kernel turns on the `using_smv` flag stored in the process's `mm_struct` so that future memory operations must go through additional privilege checks.

To simplify porting efforts, the SMV API provides an option to override all `pthread_create` calls and automatically allocate private memory domains for each `SMVthread`.

Security guarantee. The `mm_struct` of a thread is allocated in kernel space and used solely by the kernel. There are no interfaces that allow user space programs to directly or indirectly modify the memory descriptor. This strong isolation between user and kernel space is guaranteed by the trusted OS kernel. In addition, the atomic fork procedure ensures that the attacker cannot intercept the fork procedure and steal the memory descriptor for the malicious thread.

4.5 Page Fault Handler

Figure 5 shows the flow chart of the page fault handler in our kernel. The additional checks are surrounded by the gray box with a dotted line. Our kernel kills the `SMVthread` that triggers an SMV invalid (cf. Section 3.2) page fault by sending a segmentation fault signal. For the privileged `SMVthreads`, our kernel performs *SMV demand paging* to efficiently handle the page faults.

Indeed, since the SMVs use private page tables to separate SMVs' memory views, using the original demand paging routine for `SMVthreads` is insufficient as the page fault handler only updates the page tables for the current SMV, which causes inconsistent process address space. To solve this problem, our kernel tracks all the faulted pages of a process in the *SMV shadow page tables*. The page fault handler deals with faults by using the SMV shadow page tables and then copies the page table entry of the fault from the shadow page tables to the running `SMVthread`'s page tables. Note that one process has only one set of shadow page tables, which only serve as quick reference with no permission implications when `SMVthreads` locate a memory page.

Security guarantee. The page fault handler cannot be accessed, changed, or abused by the attacker as it resides in the lowest level of the software stack. The PTE bits force invalid memory accesses to be trapped to the kernel for the additional privilege checks. To access a privileged memory region, the attacker must first get around the page fault handler. However, such a scenario is infeasible because the kernel memory management subsystem must intervene and prepare the data page before the attacker can access the privileged memory region.

5. EVALUATION

The goal of our evaluation is to demonstrate that *the SMV model has all four desired requirements* when enforcing least privilege memory views for multithreaded applications in practice. We show that the SMV model supports different types of multithreaded programs with flexible policies (GF), requires minimal code changes for legacy software (EU), requires no hardware modifications (NH), and incurs negligible runtime overheads while supporting complex thread interactions and extremely intensive memory allocation/free calls in parallel (LO).

5.1 Experiment Setup

Environment. We measured the performance of our SMV model on a system with Intel i7-4790 CPU with 4 cores clocked at 2.8GHz and 16GB of RAM for our modified x86 64-bit Linux kernel 4.4.5 Ubuntu 14.04.2 SMP (NH). The benchmarks are compiled into two versions: `pthread` and `SMVthread`.

Example policy. `SMVthreads` cannot access privileged memory domains without being explicitly granted the proper privilege. To test this security guarantee in all of our experiments, the number of domains was set to $N + 1$, where N is the number of worker threads and the additional domain serves as a global pool for threads to securely share data. Each worker has its own private memory domain that can only be accessed by itself. We do not claim that the proposed policy is optimal but instead focus on the mechanics to enforce the policy. Setting up alternative policies is possible (GF).

5.2 Robustness Test

To examine the robustness, we tested our modified Linux kernel with the Linux Test Project (LTP) [25] developed and maintained by IBM, Cisco, Fujitsu, SUSE Red Hat, Oracle and others. Specifically, we used the `runltp` script in the LTP package to test the memory management, filesystem, disk I/O, scheduler, and IPC. All stress tests completed without error. We did not observe any system crashes.

5.3 Inspecting Isolation

The SMV model treats invalid memory accesses as segmentation faults. Suppose an attacker's thread triggers a segmentation fault by accessing an invalid memory domain on purpose. The main process will crash to prevent further information leakage. Our SMV library provides detailed memory logs to the programmer. Listing 1 shows an example of the memory activity log. For crashes due to wrong isolation setup, the logs can help the programmer immediately identify the `SMVthread` that accessed the invalid

Listing 1: Kernel log obtained by `dmesg` command.

```
1 [smv]Created memdom 2, start addr: 0x00f0f000, end addr: 0x00f10000
2 [smv]SMVthread pid 11157 attempt to access addr 0x00f0f0e0 in memdom 2
3 [smv]Addr 0x00f0f0e0 is protected by memdom 2
4 [smv]Read permission granted to SMVthread pid 11157 in SMV 2
5 [smv]SMVthread pid 11155 attempt to access addr 0x00f0f260 in memdom 2
6 [smv]SMV 1 is not in memdom 2
7 [smv]Detected INVALID memory reference to: 0x00f0f260
8 [smv]INVALID memory request issued by SMVthread pid 11155 in SMV 1
9 [smv]<6>chorekee[11155]: segfault at f0f260 ip 00007f09ba7d6656 error 4
```

protection domain and subsequently rectify the object compartmentalization. In addition, our library provides detailed stack traces for debugging. The logs and stack traces are unreadable by the attacker when debugging mode is disabled. A binary compiled without debugging option makes it impossible for an attacker to learn about memory activity.

5.4 Security Evaluation

To further understand how the SMV model offers strong intra-process isolation, we systematically discuss the security guarantees described in Section 4.

Trusted computing base. The TCB of the SMV model contains the SMV LKM and SMV MM with kernel level protection (cf. Table 3). The SMV API is untrusted and resides in user space as system library. The attacker may try to perform an SMV API call with a malicious intent to escalate permissions for an `SMVthread`. The SMV LKM sanitizes all user space messages sent into the kernel and verifies that the `SMVthread` executing the API call has the correct permissions for the requested change. The attacker may attempt to leverage the misuse of the SMV API to invalidate the memory isolation guarantee provided by the SMV model. Therefore, the security of the application relies on the correctness of the memory isolation setup. Once the memory boundaries are defined, all `SMVthread`s must follow the memory access rules defined by the programmer. Note that unprivileged users without root permission cannot compromise the SMV LKM (cf. Section 4.1 security guarantee).

The SMV model also relies on the privilege level enforcement imposed by the original Linux kernel to make sure that the attacker cannot tamper with the SMV model operating in the kernel space. To bypass the kernel protection, the attacker must hijack the page tables of a privileged thread or modify the metadata stored in the kernel space. The original Linux kernel ensures the integrity of the metadata and memory descriptors for all threads in the system. Using wrong page tables or metadata will cause a thread to be killed once the kernel detects the tainted kernel data structures. Thus, it is impossible for the attacker to exploit the metadata of any thread without kernel 0-day vulnerabilities (cf. Section 4.2 security guarantee).

In addition to the software TCB, the SMV model also relies on the hardware's correctness. The hardware vendors perform significant correctness validation. We believe that the security features offered by sound hardware are unlikely for the attacker to subvert (cf. Section 4.3 security guarantee). Given the extremely small source code base (less than 2000 LOC), we believe that the SMV's TCB could be formally verified.

TOCTTOU attack: stealing page tables. The attacker may attempt to steal the page tables of a privileged thread by hijacking its memory descriptor. We consider an oracle attacker who knows precisely when and how to launch a time of check to time of use (TOCTTOU) attack to steal the page tables of a privileged thread. If the attack succeeds, the attacker's malicious thread will use the hijacked page tables and read sensitive data in the privileged memory domain before the thread crashes. Assume the attacker can fork threads up to the system limit with the objective to hijack the page tables of an about-to-run privileged thread in the fork procedure, which is the only point for the attacker to exploit the `pgd_t` pointer. However, the malicious thread has to wait until the privileged `SMVthread` finishes the page tables setup in order to request the kernel to prepare its unprivileged page tables. Therefore, the attacker cannot intercept the fork procedure and steal the page tables. We conducted an experiment where 1,023 malicious `SMVthread`s tried to hijack the page tables of a privilege `SMVthread`. During the one million runs of the security test, every `SMVthread` used the correct page tables for its memory view (cf. Section 4.4 security guarantee).

Effectiveness of the SMV model. Listing 1 shows the kernel log when an invalid memory access is detected by the SMV model. In this example, the unprivileged `SMVthread` pid 11155 in SMV 1 tries to access memory in the privilege memory domain that stores the server's private key, which is only accessible by `SMVthread` pid 11157 in SMV 2. At line 5, the attempt to read the invalid memory domain triggers the page fault. The kernel rejects the invalid memory request by sending a segmentation fault signal to the unprivileged `SMVthread` pid 11155 at line 9, stopping the unprivileged `SMVthread` from accessing the server's private key. The privilege checks *cannot* be bypassed because the reference monitor is implemented entirely in the page fault handler, and arbitrary page table manipulation is beyond the attacker's scope (cf. Section 4.5 security guarantee).

5.5 PARSEC 3.0 Benchmarks

Overview. The multithreaded PARSEC benchmarks include several emerging workloads with non-trivial thread interaction. Both data-parallel and pipeline parallelization models are covered in the benchmarks with coarse to fine granularity. We used all benchmarks characterized in [7] covering all application domains that were originally multithreaded using the standard `pthread`s. The evaluated benchmarks all employ the producer-consumer pattern (cf. Section 3.6.1) that is pervasive in systems programs and parallelization models. We used the *parsecmgnt* tool in the PARSEC package to run the benchmarks with minimum number of threads set to four for the large inputs as defined by the benchmarks.

Assessment of porting effort. We ported the PARSEC benchmarks by replacing each `pthread` with an `SMVthread` running in its own SMV with a private memory domain. In each program, the main program allocates a shared memory domain to store the working set for `SMVthread`s. The porting procedure consisted of three parts: (1) including the header files to use the SMV API, (2) setting up memory domains and SMVs in the main program, and (3) replacing the `pthread`s with `SMVthread`s. All these changes required *only 2 LOC* changes as the SMV API eliminates the refactoring burden (EU). We needed to add only 1 line to include the header file and another to initialize the main process to use the SMV model. The SMV API automatically intercepts `pthread_create` and `malloc` and replaces them with `smvthread_create` and `memdom_alloc` calls. Therefore, each `SMVthread` could automatically allocate memory in its pri-

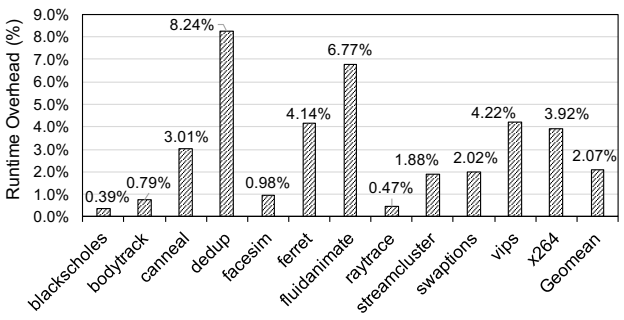

Figure 6: Runtime overhead of the SMV model for the multithreaded applications in the PARSEC benchmark suite.

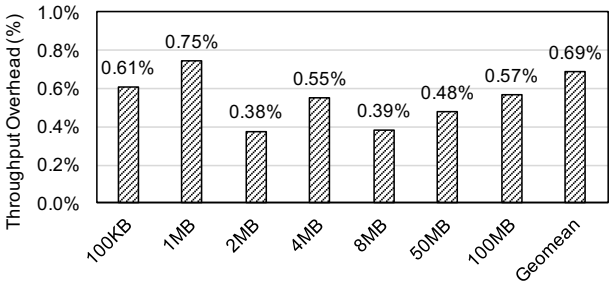

Figure 7: Throughput overhead of Cherokee server.

vate memory domain (cf. Section 3.4).

The security-enhanced PARSEC benchmarks demonstrate a general case that could be applied to any multi-threaded programs written in C/C++ (GF), even with the presence of extremely intensive memory allocation/free calls in parallel. In addition, the intra-process isolation can help prevent attacks that arbitrarily modify data on the stack using malicious threads, e.g., ROP-based attacks. One study has shown that an attacker can perform ROP and use gadgets (16 payloads is enough for $>$ 80% of GNU coreutils [16]) to achieve Turing completeness. Note that ASLR/stack canaries have been proven ineffective to protect against information leakage [20]. With SMV, programmers can secure the system with few changed lines of source code while also handling nontrivial thread interaction, if needed.

Performance. Figure 6 shows the runtime overhead of the ported PARSEC benchmarks with 10 runs for each program. The results show that the SMV model incurs negligible runtime overhead. The overall geometric mean of the runtime overhead is only 2.07% (LO) and the maximum of the runtime overhead occurs for `dedup` due to the huge amount of page faults and the highly intensive parallel memory operations.

5.6 Cherokee Web Server

Overview. The original Cherokee server uses a per-thread memory buffer system for resource management to isolate threads from remote connections. We leveraged the SMV model to provide Cherokee with the OS level privilege enforcement for different server components. Then we compared the throughput of our security-enhanced Cherokee with the original Cherokee.

Assessment of porting effort. We enhanced the security of the Cherokee version 1.2.104 server as illustrated in Section 3.6.2. The user-space SMV library automatically replaced `pthread_create` with `smvthread_create` to create `SMVthreads` for the workers to handle client requests. Each `SMVthread` worker ran in its own SMV with a private memory domain which is inaccessible by other workers. All other shared objects such as the mutex are allocated in a shared memory domain and accessible by all workers. We modified *only 2 LOC* of Cherokee to enforce the least privilege memory access with the SMV model (EU). We believe that the negligible porting effort demonstrates the practicability of the SMV model to protect real-world applications.

Performance. We used ApacheBench to measure the server throughput for the original and security-enhanced Cherokee. Both versions of Cherokee hosted two kinds of web content: (a) social networking web pages, and (b) large streaming files. Based on the total transfer size per page reported in Alexa top one million websites [19], we tested web page sizes from moderate amount of content to abundant media objects (100KB to 8MB). We also evaluated the performance of both servers hosting large streaming files (50MB and 100MB) to show the practicability of our security-enhanced Cherokee. The Cherokee server process created 40 worker threads by default to handle client requests. The client initiated the ApacheBench for 100,000 requests with concurrency level set to four (matches the number of cores). We conducted the experiment 20 times for each object size and present the results in Figure 7. Overall, the SMV model reduced throughput by only 0.69% in exchange for strictly enforcing a least privilege security policy (LO).

We also ported the popular Apache `httpd-2.4.18` with only 2 LOC (EU). Using Apache as a file sharing server (GF) to host large objects with size of 10MB, 50MB, 100MB, and 1GB we conducted the same experiment. Overall, SMVs reduced the throughput of the `httpd` server by only 0.93% (LO). As Cherokee already presents the case for web servers, we exclude the details for Apache `httpd` due to space limitation.

5.7 Mozilla Firefox Web Browser

Overview. The developers of modern web browsers have made tremendous efforts to ensure resource isolation. In 2011, Firefox introduced an abstraction called "compartments" for its JavaScript engine SpiderMonkey to manage JavaScript heaps with security in mind [45]. However, the isolation is not enforced by any mechanism stronger than the compartments' logical boundaries. As a result, any memory corruption can still lead to serious attacks. Here we demonstrate that the SMV model can be easily deployed to protect Firefox's JavaScript engine from memory corruption by confining each compartment to access only its private and the system compartments.

Assessment of porting effort. Firefox uses threads for UI rendering, processing network packets, monitoring browser status, handling JavaScript jobs, etc. In our evaluation, we replaced Firefox 45.0 SpiderMonkey's NSPR (Netscape Portable Runtime) threads with `SMVthreads` running in a private memory domain by adding a new thread type named `PR_SMV_THREAD` to the NSPR library. SpiderMonkey creates 8 threads (1 thread per core + 4 excess threads) in total. We

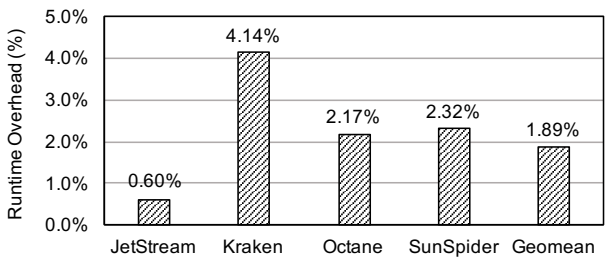

Figure 8: Runtime overhead of security-enhanced Mozilla Firefox web browser.

modified *only 12 LOC* of the entire Firefox source to use the SMV model. Although we designed a per-tab isolation policy, the workloads of individual JavaScript benchmark suites are run in the same tab. For example, JetStream executes 40 benchmark programs in the same tab. The performance numbers faithfully report the overhead for privilege checks as each memory page reference is monitored by our page fault handler.

Performance. We evaluated our security-enhanced Firefox with four popular JavaScript benchmarks and report the numbers in Figure 8. The overall geometric mean for all benchmarks is only 1.89%. The performance numbers report the overhead when Firefox performs additional privilege checks for SpiderMonkey's helper threads. We believe that such negligible overhead numbers allow efficient and strong isolation for multithreaded browsers to be used in practice and provide the Mozilla team an alternative to the ongoing multi-process Firefox *e10s* project [14].

5.8 Limitations

The low performance overhead for checking privileges in the SMV model builds on the virtual memory protection at page granularity. At this point, the SMV model does not guard against unprivileged memory references within the same page as the kernel relies on page table entry (PTE) permission bits. However, poorly organized data structures mixing privileged and non-privileged data within a region are intrinsically insecure and avoided by real-world software (e.g., Hoard [6] memory allocator, connection buffers in Cherokee, worker pools in Apache `httpd`, compartments in Firefox). Therefore, SMVs can be seamlessly integrated into modern software, eliminating the chances for threads to unintentionally access the same page while enforcing memory boundaries at kernel level. Software monitors for byte-granularity protection has inevitably high overhead (e.g., decentralized information flow control systems) since the memory boundary is neither supported by hardware nor the kernel subsystem, making every memory load/store instruction a candidate for a privilege check. In contrast, page-granularity offers strong memory isolation and superior performance with hardware/kernel support.

Although SMVs cannot protect against malicious library threads once they are installed on the system (requires root privilege, which is out of scope), a user can compile any third-party threading libraries to use SMVs, as we demonstrated in the PARSEC benchmark with GThread in `vips` and RTThread in `raytrace`.

6. RELATED WORK

Techniques for achieving intra-process isolation have been studied for decades. In this section, we summarize and compare the related work following a more detailed breakdown.

Memory safety is the goal of many proposals, as memory corruption is the root cause of various well-known software vulnerabilities. We refer the reader to Nagarakatte et al. [32] and Szekeres et al. [42] for two surveys on memory safety. In short, solutions for complete memory safety do not handle intra-process privilege separation problem and impose significant cost for practical systems (cf. LO).

The first generation privilege separation techniques focus on partitioning a process into single-process components. Provos et al. [33] were the first to manually partition OpenSSH by running components in different processes and coordinating them through inter-process communication (IPC). Privtrans [9] automated the retrofitting procedure for legacy software by partitioning one program into a privileged monitor process and an unprivileged slave process with just few programmer-added annotations. Wedge [8] extended the idea of privilege separation to provide fine-grained privilege separation with static capabilities, which was improved by Dune [5] through the Intel VT-x technology (cf. NH) for better performance and by Salus [40] for dynamic security policy. The disadvantage of these first generation techniques is that they lack support for multiple computing entities within the same compartment (cf. LO). This limitation hurts performance of multithreaded programs and restricts the usability of these solutions in practice.

The second generation privilege separation technique Arbiter [47] allowed multiple threads to run in the same compartment. However, Arbiter still faces similar limitations on parallel memory operations and their evaluation does not use multithreaded benchmarks that have intensive memory operations to demonstrate the system's parallelism, even though the design aims at concurrent execution for threads (cf. LO). We identify two major causes of the limitation on Arbiter's parallelism. First, the highly serialized memory management in their user-space library incurs inevitable runtime overhead of up to 400%. Second, the design choice of separating `mm_structs` forces their kernel to aggressively synchronize the global process address space for every thread's memory descriptors. The synchronization costs increase when an application performs intensive memory operations or generates a huge amount of page faults. These limitations on parallelism manifest themselves when running real-world applications (e.g., Firefox) with large inputs (e.g., web server hosting a 100MB file). However, the largest input in the authors' evaluation is only 1MB. In addition, programmers are on their own to partition applications as the solution does not provide assistance in retrofitting applications (cf. EU). As we showed in this paper, the SMV model addresses the limitations of the first and second generation privilege separation techniques without sacrificing security or parallelism.

OS-level abstraction mandatory access control solutions such as SELinux [39], AppArmor [4], and Capsicum [48] protect sensitive data at process/thread granularity. However, fine-grained privilege separation for software objects (e.g., arrays) within a process is not supported in these techniques. On the other hand, SMVs tackle issues for intra-process data protection with capabilities. PerspicuOS [13] separates privileges between trusted and untrusted kernel

components defined by kernel developers using an additional layer of MMU. Such an intra-kernel design does not facilitate intra-process privilege separation as SMV does for user-space applications (cf. GF and EU). These security policies are orthogonal to the SMV memory policies and SMVs can be used in conjunction with these techniques to gain additional inter-process protection.

Decentralized information flow control (DIFC) systems allow programmers to associate secrecy labels with data and enforce information flow to follow security policies. HiStar [51] is an OS that fundamentally enforces DIFC that could likely address intra-process isolation. However, HiStar is not based on a general OS kernel such as Linux and thus cannot be incrementally deployed to commodity systems. Moreover, the applications have to be completely rewritten in order to use HiStar (cf. GF and LO). Thus, the solution is infeasible for legacy software (e.g., Firefox) in practice. Flume [23] focuses on process-level DIFC for OS-level abstractions (e.g., files, processes) in UNIX but it does not handle intra-process privilege separation within a multithreaded application. Laminar [34] supports multithreaded application running in its specialized Java virtual machine (cf. GF). However, the additional layer in the software stack and its dynamic checker incur significant runtime overhead (cf. LO). As noted in Section 5.8, byte-granularity checkers in DIFC systems incur high performance overhead in practical applications that have intensive memory operations (cf. LO).

Software-based fault isolation (SFI) [26, 46] isolates software components within the same address space by constructing distinct fault domains for code and data. SFI prevents code from modifying data or jumping to code outside of a fault domain. Native client [3, 50] utilizes SFI with x86 hardware segmentation for efficient reads, writes, control-flow integrity [1], and component isolation. However, the untrusted code is statically associated with a specific fault domain as the approach does not provide simple means of implementing a dynamic and flexible security policy for practical multithreaded applications (cf. EU and LO). In contrast, SMVs offers solutions for programmers to structure the protected memory regions in a dynamic and non-hierarchical manner.

Language-based techniques utilize safe language semantics to provide isolation for applications written in type-safe languages (e.g. [10, 21]) and implement information flow control for objects within a process (e.g. [11, 30, 31]). However, the vast majority of legacy software are still written in an unsafe language for efficiency. As a result, programmers need to completely rewrite their legacy software using safe languages (cf. GF). Ribbons [18] is a programming model developed entirely for user space that provides fine-grained heap isolation for multithreaded applications. While the access privileges of threads are tracked pair-wise between domains hierarchically in user space in Ribbons, the SMV model leverages the OS memory management subsystem to organize the access privileges of threads systematically in kernel space at negligible overhead (cf. EU and LO).

Special hardware support and virtualization technologies is another line of research that seeks for strong isolation of program secrets. Flicker's [28] significant overhead due to its intensive use of the TPM chip (cf. NH) makes it impractical for performance-critical applications (cf. LO). Although TrustVisor [27] mitigates the overhead by a hypervisor and a software-based TPM chip, the system is impractical for applications that require multiple compartments with different capabilities (cf. GF). Fides [41] points out the limitations in TrustVisor and improves it by supporting more flexible secure modules with a dual VM architecture on top of its special hypervisor. Hypervisors can be used for guest OSs (e.g. SMV OS kernel) on a shared host while SMVs (providing a richer API) directly run on bare metal at full speed (cf. GF and LO). The additional software level in the hypervisor introduces overheads as the VMM intervenes for the guest OSs page tables, causing TLB cache misses. Recent studies [2, 17, 29] by Intel indicate that hardware support for secure computing will become available on mainstream X86 environments in the near future. Intel Software Guard Extensions (SGX) is a mechanism to ensure confidentiality and integrity (but not availability) of a trusted unprivileged software module under an untrusted OS with limited trusted hardware. SGX protects one component from possible interaction using an "enclave" enforced by hardware. Although the goal of Intel SGX is similar to SMVs, our pure-software solution allows SMVs to be adopted by any OSs that have MMU subsystems with commodity hardware (cf. NH). Loki's [52] tagged memory architecture, CODOMs' [44] tagged pages, and CHERI's [49] capability registers can isolate modules into separate domains with efficient access protection check logic. But these approaches require hypothetical hardware support which make them incompatible with commodity systems (cf. NH).

7. CONCLUSIONS

We have presented the design, implementation, and evaluation of SMVs, a programming abstraction that allows efficient memory compartmentalization across concurrent threads. SMVs yield a comprehensive architecture with all four desired requirements – *genericity and flexibility*, *ease of use*, *no hardware modifications*, and *low runtime overhead* – for efficient fine-grained intra-process memory separation in multithreaded applications. Our performance evaluation demonstrates that the SMV model imposes negligible overhead in exchange for greatly improved security guarantees, enforcing intra-process isolation for concurrent threads. The runtime overhead of the multithreaded benchmark PARSEC for using the SMV model is only 2.07% overall with only 2 LOC changes. For popular web servers, the reduction in throughput is only 0.69% overall for Cherokee and 0.93% overall for Apache `httpd`. Both applications required only 2 LOC changes. We also showed that the real-world web browser Firefox can be easily ported to the SMV model with only 1.89% runtime overhead overall, requiring only 12 LOC modifications to the large code base (13M LOC). The simplicity of the porting effort allows legacy software to be quickly adapted to the SMV model. In summary, we believe that the SMV model can greatly reduce the vulnerabilities caused by improper software component isolation and encourages more research on the efficient and practical intra-process isolation for general multithreaded applications.

8. ACKNOWLEDGEMENTS

This work was supported by NSF TC-1117065, NSF TWC-1421910, and NSF CNS-1464155. P. Eugster was partly supported by European Research Council under grant FP7-617805 "LiVeSoft – Lightweight Verification of Software".

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU based Attestation and Sealing. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. ACM.

[3] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 355–366, New York, NY, USA, 2011. ACM.

[4] AppArmor. https://wiki.ubuntu.com/AppArmor.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[8] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[9] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[10] C. Bryce and C. Razafimahefa. An Approach to Safe Object Sharing. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 367–381, New York, NY, USA, 2000. ACM.

[11] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for Usable Information Flow Control in Aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[12] Cherokee Web Server. http://cherokee-project.com/.

[13] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM.

[14] Multiprocess Firefox. https://developer.mozilla.org/en-US/Firefox/Multiprocess_Firefox.

[15] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[16] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association.

[17] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1, New York, NY, USA, 2013. ACM.

[18] K. J. Hoffman, H. Metzger, and P. Eugster. Ribbons: A Partially Shared Memory Programming Model. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 289–306, New York, NY, USA, 2011. ACM.

[19] Interesting stats based on Alexa Top 1,000,000 Sites. http://httparchive.org/interesting.php.

[20] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., Aug. 2015. USENIX Association.

[21] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 1–11, New York, NY, USA, 2007. ACM.

[22] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[23] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[24] A. Kurmus and R. Zippel. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1366–1377, New York, NY, USA, 2014. ACM.

[25] Linux Test Project. http://sourceforge.net/projects/ltp/.

[26] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[27] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[28] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[29] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[30] A. Mettler, D. Wagner, and T. Close. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed Systems Symposium*, NDSS 2010. Internet Society, 2010.

[31] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[32] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 190–208, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[33] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[34] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

[35] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept 1975.

[36] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. *Commun. ACM*, 17(7):388–402, July 1974.

[37] Same-origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[38] SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[39] SELinux. https://wiki.centos.org/HowTos/SELinux.

[40] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel Support for Secure Process Compartments. *EAI Endorsed Transactions on Security and Safety*, 15(3), 1 2015.

[41] R. Strackx and F. Piessens. Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[42] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[43] Valgrind. http://valgrind.org/.

[44] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 469–480, Piscataway, NJ, USA, 2014. IEEE Press.

[45] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental Memory Management in a Modern Web Browser. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 119–128, New York, NY, USA, 2011. ACM.

[46] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[47] J. Wang, X. Xiong, and P. Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 361–373, Santa Clara, CA, July 2015. USENIX Association.

[48] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security 2010*, pages 29–46, 2010.

[49] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[50] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, May 2009.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[52] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.