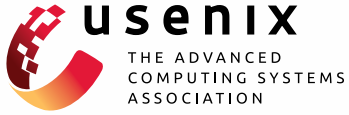

Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications

Jun Wang, ***The Pennsylvania State University;*** **Xi Xiong,** ***Facebook Inc. and The Pennsylvania State University;*** **Peng Liu,** ***The Pennsylvania State University***

https://www.usenix.org/conference/atc15/technical-session/presentation/wang_jun

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications

Jun Wang, Xi Xiong[†1], Peng Liu
Pennsylvania State University, Facebook Inc.[†]

Abstract

Threads in a multithreaded process share the same address space and thus are implicitly assumed to be mutually trusted. However, one (compromised) thread attacking another is a real world threat. It remains challenging to achieve privilege separation for multithreaded applications so that the compromise or malfunction of one thread does not lead to data contamination or data leakage of other threads.

The Arbiter system proposed in this paper explores the solution space. In particular, we find that page table protection bits can be leveraged to do efficient reference monitoring if data objects with the same accessibility stay in the same page. We design and implement Arbiter which consists of a new memory allocation mechanism, a policy manager, and a set of APIs. Programmers specify security policy through annotating the source code. We apply Arbiter to three applications, an in-memory key/-value store, a web server, and a userspace file system, and show how they can benefit from Arbiter in terms of security. Our experiments on the three applications show that Arbiter reduces application throughput by less than 10% and increases CPU utilization by 1.37-1.55×.

1 Introduction

While multithreaded programming brings clear advantages over multiprocessed programming, the classic multithreaded programming model has an inherent security limitation, that is, it implicitly assumes that all the threads inside a process are mutually trusted. This is reflected by the fact that all the threads run in the same address space and thus share the same privilege to access resources, especially data.

However, one thread attacking another thread of the same application process is a real world threat. Here are a few examples: (1) For the multithreaded in-memory key/value store Memcached [9], it has been shown that many large public websites had left it open to arbitrary access from Internet [2], making it possible to connect to (a worker thread of) such a server, dump and overwrite cache data belonging to other threads [7]. In addition, vulnerabilities [11, 10] could be exploited by an adversary (*e.g.,* buffer overflow attack via CVE-2009-2415) so that the compromised worker thread can arbitrarily access data privately owned by other threads. (2) For the multithreaded web server Cherokee [3], an attacker could exploit certain vulnerabilities (*e.g.,* format string CVE-2004-1097) to inject shellcode and thus access the private data of another connection served by a different thread. Meanwhile, logic bugs (*e.g.,* Heartbleed [8]) might exist so that an attacker can fool a thread to steal private data belonging to other threads. (3) For the multithreaded userspace file system FUSE [6], logic flaws or vulnerabilities might also allow one user to read a buffer that contains private data of another user, which violates the access control policy. This is especially critical for encrypted file systems built upon FUSE (*e.g.,* EncFS [5]), wherein data can be stored as cleartext in memory and a malicious user could enjoy a much easier and more elegant way to crack encrypted files than brute force.

A common characteristic of the above applications is that they may concurrently serve different users or clients, which represent distinct principals that usually do not fully trust each other. This characteristic directly contradicts the "threads-are-mutually-trusted" assumption. Therefore, a fundamental multithreaded application security problem arises, that is, *how to retrofit the classic multithreaded programming model so that the "threads-are-mutually-trusted" assumption can be properly relaxed?* In other words, could different principal threads have different privileges to access shared data objects so that the compromise or malfunction of one thread does not lead to data contamination or data leakage of another thread?

1.1 Prior Work and Our Motivation

From a programmer's point of view, we identify two kinds of privilege separation problems. The first problem is to split a monolithic application into least-privilege compartments. For example, an SSH server only requires root privilege for its *monitor* (listening to a port and performing authentication), rather than the *slave* (processing user commands). Since the two parts are usually closely coupled, developers in the old days simply put the two into one program. Due to the emergence of buffer overflow and other relevant attacks against the root privileged part, however, this monolithic program design is no longer appropriate. Separation of the two parts into different privileged processes with IPC mechanisms in between (*e.g.,* via pipes) becomes a more appropriate approach. Actually, OpenSSH has already adopted this approach.

The second problem is to do fine-grained privilege separation in multithreaded applications. As introduced earlier, threads in a multithreaded program were implicitly assumed to be mutually trusted. However, the evolving

[1] work was done while this author was at Pennsylvania State University.

of multithreaded applications tends to break this assumption by concurrently serving different principals. Usually the principals do not fully trust one another.

The first problem has been studied for many years [29, 27, 19, 15, 14]. Provos *et al.* [27] pioneered the methodology and design of privilege separation. Privtrans [15] can automatically partition a program into a privileged monitor and an unprivileged slave. Wedge [14] combines privilege separation with capabilities to do finer-grained partitioning. For the second one, however, there are no systematic research investigations that we are aware of.

This paper focuses on the second privilege separation problem. Our goal is to apply least privilege principle on (shared) data objects so that a data object can be read-writable, read-only, or inaccessible to different threads at the same time and, more importantly, to require minimum retrofitting effort from programmers. First of all, let's look at existing mechanisms to see whether they can be applied to solve this problem.

1) Process isolation. Process isolation is the essential idea behind existing approaches to the first privilege separation problem. OpenSSH [27] and Privtrans [15] leverage process address space isolation while using IPC to make the privileged part and the unprivileged part work together. However, neither of them handles data object granularity. In addition, when there are many principal threads, IPC might become very inefficient. Wedge [14] advances process isolation with new ideas. It creates compartments with default-deny semantics and maps shared data objects into appropriate compartments. However, Wedge is proposed to address the first privilege separation problem, which has very different nature from the problem we consider, as shown in Table 1. Due to these differences, Wedge's all-or-nothing privilege model with default-deny semantic is not very applicable to a multithreaded program, wherein threads by default share lots of resources. To apply Wedge on our problem, one still needs to address the challenges considered in this paper.

Manually retrofitting a multithreaded program to use multiple processes is possible. However, commodity shared memory mechanisms, such as `shm_open` and `mmap`, do not allow one thread to specify the access right of another thread on the shared memory. Alternatively, designing a sophisticated one-on-one message passing scheme (*e.g.,* using Unix socket) can enforce more control on data. However, the programming difficulty and complexity (*e.g.,* process synchronization, policy handling and checking) could be much higher and thus requires lots of retrofitting effort from programmers.

Another notable idea is to redesign an application from scratch using a multi-process architecture, as what is done in Chrome [4]. However, one of our quick survey reveals that over 80% of existing web servers are multithreaded. It is impractical to redesign all those applications that are already multithreaded.

1st PS Problem (OpenSSH [27], Privtrans [15], Wedge [14], *etc.*)	**2nd PS Problem** (Arbiter)
Sequential invocation of compartments with different privileges	Concurrent execution
Only privileged process/thread can access sensitive data	Data shared among different (unprivileged) principal threads
Static capability policy	Dynamic (label) policy

Table 1: Different assumptions of 1st and 2nd privilege separation (PS) problem

2) Software fault isolation. Address space isolation puts each process into a protection domain, but does not do finer-grained isolation inside an address space. Software fault isolation [30, 17] did an innovative work on making a segment of address space as a protection domain by using software approaches like a compiler. Nevertheless, it is difficult for SFI to map program data objects (*e.g.,* array) into a protection domain: address-based confinement and static instrumentation cannot easily deal with dynamically allocated data. LXFI [24] instruments `kmalloc` so that the principal and address information of dynamic kernel data objects are made aware to the reference monitor. However, this is done only to kernel modules and kernel data. In addition, LXFI focuses on integrity and does not check memory reads due to performance reasons. However, our goal is to prevent both unauthorized reads and writes. Therefore, we need to catch invalid reads as well.

3) Other related mechanisms. We investigate four additional types of related mechanisms to see whether they can handle our problem. (a) OS abstraction level access control has been extensively studied (*e.g.,* SELinux [23], AppArmor [1], Capsicum [31]). However, these mechanisms treat a process/thread as an atomic unit and do not deal with data objects "inside" a process. So a granularity gap exists between these techniques and our goal. (b) HiStar [33] is a from-scratch OS design of decentralized information flow control (DIFC). Perhaps HiStar can meet our goal of privilege separation on data objects. However, HiStar does not apply to commodity systems. Besides, to use HiStar to achieve our goal, there still needs to be a major change in the programming paradigm. Flume [20] implements DIFC in Linux. However, it focuses on OS-level abstractions such as processes, files, and sockets and thus does not address the privilege separation problem at data object granularity within a multithreaded program. It can be complementary to the approach proposed in this paper. (c) With the tagged memory in Loki [34] or the permission table lookup mechanism in MMP [32], as new features to the CPU, access to each individual memory word can be checked. Both methods can enforce privilege separation policy on data objects. However, they require architectural changes to commodity CPUs. (d) Language-based

solutions, such as Jif [26], Joe-E [25], and Aeolus [16] can realize information flow control and least privilege at the granularity of program data object. However, they need to rely on type-safe languages like Java. As a result, programmers have to rewrite legacy applications not originally developed in a type-safe language.

1.2 Challenges and Our Approach

We would like to solve this problem in a new way based on this insight: *we find that page table protection bits can be leveraged to do efficient reference monitoring, if the privilege separation policy can be mapped to those protection bits.* We find that this mapping is possible through a few new kernel primitives and a tailored memory management library. However, doing so still introduces three major challenges:

- **Mapping Challenge (C1)** In the current multithreaded programming paradigm, all the threads in the same process share one set of page tables. This convention, however, would disable the needed mapping from privilege separation policy to protection bits.
- **Allocation Challenge (C2)** To make the protection bits work, data objects that demand distinct privileges cannot be simply allocated onto the same page because this will result in the same access rights. Existing memory management algorithms have difficulty meeting such a requirement because they were not designed to enforce privilege separation.
- **Retrofitting Challenge (C3)** It is challenging to minimize programmers' retrofitting effort to communicate complex privilege separation policies with the underlying system without modifying the source code drastically.

We present Arbiter to address the above challenges. To address the mapping challenge (C1), we associate a separate page table to each thread and create a new memory segment named Arbiter Secure Memory Segment (ASMS) for *all* threads. ASMS maps the shared data objects onto the same set of physical pages and set the page table permission bits according to the privilege separation policy. To deal with the allocation challenge (C2), we design a new memory allocation mechanism to achieve privilege separation at data-object granularity on ASMS. To resolve the retrofitting challenge (C3), we provide a label-based security model and a set of APIs for programmers to make source-level annotations to express privilege separation policy. We design and implement Arbiter based on Linux, including a new memory allocation mechanism, a policy manager, and a set of kernel primitives.

We port three types of multithreaded applications to Arbiter, *i.e.,* an in-memory key/value store (Memcached), a web server (Cherokee), and a userspace file system (FUSE), and show how they can benefit from Arbiter in terms of security. Our own experiences indicate that porting programs to Arbiter is a smooth procedure. The changes to the program source code is 0.5% LOC on average. Regarding performance, our experiments show that the runtime throughput reduction is below 10% and CPU utilization increase is 1.37-1.55×.

2 Overview

2.1 Motivating Examples

Programmers have both *intended privilege separation* and *intended sharing* of data objects when writing multithreaded programs. We classify these intentions into three categories.

- **Category 1**: A data object is intended to be exclusively accessed by its creator thread.

Figure 1(a) shows the request processing code snippet from Cherokee. The data object `buf` is allocated by a worker thread and then used to store the incoming packet. Therefore, this data object belongs to that particular worker thread and other worker threads are not supposed to access it.

- **Category 2**: A data object is intended to be accessed by a subset of threads.

Figure 1(b) and 1(c) show the connection handling code snippets from Memcached. The main thread receives a network request, allocates a data object `item` to store the connection information, selects a worker thread and then pushes the `item` into the thread's connection queue. The worker thread wakes up, dequeues the connection information and handles the request. Ideally, the data object `item` is only intended to be accessed by the main thread and the particular worker thread, excluding any other worker thread.

- **Category 3**: A data object is intended to be shared among all the threads.

This data sharing intention is commonly seen, especially on metadata. For instance, the `struct cherokee_server` and the `struct fuse` store the global configurations of Cherokee and FUSE, respectively, and are intended to be accessible to all the threads.

Overall, Category 1 and 2 are two very representative privilege separation intentions. Unfortunately, there is actually no such enforcement in real world execution environments. Only the intention in Category 3 has been taken care. We propose Arbiter, a general purpose mechanism so that every category is respected.

2.2 Threat Model

We consider two types of threats. First, some threads could get compromised by malicious requests (*e.g.,* buffer overflow attacks, shellcode injection, return-to-libc attacks, ROP attacks). Second, application has certain logic bugs (*a.k.a.* logic vulnerabilities [18] or logic flaws [22]). For example, the logic bug exploited by HeartBleed [8] can potentially lead to a buffer overread attack,

```
process_active_connections(cherokee_thread_t *thd) {
        ...
        buf = (char *) malloc (size);
        ...
        len = recv (SOCKET_FD(socket), buf, buf_size, 0);
        ...
}
```

(a) Cherokee-1.2.2

```
void dispatch_conn_new(...) {
        ...
        CQ_ITEM *item = malloc(sizeof(CQ_ITEM));
        ...
        cq_push(thread->new_conn_queue, item);
        ...
}
```

(b) Memcached-1.4.13 Main thread

```
static void *worker_libevent(...) {
        ...
        item = cq_pop(me->new_conn_queue);
        ...
}
```

(c) Memcached-1.4.13 Worker thread

Figure 1: Motivating examples

which allows an attacker to steal sensitive information of other users from a web server. In reality, both threats can lead to data leakage and data contamination of a victim thread, which usually result in the compromise of end user's data secrecy and integrity. Besides, we assume that the application is already properly confined by well-defined OS level access control policies (*e.g.,* which files the application can access) using SELinux, AppArmor, *etc.* We also assume that the kernel is inside TCB. The fact that the kernel could be compromised is orthogonal to the problem we aim to solve.

2.3 Problem Statement

How to deal with the two types of threats through a generic data object-level privilege separation mechanism so that all of the three categories of how a data object is intended to be accessed by threads can get respected?

2.4 System Architecture

Figure 2 shows the architecture of our system. In Arbiter, threads are created in a new way, resulting in what we call *Arbiter threads*. Arbiter threads resemble traditional threads in almost every aspect such as shared code segment (`.text`), data segment (`.data`,`.bss`), and open files, but they have a new dynamically allocated memory segment ASMS. To give threads different permissions to access the same data object, we maintain a separate page table for each thread and maps the shared data objects on ASMS to the same set of physical pages. To set the needed permissions, protection bits inside each page table will be set up according to the privilege separation policy. In kernel, these are realized by the ASMS Management component, including system call code plus a set of kernel functions, and the corresponding additions to the page fault handling routine. Due to ASMS, two objects with different *accessibility* will be allocated on two different pages. By accessibility, we mean which threads can access an object in what way. However, many pages

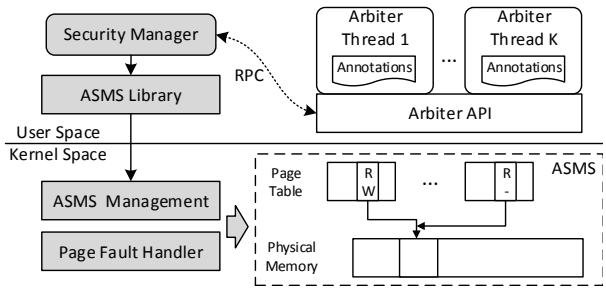

Figure 2: System architecture. Shaded parts indicate Arbiter's trusted computing base (TCB).

	Main Thread	Thread A	Thread B
A's Data `buf`	–	RW	–
B's Data `buf`	–	–	RW
Shared Data `item`	RW	R	R

Table 2: Accessibility generated from Figure 1

could end up with being half empty by doing so. Our solution is to leverage homogeneity, that is, objects with the same accessibility are put into the same page. Such memory allocation is achieved by the ASMS Library.

There are three things a thread needs to go through Arbiter: (1) memory allocation and deallocation, (2) thread creation, and (3) policy configuration. For security purpose, Arbiter threads delegate these operations to the Security Manager running in a different address space via remote procedure calls (RPC).

To specify security policy, programmers will need to make annotations to the source code via the Arbiter API according to our label-based security model. The Security Manager will figure out the permissions at runtime and the page table protection bits will be set up properly before the corresponding data object is accessed by an Arbiter thread.

3 Design

3.1 Accessibility

In our system, accessibility means which threads can access an object in what way. Conceptually, we need to map the aforementioned three categories of intentions onto accessibility before we can enforce fine-grained privilege separation.

Table 2 shows a formally defined accessibility generated from the motivating examples in Figure 1. Accessibility is defined in terms of a set of threads. Given a set of threads $\{th_1, \cdots, th_k\}$, the accessibility of data object x is defined as a *vector* of k elements. For example, the accessibility vector of A's data `buf` is $< \emptyset, RW, \emptyset >$. Two data objects have the same accessibility if and only if they have the same vector in term of all of the k threads.

3.2 Design Goal

At a high level, our goal is that through Arbiter the accessibility originated from the privilege separation intentions can be enforced. This goal boils down to the following three design requirements. (1) From a system's perspective, separate page tables are required in order to enforce accessibility vectors and a synchronized virtual-to-physical mapping is required to make such separation transparent to the threads. (2) From a program's perspective, a smart memory allocation strategy is required in

order to bridge the granularity gap between page-level protection and individual data objects and do it in an efficient way, for which we propose the idea of "same accessibility, same page". These two requirements lead to the kernel-level and user-level design of ASMS (§3.3). (3) From a programmer's perspective, it is important to correctly code accessibility in the program without changing the program drastically. We create a label-based security model and a set of APIs for this purpose (§3.4). In addition, how Arbiter converts accessibility into protection bits is introduced in §3.5. §3.6 discusses the thread creation and context switch issues incurred by our design.

3.3 ASMS Mechanism

Kernel Memory Region Management. To grant threads with different permissions to the shared memory, our initial thought was to leverage the file system access control mechanism `user/group/others` to `mmap` files with allowed open modes so as to realize different access rights. Since this method has to assign a unique UID for each principal thread, however, it would mess up the original file access permission configurations. In addition, `mmap` cannot automatically do memory allocation and configuration for multiple sets of page tables in a single invocation.

We design a new memory abstraction called Arbiter Secure Memory Segment (ASMS) to achieve efficient privilege separation. ASMS is a special memory segment compared to other segments like code, data, stack, heap, *etc.* The difference is that when creating or destroying ASMS memory regions for a calling thread, the operation will also be propagated to all the other Arbiter threads. In other words, ASMS has a synchronized virtual-to-physical memory mapping for all the Arbiter threads, yet the access permissions (page protection bits `Present` and `Read/Write`) could be different. Furthermore, only the Security Manager has the privilege of controlling ASMS. Arbiter threads, in contrast, cannot directly allocate/deallocate memory on ASMS. Neither can they modify their access rights of ASMS data objects on their own.

User-level Memory Management Library. A granularity gap exists between page-level protection (enabled by the per-page protection bits) and individual program data objects. Data objects demanding distinct accessibility can no longer be allocated on the same page. To this end, existing memory allocation algorithms (*e.g.,* dlmalloc [21]) cannot directly work for ASMS. An intuitive solution is to allocate one page per data object. However, this is not preferable mainly because a huge amount of memory will be wasted if the sizes of data objects are much smaller than the page size.

We design a special memory allocation mechanism for ASMS: *permission-oriented* allocation. The key idea is to put data objects with identical accessibility onto the same page, or "same accessibility, same page". When we allocate memory for a new data object x with accessibility vector v, we search for a page containing data objects with the same vector v and put x into that page. If that page is full, we search for another candidate page. If all candidate pages are full, we allocate a new page and put x into it. In practice, we allocate from the system one memory *block* instead of one page per time so as to save the number of system calls. Here a memory block means a contiguous memory area containing multiple pages. Figure 3 demonstrates this idea (further details in §4.1). In this way, both memory waste and performance overhead can be reduced.

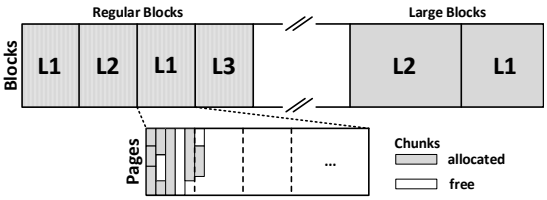

Figure 3: A typical memory layout of ASMS. L1/L2/L3 indicate different accessibility.

3.4 Label-based Security Model

To accommodate programmers' privilege separation intentions, we need a security model for specifying and enforcing accessibility vectors. Our initial attempt was to load the entire accessibility table into the Security Manager as an access control list (ACL) so that it can check and determine each thread's permission for a data object. However, to regulate each thread's capability of making allocation requests (*e.g.,* thread A is not allowed to allocate objects that are accessible by everyone) and to deal with dynamic policies (*e.g.,* thread A first grants thread B permission and later on revokes it), ACL is insufficient and further mechanisms must be employed. It is desirable to have a unified and flexible security model.

To achieve unification and flexibility, we develop a label-based security model wherein threads and data objects are associated with labels so that data access permissions and allocation capabilities can be dynamically derived and enforced. Essentially, it is a special form of "encoding" of the accessibility table. The basic notions and rules follow existing dynamic information flow (DIFC) models [33, 28] with a few adaptations. It should be noted that Arbiter itself is not a DIFC system (see §7 for more discussion).

We use *labels* to describe the security properties of principal threads and data objects. A *label L* is a set that consists of *secrecy categories* and/or *integrity categories*. For a data object, secrecy categories and integrity categories help to protect its secrecy and integrity, respectively. For a thread, the possession of a secrecy category ($*_r$, where $*$ represents the name of a category) denotes its `read` permission to data objects protected by that category; likewise, an integrity category ($*_w$) grants a thread

the corresponding `write` permission. Meanwhile, we use the notion *ownership O* to mark a thread's privilege to bypass security checks on specific categories. A thread that creates a category also owns that category (*i.e.,* has the ownership). Different from threads, data objects do not have ownership.

We define the rules that govern threads' permissions and activities as follows:

RULE 1 – Data Flow: We use $L_A \sqsubseteq L_B$, to denote that data can flow from A to B (A and B represent threads or data objects). This means: 1) every secrecy category in A is present in B; and 2) every integrity category in B is present in A. If the bypassing property of ownership is considered, a thread T can read object A iff: $L_A - O_T \sqsubseteq L_T - O_T$, which can be written as: $L_A \sqsubseteq_{O_T} L_T$. Similarly, thread T can write object A iff: $L_T \sqsubseteq_{O_T} L_A$.

RULE 2 – Thread Creation: Thread creation is another way of data flow and privilege inheritance. Therefore, a thread T is allowed to create a new thread with label L and ownership O iff: $L_T \sqsubseteq_{O_T} L, O \subseteq O_T$. The new thread is not allowed to modify its own labels.

RULE 3 – Memory Allocation: Memory allocation also implies that data flows from a thread to the allocated memory. As the result, a thread T can get a memory object allocated on ASMS with label L iff: $L_T \sqsubseteq_{O_T} L$.

Therefore, one could make the following label assignment to realize the accessibility vectors in Table 2. For instance, Thread *A* can read but not write the Shared Data *item* because of $L_{item} \sqsubseteq_{O_A} L_A$ and $L_A \not\sqsubseteq_{O_A} L_{item}$. Neither can Thread *A* create a thread with the Main Thread's privilege ($O_A \not\subseteq O_{Main}$) nor allocate a forged data *item* ($L_A \not\sqsubseteq_{O_A} L_{item}$). As such, our model unifies permission and capability.

Thread	**Main**	**A**	**B**
label	{}	{mr}	{mr}
ownership	{mr,mw}	{ar,aw}	{br,bw}

Data	**A's Data** `buf`	**B's Data** `buf`	**Shared Data** `item`
label	{ar,aw}	{br,bw}	{mr,mw}

The labels are attached by a programmer to the corresponding threads or data objects through annotating the source code via Arbiter API. Appendix A.1 presents a list of Arbiter API.

3.5 Protection Bits Generation

The Security Manager is responsible for converting labels to page table protection bits. The Security Manager maintains a real-time registry containing label information of every thread and every ASMS memory block. The conversion happens in two occasions: memory allocation and thread creation. First, whenever a thread wants to allocate memory with certain labels, the Security Manager determines the permissions for *every* thread by checking our label model, and then invokes our system calls to construct and configure ASMS memory regions accordingly. Second, when a new thread is created, the Security Manager walks through *every* ASMS memory block, determines the allowed permissions, and initializes the ASMS correspondingly.

Note that in Linux a page table entry is not established until the data on that page is actually accessed. Therefore, the page fault handler will eventually further convert the permissions stored in the flags of ASMS memory regions into page table protection bits (further details in §4.2). As the result, before a data object is accessed by any thread, the page table protection bits would have been set up properly.

3.6 Thread Creation and Context Switch

We identify two options to create an Arbiter thread. Option 1: Conceptually, one can create a new address space for every new Arbiter thread, reconfigure ASMS permissions, and disable copy-on-write for all the other memory segments to retain memory sharing. In this case, although the context switch between two Arbiter threads will lead to TLB flush (which is just like the context switch between two processes), it can be automatically done by existing kernel procedure and requires no further code modification.

Option 2: A possible optimization is to create a new set of page table only for ASMS when creating a new Arbiter thread. Thus only part of the TLB needs to be flushed during context switch between two Arbiter threads. While this can potentially reduce the TLB-miss rate, it would require lots of modifications to the kernel, especially on the context switch procedure to determine the type of context switch, reload the page table for ASMS, and flush the TLB partially.

In sum, there is a trade-off between "TLB-miss overhead" and "how much code modification is needed". Both options have pros and cons. We take the first option and our evaluation shows that the performance overhead is already acceptable.

4 Implementation

We implement Arbiter based on Linux. This section highlights a few implementation details.

4.1 ASMS Mechanism

Kernel Memory Region Management. To properly create or destroy ASMS memory regions in the kernel so as to enlarge or shrink ASMS, we implement a set of kernel functions similar to their Linux equivalents such as `do_mmap` and `do_munmap`. The difference is that when creating or destroying ASMS memory regions for a calling thread, the operation will also be propagated to all the other Arbiter threads. How to configure the protection bits is determined by the arguments passed in from our special system calls (by the Security Manager), including `absys_sbrk`, `absys_mmap`, and `absys_mprotect`. They all have similar semantics to their Linux equivalents, but with additional arguments to denote the permissions.

We add a special flag `AB_VMA` to the `vm_flags` field of the memory region descriptor (*i.e.,* `vm_area_struct`),

which differentiates ASMS from other memory segments. The page fault handler also relies on this flag to identify ASMS page faults. To make sure that only the Security Manager can do allocation, deallocation, and protection modification on ASMS memory regions, we modify related system calls, such as `mmap` and `mprotect`, to prevent them from manipulating ASMS.

User-level Memory Allocation Library. Built on top of our special ASMS system calls is our user-level memory allocation library Memory blocks are sequentially allocated from the start of ASMS. Some data objects might have larger size and cannot fit in a *regular block*. In this case, *large blocks* will be allocated backward starting at the end of ASMS. The pattern of this memory layout is shown in the top half of Figure 3. Inside each block, we take advantage of the dlmalloc algorithm [21] to allocate memory chunks for each data object. The bottom half of Figure 3 depicts the memory chunks on pages inside a block. Further details on our allocation/deallocation algorithms can be found in Appendix A.2.

4.2 Page Fault Handling

A page fault on ASMS typically leads to two possible results: ASMS demand paging and segmentation fault. ASMS demand paging happens when a Arbiter thread legally accesses an ASMS page for the first time. In this case, the page fault handler should find the shared physical page frame and create and configure the corresponding page table entry for the Arbiter thread. The protection bits of the page table entry are determined according to the associated memory region descriptor. In this way, subsequent accesses to this page will be automatically checked by MMU and trapped if illegal. This hardware enforced security check significantly contributes to the runtime performance of Arbiter. An illegal access to an ASMS page will result in a `SIGSEGV` signal sent to the faulting thread. We implement a kernel procedure `do_ab_page` as a subprocedure to the default page fault handler to realize the above idea.

4.3 Miscellaneous

Application Startup. In Arbiter, an application is always started by a Security Manager. A Security Manager first executes and initializes the needed data structures, such as the label registry. Then, it registers its identity to the kernel so as to get privileges of performing subsequent operations on ASMS. We implement a system call `ab_register` for this purpose. Next, the Security Manager starts the application using `fork` and `exec`, and then blocks until a request coming from the Arbiter threads. The application process can create child thread by calling `ab_pthread_create`, which is implemented based on the system call `clone`. The label and ownership of the new thread, if not specified, default to its parent's.

RPC. A reliable RPC connection between Arbiter threads and the Security Manager is quite critical in our system. We implement the RPC based on Unix socket. A major advantage of Unix socket for us is about security: it allows a receiver to get the sender's Unix credentials (*e.g.,* PID), from which the Security Manager is able to verify the identity of the sender. This is especially important in situations where the sender thread is compromised and manipulated by the attacker to send illegal requests or forged information on behalf of an innocent thread.

Authentication and Authorization. The Security Manager needs to perform two actions before processing an RPC: authentication and authorization. Authentication helps to make sure the caller is a valid Arbiter thread. This is done by verifying the validity of its PID acquired from the socket. Authorization ensures that the caller has the needed privilege for the requested operation. For example, RULE 2 must be satisfied for a thread creation request, and RULE 3 must be satisfied for a memory allocation request. If either of the two verifications fail, the Security Manager simply returns the RPC with an indication of security violation.

Futex. Due to our implementation of thread creation, a problem arises with the futexes (*i.e.,* fast userspace mutex) located on data segment (including both `.data` and `.bss`). Multithreaded programs often utilize mutexes and condition variables for mutual exclusion and synchronization. In Pthreads, both of them are implemented using futex. Originally, kernel assigns the key (*i.e.,* identifier) of each futex as either the address of `mm_struct` if the futex is on an anonymous page or the address of `inode` if the futex is on a file backed page. In Arbiter, since data segment is anonymous mapping but the `mm_struct`'s of the Arbiter threads are different, kernel will treat the same mutex or condition variable as different ones. Nonetheless, we can force programmers to declare them on ASMS (which resembles file mapping) that does not have this issue. However, we decide to reduce programmers' effort by modifying the corresponding kernel routine `get_futex_key` and set the key to a same value (*i.e.,* the address of `mm_struct` of the Security Manager). As such, the futex identification problem is resolved.

5 Application

We explore Arbiter's applicability through case studies across various multithreaded applications. We find that the inter-thread privilege separation problem are indeed real-world security concerns. This section introduces our case studies on three different applications: (1) Memcached, (2) Cherokee, and (3) FUSE.

5.1 Memcached

Overview. Memcached [9] is an in-memory data object caching system. It caches data objects from the results of database queries, API calls, or page renderings into memory so that the average response time can be largely reduced. There are mainly three types of threads

in a Memcached process: main thread, worker thread, and maintenance thread. Upon arrival of each client request, the main thread first does some preliminary processing (*e.g.,* packet unwrapping) and then dispatches a worker thread to serve that request. Periodically, maintenance threads wake up to maintain some important assets like the hash table.

Security concern. We identify two potential security concerns. (1) It is reported that a number of large public websites had left Memcached open to arbitrary access from the Internet [2]. This is probably due to the fact that the default configuration of Memcached allows it to accept requests from any IP address plus its authentication support SASL (Simple Authentication and Security Layer) is by default disabled (as Memcached is designed for speed, not security). It has been shown possible to connect to such a server, extract a copy of cache, and write data back to the cache [7]. (2) The vulnerabilities in Memcached [11, 10] could be exploited by an adversary (*e.g.,* buffer overflow attack via CVE-2009-2415) so that the compromised worker thread can arbitrarily access data privately owned by other threads.

Retrofitting. We adapt Memcached to realize the accessibility shown in Table 2. In particular, we assume that a Memcached server is used to serve two applications or two users, A and B. Both A and B privately own their cached data objects that are not supposed to be viewed by the other. For the Shared Data, we make `CQ_ITEM` and a few other metadata read-writable to the main thread but read-only to the worker threads. We slightly change the original thread dispatching scheme so that requests from different principals can be delivered to the associated worker threads. This modification does not affect other features of Memcached.

5.2 Cherokee

Overview. Cherokee [3] is a multithreaded web server designed for lightweight and high performance. Essentially there is only one type of thread in Cherokee: worker thread. Every worker thread repeats the same procedure, that is, it first checks and accepts new connections, adds the new connections to the per-thread connection list, and then processes requests coming from these connections. All the requests coming from the entire life cycle of a connection will be handled by the same thread.

Security concern. (1) An attacker could exploit the vulnerabilities of the Cherokee (*e.g.,* format string vulnerability CVE-2004-1097) to inject shellcode and thus access the data of another connection served by a different thread. (2) Logic bugs might exist in the web server so that an attacker can fool the thread to overread a buffer, which may contain the data belonging to another connection/thread. A recent bug of this type is the Heartbleed bug in OpenSSL [8].

Retrofitting. Our goal is to prevent the threads from accessing each other's private data without affecting the normal functionality. Therefore, we make the buffers allocated for individual connections only accessible by the corresponding thread. Global data structures are made accessible to all the threads, for example, the `struct cherokee_server` which stores the server global configuration, listening sockets file descriptors, mutexes, *etc.*

5.3 FUSE

Overview. FUSE (Filesystem in Userspace) [6] is a widely used framework for developing file systems in user space. Common usages include archive file systems– accessing files inside archives like tar and zip, database file systems– storing files in a relational database or allowing searching using SQL queries, encrypted file systems– storing encrypted files on disk, and network file systems– storing files on remote computers.

When a FUSE volume is mounted, all file system operations against the mount point will be redirected to the FUSE kernel module. The kernel module is registered with a set of callback functions in a multithreaded user space program, which implements the corresponding file system operations. Each worker thread can individually accept and handle kernel callback requests.

Security concern. (1) Logic flaws like careless boundary checking might allow one user to overread a buffer that contains private data of another user. The two users could have very different file system permissions and thus should not share the same set of files. This is especially critical for encrypted file systems (*e.g.,* EncFS [5]), since the intermediate file data is in memory as cleartext. A malicious user can enjoy a much easier and more elegant way to steal data, compared with cracking the encrypted file on disk by brute force. (2) Although the chance is low due to the limited attack surface, we envision a type of attack in which an attacker can compromise a particular thread and inject shellcode. Then the attacker will be able to directly read the data of another user in memory.

Retrofitting. In general, we make the buffers allocated inside `process_cmd()` private to each thread. The global data structure `struct fuse` is shared among all threads, which contains information like callback function pointers, lookup table, metadata of the mount point, *etc.* In addition, we change the thread dispatching scheme from round robin to associating users with threads, which is similar to what we do for Memcached.

5.4 Summary of Porting Effort

Porting these applications to Arbiter was a smooth experience. Actually, most of our time is spent on understanding the source code and data sharing semantics. After that, we define accessibility and devise label assignments accordingly. Finally, we modify the source code, replacing related thread creation and memory allocation functions with Arbiter API. Table 3 summarizes the total LOC and the LOC added/changed for each application.

Application	Total LOC (approx.)	LOC added/changed
Memcached-1.4.13	20k	100 (0.5%)
Cherokee-1.2.2	60k	188 (0.3%)
FUSE-2.3.0	8k	129 (1.6%)

Table 3: Summary of porting effort in the amount of source code change

6 Evaluation

6.1 Protection Effectiveness

As stated in our threat model, we assume that the target application is already properly confined by OS abstraction level access control mechanisms, such as SELinux or AppArmor. To this end, our system can be considered complementary to these OS abstraction level mechanisms. Here our goal is not to evaluate whether our system can achieve the OS abstraction level access control (*e.g.,* preventing a compromised thread from accessing a confidential file). Instead, we want to see under the protection of Arbiter whether a compromised thread can still contaminate or steal the data belonging to another thread.

We assume that an adversary has exploited a program flaw or vulnerability in the three applications ported by us and thus taken control of a worker thread. We simulate various malicious attempts based on the security concerns we presented earlier in §5.

Memcached. We simulate two types of attacks mentioned in §5.1. (1) We simulate an attacker connecting to Memcached via `telnet`. For the vanilla Memcached, the attacker can successfully extract or overwrite any data using the corresponding keys. On the ported Memcached (protected by Arbiter), our attempts to retrieve data belonging to a different user always fail. (2) We then simulate the scenario presented in §5.1 to simulate a buffer overflow attack. We assume that B is an attacker. To simplify simulation, we hard-code our "shellcode" in the source code. Our "shellcode" try to overwrite `CQ_ITEM` and read A's data by traversing the slablist (`(&slabclass[i])->slab_list[j]`). We find that writing to `CQ_ITEM` always fail and traversing the slablist will fail whenever encountering a slab storing A's data.

Note that in both (1) and (2), a failed attempt always triggers a segmentation fault and thus program crash. In practice, the signal handler can be used with Arbiter to deal with such security violations in a more robust way (*e.g.,* sending no response back or dropping the connection). In our experiments, we simply omit this part.

Cherokee. (1) We first simulate the format string attack. We add our "shellcode" to the source code to get another thread's data via the `header` and `buffer` field of the connection structure (`struct cherokee_connection`), which is referenced by the victim thread's active connection list (`&thd->active_list`). We observe that both read and write attempts fail without exception. (2) Then we simulate the logic bug. Particularly, we craft a buffer overread bug by substituting the `buf_size` parameter in the `cherokee_socket_write()` function with a number from our input. When we use a small value for `buf_size`, the buffer overread does not fail in most cases because the adjacent memory is also allocated with the same label. This is tolerable since the attacker only gets the data of his own. When we input a value that is larger than the size of a regular block (*i.e.,* 40KB in our case), the attack always fail. Again, in both (1) and (2), a failure always leads to a segmentation fault in the web server.

FUSE. The simulation of FUSE is very similar to what we do for Cherokee. Arbiter can successfully defeat both (1) logic flaw exploits and (2) code injection attacks.

Counterattacks. We enumerate a few typical counterattacks that are intended to bypass the Arbiter protection.

1) The adversary may want to call `mprotect` to change the permission of ASMS and then access the data.

2) The adversary may attempt to call `ab_munmap` first and then `ab_mmap` to indirectly modify the permission.

3) The adversary may call `fork` or `pthread_create` to create a normal process or thread that is out of the Security Manager's control so as to access the data.

4) The adversary may also want to `fork` a child process and let the child process call `ab_register` to set itself as a new Security Manager. In this way, the adversary hopes to gain full control of the ASMS.

5) The adversary forges a reference and fools an innocent thread to access data on behalf of the adversary.

We try each of the above counterattacks for multiple times, but no one succeeds. The reasons are as below. For 1), it is because Arbiter forbids normal system calls including `mprotect` to operate ASMS. For 2), since the adversary does not have permission to access the data, the Security Manager simply denies the `ab_munmap` request. For 3), unfortunately ASMS will not be mapped to the normal processes or threads. For 4), there do exist ASMS now and the child process does gain full control. However, the ASMS no longer has the same physical mapping. For 5), it would actually have a chance to succeed. However, Arbiter provides an API `get_privilege` which allows the innocent thread to verify if the requesting thread has the necessary permission. As such, Arbiter can still defeat this counterattack. In sum, we believe that within our threat model no counterattack can succeed.

6.2 Microbenchmarks

We build a set of microbenchmarks to examine the performance overhead of Arbiter API. Our experiments were run on a Dell T310 server with Intel Xeon quad-core X3440 2.53GHz CPU and 4GB memory. We use 32-bit Ubuntu Linux (10.04.3) with kernel 2.6.32 and glibc 2.11.1. Since we implement the ASMS Library based on uClibc 0.9.32, we use the same version for comparison on memory allocation. Each result is averaged over 1,000 times of repeat.

Operation	Linux (μs)	Arbiter (μs)	Overhead
(ab_)malloc	4.14	9.09	2.20
(ab_)free	2.06	8.36	4.06
(ab_)calloc	4.14	8.41	2.03
(ab_)realloc	3.39	8.27	2.43
(ab_)pthread_create	91.45	145.33	1.59
(ab_)pthread_join	36.22	41.00	1.13
(ab_)pthread_self	2.99	1.98	0.66
create_category	–	7.17	–
get_label	–	7.65	–
get_ownership	–	7.55	–
get_mem_label	–	7.66	–
ab_null (RPC round trip)	–	5.84	–
(absys_)sbrk	0.65	0.76	1.36
(absys_)mmap	0.60	0.83	1.38
(absys_)mprotect	0.83	0.92	1.11

Table 4: Microbenchmark results in Linux and Arbiter

Table 4 shows the comparison of microbenchmarks. The overhead of memory allocation functions (*e.g.,* `ab_free`) is non-trivial. This is because they have to go through the Security Manager via an RPC round trip, which consists of RPC marshalling, socket latency, *etc.* We find that a pure RPC round trip (`ab_null`) itself already takes 5.84μs, which helps to justify the time consumption of most Arbiter API functions. Due to our implementation of thread creation, we directly use `getpid` to return the thread ID. As the result, `ab_pthread_self` runs even faster than its Linux equivalent. In addition to the RPC latency, the system calls made by the Security Manager also contribute to the API overhead. We examine `sbrk`, `mmap`, and `mprotect` and find that Arbiter incurs 28% overhead on average.

There are two other factors that might affect the overhead of Arbiter API: (1) The number of threads can affect the memory allocation overhead. Figure 4(a) shows that the time consumption of `ab_malloc` is roughly correlated with the number of threads. The time consumption increases by around 5.7% per additional thread. This is because memory allocation on ASMS for one thread is also propagated to other threads. For comparison, we also show the result of `get_label`. This operation does not involve any "propagation" and thus is not affected by the number of threads. (2) The size of allocated ASMS can affect the thread creation overhead. This is because thread creation involves the permission reconfiguration of ASMS. Figure 4(b) shows that the time consumption of `ab_pthread_create` increases along with the size of allocated ASMS (note the logarithmic scale on x-axis). This is also in line with our expectation.

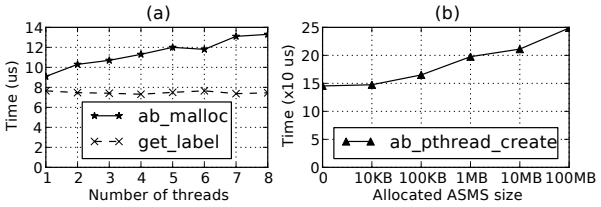

Figure 4: Arbiter API performance regarding number of threads and allocated ASMS size

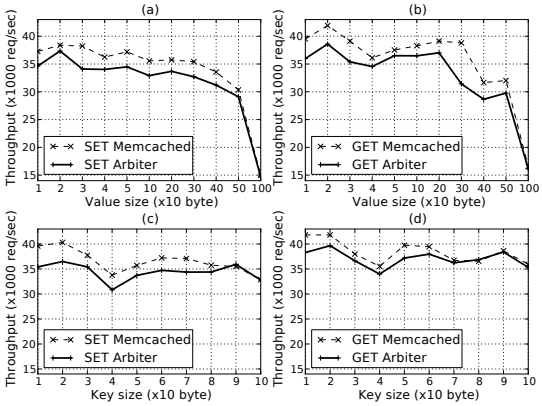

Figure 5: Performance comparison for Memcached

6.3 Application Performance

Memcached. We build a security-enhanced Memcached based on its version 1.4.13 and we use libMemcached 1.0.5 as the client library. We measure the throughput of two basic operations, SET and GET, with various value sizes and key sizes. The results are compared with unmodified Memcached. In Figure 5(a) and 5(b), we anchor the key size to 32 bytes and change the value size. In Figure 5(c) and 5(d), we fix the value size to 256 bytes and adjust the key size. Each point in the figure is an average of 100,000 times of repeat. All together, the average performance decrease incurred by Arbiter is about 5.6%.

Cherokee. We port Cherokee based on its version 1.2.2. We use the `ApacheBench` version 2.3 and static HTML files to measure its performance. First, we measure the influence of file size. We choose files with sizes of 1KB, 10KB, 100KB, and 1MB. Figure 6(a) shows the comparison between vanilla Cherokee and the ported version. The average slowdown is 1.8%. Second, we test the system scalability by tuning the number of threads from 5 to 40. We fix the file size to 1KB during this round of test. The throughput comparison is shown in Figure 6(b). The average performance degradation is around 3.0%. This comparison indicates that running more threads does not necessarily induce more overhead. For each individual test, we set `ApacheBench` to issue 10,000 requests with the concurrency level of 10.

FUSE. We retrofit FUSE based on its version 2.3.0. For the custom userspace file system, we use the exam-

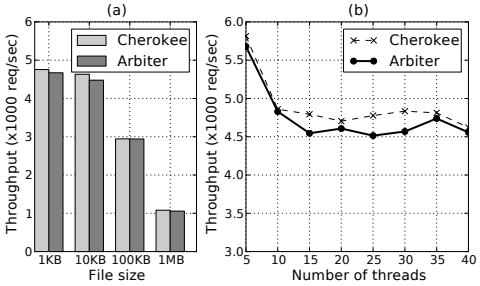

Figure 6: Performance comparison for Cherokee

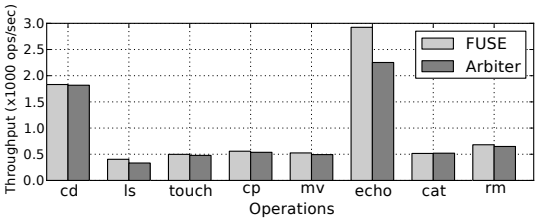

Figure 7: Performance comparison for FUSE

ple implementation `fusexmp` provided by FUSE source package. It simply emulates the native file system. We then select 8 representative commands relevant to file system operations, namely, `cd`, `ls`, `touch`, `cp`, `mv`, `echo`, `cat`, and `rm`. Note that the `echo` command is used to write a 32-byte string to files. Each command is repeated for 10,000 times. Figure 7 shows the comparison between unmodified FUSE and the ported version. On average, the slowdown is 7.4%.

Overall, the application performance overhead is acceptable. This is partially contributed by the fact that the extra cost of Arbiter API calls is amortized by other operations of these programs.

6.4 CPU and Memory Overhead

In addition to the throughput comparison, we further evaluate the CPU cost. As shown in Table 5, Arbiter increases the CPU utilization by 1.29–1.55×. We leverage the CPU time information in /proc/[pid]/stat to do the calculation. We also count the types of labeled objects (not to be confused with runtime instances), shown in the last column of Table 5. Interestingly, the number of labeled objects is roughly correlated with the CPU overhead.

Although our "same accessibility, same page" strategy has already come with much less memory waste than "one object per page", it still incurs some memory overhead. Table 6 shows the average resident memory (RSS) usage of the three applications during the performance test. We measure RSS by checking the `VmRSS` value of /proc/[pid]/status around ten times per second. Given that the policy we used for the three applications are quite typical, we believe real-world memory overhead should be close to the measured overhead.

7 Discussion and Limitations

We believe that Arbiter provides a generic and practical mechanism for inter-thread privilege separation on data objects. Nonetheless, it still has limitations in defending against certain security threats. When two principal users or clients are served by the same thread, Arbiter can no longer enforce privilege separation for the two principals. Thus, programmers have to be very careful dealing with user authentication and thread dispatching to associate principals with appropriate worker threads. To fully address this issue, one possible solution is to have a per-principal-user "virtual" thread to further separate the privileges. We leave this as a future work.

Application	Original	Arbiter	Overhead	Labeled objects
memcached	49.4%	76.7%	1.55×	14
cherokee	58.8%	76.1%	1.29×	8
FUSE	42.3%	58.0%	1.37×	10

Table 5: Comparison of CPU utilization and labeled objects

Application	Original (KB)	Arbiter (KB)	Overhead
memcached	60,664	64,452	6.2%
cherokee	3,916	4,120	5.2%
FUSE	732	760	3.9%

Table 6: RSS memory overhead

One limitation of our implementation is that the user-space memory allocator uses a single lock for allocation/deallocation. Therefore, the processing of allocation and deallocation requests have to be serialized. A finer lock granularity can help to improve parallelism and scalability, such as Hoard [13] and TCMalloc [12]. In fact, Arbiter's memory allocation mechanism inherently has the potential to adopt a per-label lock. We are looking at ways to implement such a parallelized allocator.

Arbiter's security relies on the correctness of privilege separation policy configured by the programmer. However, it may not be that easy to get all the label assignments correct, especially in complex and dynamic deployment scenarios. Actually, DIFC systems also confront similar policy configuration challenges and research efforts have been made to debug DIFC policy misconfiguration [35]. Our system is also able to incorporate a policy debugging or model checking tool that can verify the correctness of label assignments.

Arbiter's security model, including notions and rules, is inspired by DIFC. However, it should be noted that Arbiter does not perform information flow tracking inside a program, mainly due to two observations: (1) For a runtime system approach, tracking fine-grained data flow (*e.g.,* moving a 4-byte integer from memory to a CPU register) could incur tremendous overhead, making Arbiter impractical to use; (2) The fact that information flow tracking can enhance security does not logically exclude the possibility of solving real security problems without information flow tracking. The main contribution of Arbiter is that it provides fine-grained privilege separation for data objects using commodity hardware, while still preserving the traditional multithreaded programming paradigm.

8 Conclusion

Arbiter is a system targeting at fine-grained, data object-level privilege separation for multi-principal multithreaded applications. Particularly, we find that page table protection bits can be leveraged to do efficient reference monitoring if data objects with same accessibility are put into the same page. We find that Arbiter is applicable to a verity of real-world applications. Our experiments demonstrate Arbiter's ease of adoption, effectiveness of protection, as well as reasonable performance overhead.

Acknowledgement

We would like to thank our paper shepherd Xi Wang and the anonymous reviewers, for their insightful feedback that helped shape the final version of this paper.

This work was supported by NSF CNS-1223710, NSF CNS-1422594, and ARO W911NF-13-1-0421 (MURI).

References

[1] Apparmor. http://www.novell.com/linux/security/apparmor/.

[2] Blackhat write-up: go-derper and mining memcaches. http://www.sensepost.com/blog/4873.html.

[3] Cherokee. http://cherokee-project.com.

[4] The chromium projects: Multi-process architecture. http://www.chromium.org/developers/design-documents/multi-process-architecture.

[5] Encfs. http://www.arg0.net/encfs.

[6] Fuse: Filesystem in userspace. http://fuse.sourceforge.net.

[7] go-derper. http://research.sensepost.com/tools/servers/go-derper.

[8] The heartbleed bug. http://heartbleed.com.

[9] Memcached. http://www.memcached.org.

[10] Sa-contrib-2010-098 - memcache - multiple vulnerabilities. http://drupal.org/node/927016.

[11] Security vulnerabilities in memcached. http://www.cvedetails.com/vulnerability-list/vendor_id-9678/producti_id-17294/Memcachedb-Memcached.html.

[12] Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[13] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS* (2000), pp. 117–128.

[14] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 309–322.

[15] BRUMLEY, D., AND SONG, D. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 5–5.

[16] CHENG, W., PORTS, D. R. K., SCHULTZ, D., POPIC, V., BLANKSTEIN, A., COWLING, J., CURTIS, D., SHRIRA, L., AND LISKOV, B. Abstractions for Usable Information Flow Control in Aeolus. In *USENIX ATC '12*.

[17] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *OSDI* (2006).

[18] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium* (2010), pp. 143–160.

[19] KILPATRICK, D. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track* (2003), USENIX, pp. 273–284.

[20] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. SOSP '07.

[21] LEA, D. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[22] LI, X., YAN, W., AND XUE, Y. Sentinel: securing database from logic flaws in web applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012), ACM, pp. 25–36.

[23] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the linux operating system. In *USENIX ATC* (2001).

[24] MAO, Y., CHEN, H., ZHOU, D., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Software fault isolation with api integrity and multi-principal modules. In *In SOSP* (2011).

[25] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-E: A Security-Oriented Subset of Java. In *NDSS '10*.

[26] MYERS, A. C. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), ACM, pp. 228–241.

[27] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), USENIX Association, pp. 16–16.

[28] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: practical fine-grained decentralized information flow control. PLDI '09.

[29] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[30] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. SOSP '93.

[31] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: practical capabilities for unix. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 3–3.

[32] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 304–316.

[33] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. *Commun. ACM 54*, 11 (Nov. 2011), 93–101.

[34] ZELDOVICH, N., KANNAN, H., DALTON, M., AND KOZYRAKIS, C. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 225–240.

[35] ZHAO, M., AND LIU, P. Modeling and checking the security of DIFC system configurations. In *Automated Security Management*. Springer, 2013, pp. 21–38.

A Appendix

A.1 Arbiter API

Figure 8 lists the Arbiter's API, which are used for labeling, threading, and memory allocation. To preserve the multithreaded programming paradigm, the function syntax is fully compatible with the C Standard Library and the Pthreads Library. For example, if a programmer uses `ab_malloc` without assigning any label (`L = NULL;`), it will behave in the same way as libc `malloc`, *i.e.,* allocating a memory chunk read-writable to every thread. This makes it possible for programmers to incrementally adapt their programs to our system.

A.2 ASMS Memory Allocation Algorithm

§3.3 and §4.1 described our permission-oriented allocation mechanism. Here we explain the detailed algorithm shown in Figure 9. For clarity, we omit the discussion on the strategy of memory chunk management adopted from dlmalloc.

- **Allocation** If the size of the data is larger than a regular block size (*i.e.,* threshold), a large block will be allocated using `absys_mmap` (line 5). Otherwise, the allocator will search for free chunks inside blocks with that label (line 7). If there is an available free chunk, the allocator simply returns it. If not, the allocator will allocate a new regular block using `absys_sbrk` (line 12).
- **Deallocation** For a large block, the allocator simply frees it using `absys_munmap` (line 3) so that it can be reused later on. Otherwise, the allocator puts the chunk back to the free list (line 5). Next, the allocator checks if all the chunks on this block are free. If so, this block will be recycled for later use (line 7).

- `cat_t create_category(cat_type t);`
  Create a new category of type `t`, which can be either secrecy category `CAT_S` or integrity category `CAT_I`.
- `void get_label(label_t L);`
  Get the label of a thread itself into *L*.
- `void get_ownership(own_t O);`
  Get the ownership of a thread itself into *O*.
- `void get_mem_label(void *ptr, label_t L);`
  Get the label of a data object into *L*.
- `int ab_pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg, label_t L, own_t O);`
  Create a new thread with label *L* and ownership *O*.
- `int ab_pthread_join(pthread_t thread, void **value_ptr);`
  Wait for thread termination.
- `pthread_t ab_pthread_self(void);`
  Get the calling thread ID.
- `void *ab_malloc(size_t size, label_t L);`
  Allocate dynamic memory on ASMS with label *L*.
- `void ab_free(void *ptr);`
  Free dynamic memory on ASMS.
- `void *ab_calloc(size_t nmemb, size_t size, label_t L);`
  Allocate memory for an array of elements on ASMS with label *L*.
- `void *ab_realloc(void *ptr, size_t size);`
  Change the size of the memory on ASMS.
- `void *ab_mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset, label_t L);`
  Map files to ASMS with label *L*.
- `int get_privilege(pthread_t thread, void *ptr);`
  Query the permission of a thread to accessing memory on ASMS.

Figure 8: List of Arbiter API

```
1  ablib_malloc(sz, L)
2      if sz > threshold then
3          for every member thread do
4              Compute permission
5          Allocate a large block
6          return
7      Search free chunks in blocks with label L
8      if there is an available free chunk then
9          return
10     for every member thread do
11         Compute permission
12         Allocate a regular block
13     return

1  ablib_free ( ptr )
2      if it is a large block then
3          Free the block
4          return
5      Free the chunk pointed by ptr
6      if the whole block is free now then
7          Free the block
8      return
```

Figure 9: ASMS memory allocation algorithm