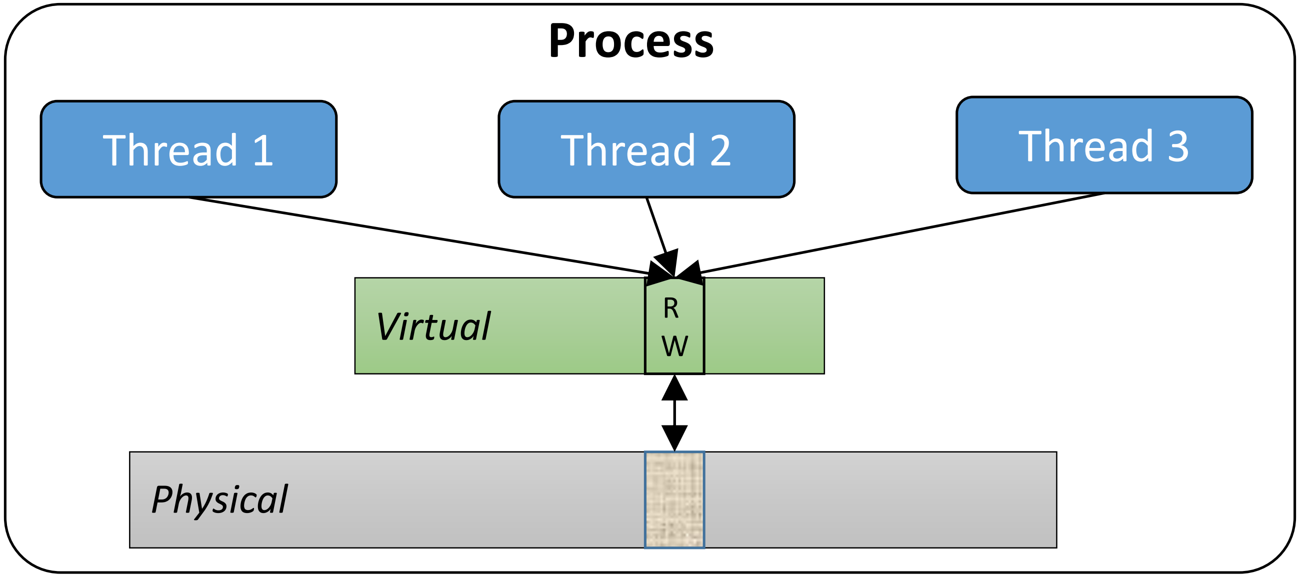
Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications

Jun Wang, Xi Xiong, Peng Liu

PENNSTATE 1855

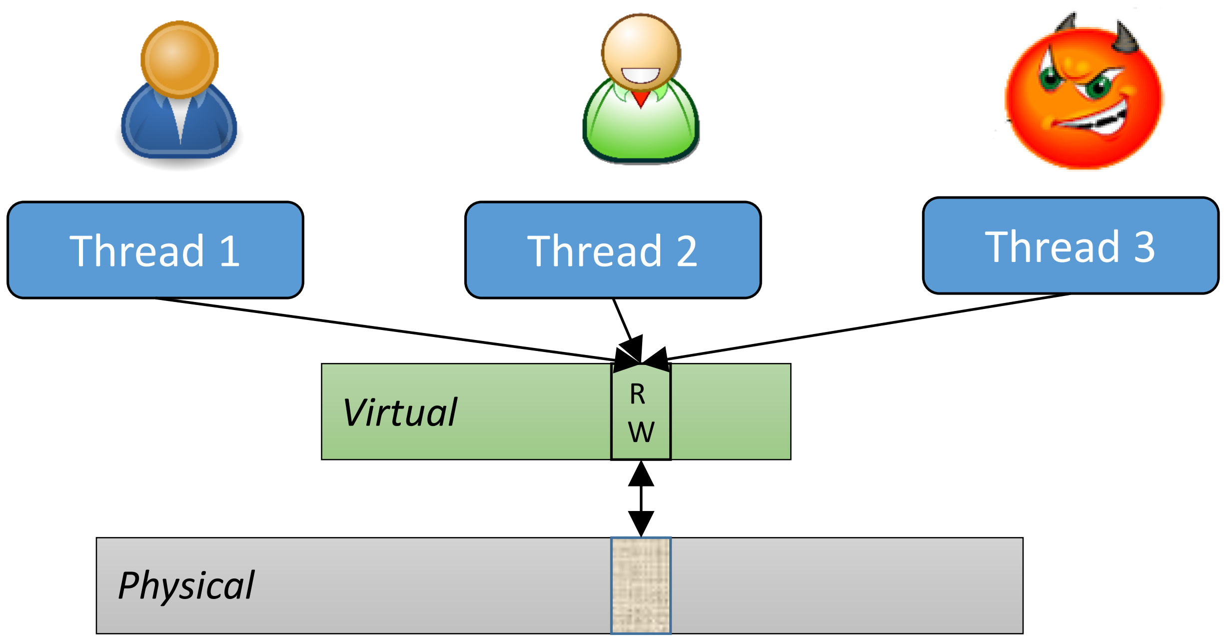
An inherent security limitation in multithreaded programming model

- All the threads inside a process (implicitly) assumed to be mutually trusted:
  - Same address space
  - Same privilege to access recourses, especially data



In reality…

- A multithreaded application can concurrently serve different principals (users or clients) that usually do not fully trust each other.



One thread attacking another is a real world threat

- A compromised (worker) thread can arbitrarily access data privately owned by other threads.

Memcached

- Insufficient user authentication
- Buffer overrun
  CVE-2009-2415

Cherokee

- Format string
  CVE-2004-1097
- Logic bug
  CVE-2014-0160

FUSE

- Logic bug
- Especially critical for encrypted file systems built upon FUSE

In a programmer's perspective

- Both intended privilege separation and intended sharing of data objects when writing programs

Category	Programmer's Intention on data	Possible
1	Privately owned/accessed	X
2	Shared by a subset of threads	X
3	Shared among all the threads	√

- Only the intention in category 3 is attainable…

In a programmer’s perspective

- Category 1 – Privately owned/accessed

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

Cherokee-1.2.2

- Category 2 – Shared by a subset of threads

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

Memcached-1.4.13 Main thread

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

Memcached-1.4.13 Worker thread

Our goal

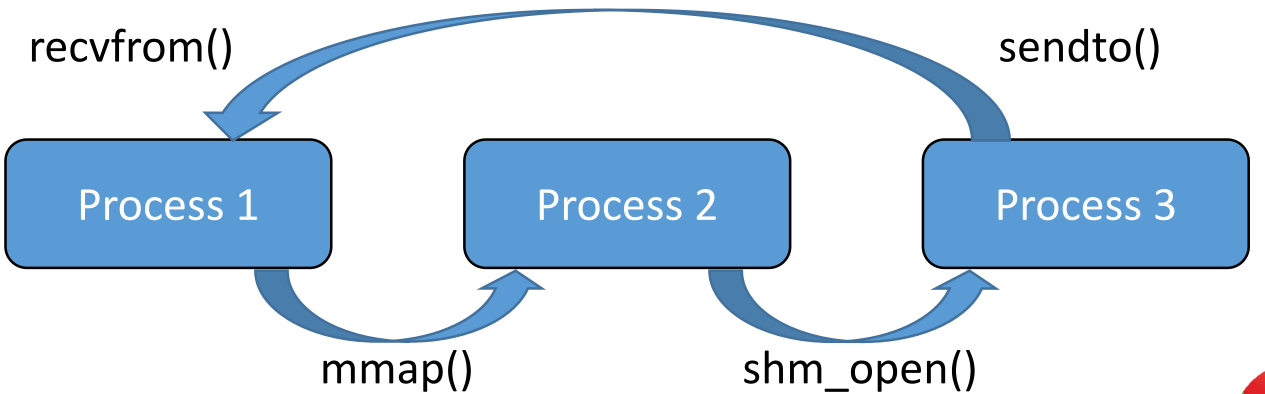
- How to develop a generic data object-level privilege separation mechanism so that all of the three categories of how a data object is intended to be accessed by threads can be achieved?

Outline

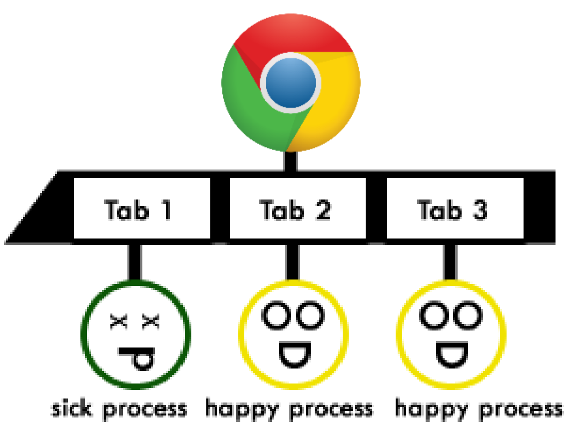
- Motivation
- Challenges and Our Approach
- Design and Implementation
- Evaluation
- Discussion and Limitations
- Conclusion

Approach I – Process Isolation

- Put threads into separate processes
  - Complex IPC design and implementation
    - process synchronization, policy handling and checking

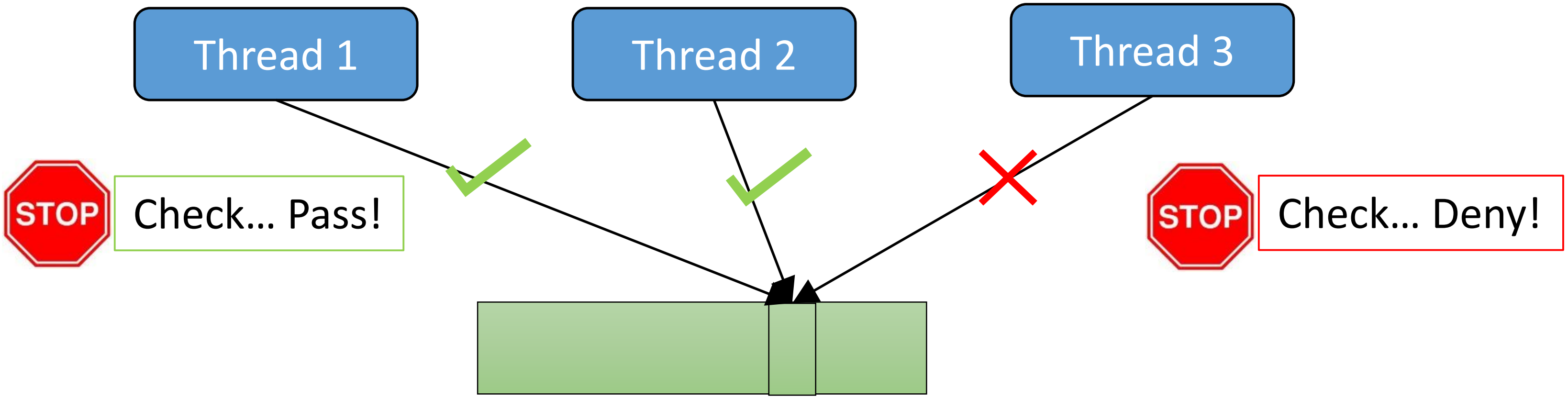


- Multi-process architecture
  - Unpractical for legacy applications
    - 80% web servers are multithreaded



Approach II – Software Fault Isolation

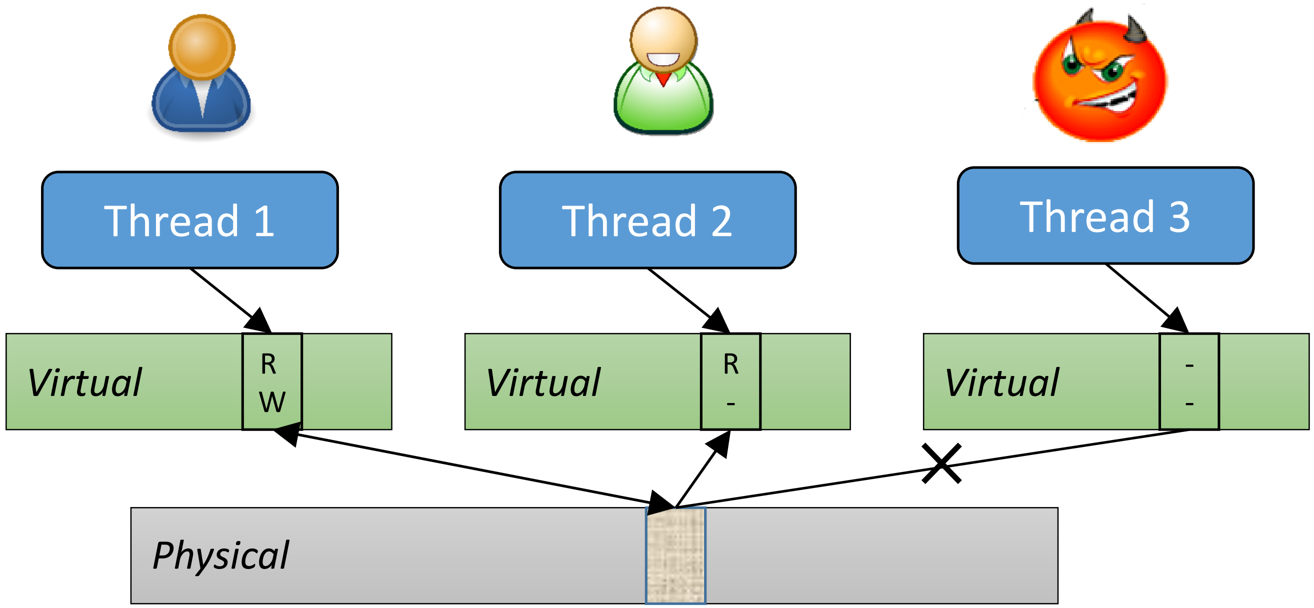
- Approach
  - Programmer annotates source code
  - Compiler translates annotations to runtime checks of memory reads and writes



- However, performance is a serious concern…

Our Idea

- Key Observation:
  - Page table protection bits can be leveraged to do efficient reference monitoring, if the privilege separation policy can be mapped to those protection bits.



Challenges

- Mapping Challenge
  - Shared (single) page table vs "policy-to-protection-bits" mapping
- Allocation Challenge
  - Data objects demanding distinct privileges cannot be simply allocated onto the same page
  - Existing memory management algorithms not applicable
- Retrofitting challenge
  - Minimize programmers' porting effort
  - Policy specification, source code change, etc.

Our Approach: Arbiter

- Associate a separate page table to each thread
- A new dynamic memory segment: ASMS
  - Map shared data objects onto the same set of physical pages and set the page table permission bits according to the privilege separation policy.
- A new memory allocation mechanism to achieve privilege separation at data-object granularity
- A label-based security model and a set of APIs

An Example

	Thread A {pr, pw}	Thread B {pr}	Thread C {}
passwd {pr, pw}	**RW**	**R**	**-**

```
int main() { //thread A
    ...//initialization
    //create thread B and C
    label_t L_B={pr}, L_C={};
    ab_pthread_create(&threadB,...,L_B,{})
    ab_pthread_create(&threadC,...,L_C,{})
    //allocation memory for passwd
    label_t L_passwd={pr,pw};
    passwd=ab_malloc(256,L_passwd);
    ...
}
```

Ported code

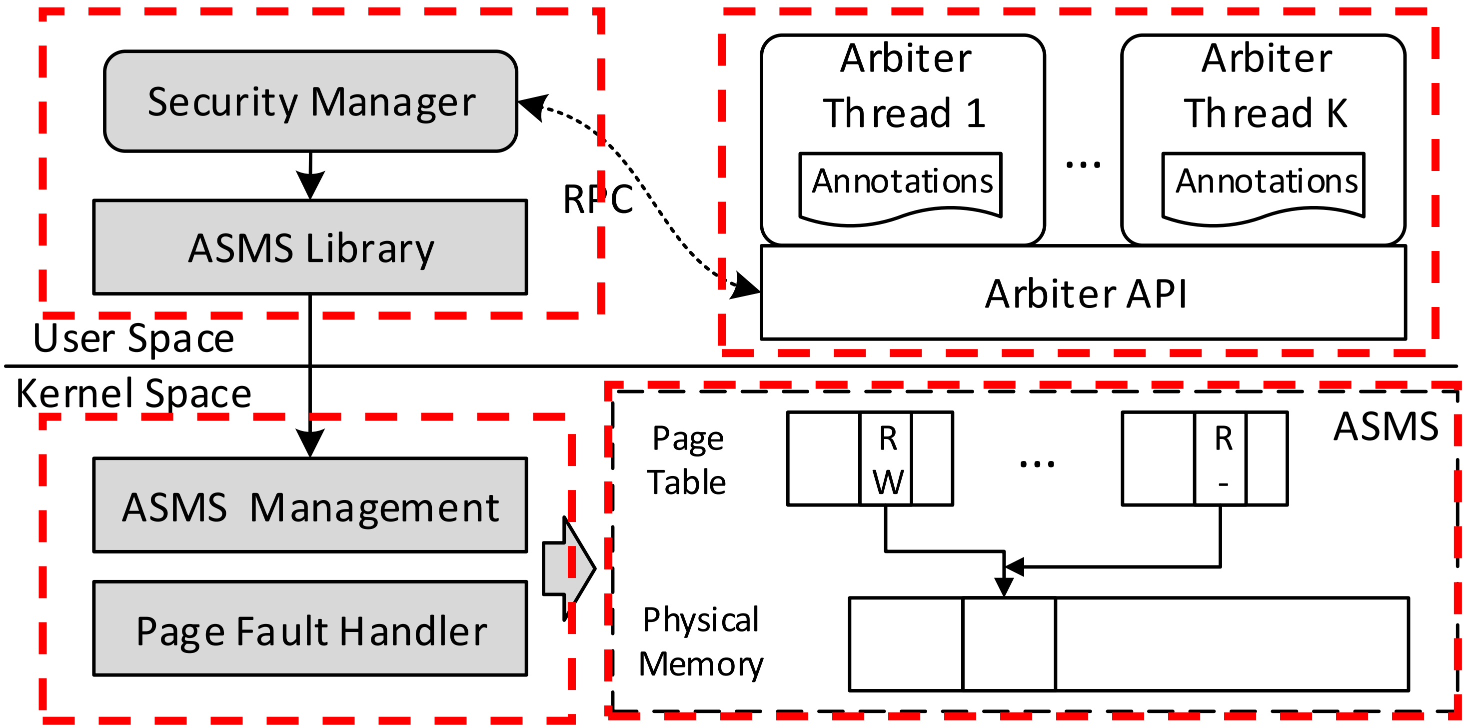
Outline

- Motivation
- Challenges and Our Approach
- Design and Implementation
- Evaluation
- Discussion and Limitations
- Conclusion

Design and Implementation

- Arbiter threads
  - Resemble traditional threads in almost every aspect
    - Shared code seg (.text), data seg (.data, .bss), open files
  - A new dynamically allocated memory segment ASMS
- Major system components
  - Kernel memory region management
  - Page fault handling
  - User space memory allocation
  - Label model and APIs

System Architecture



Outline

- Motivation
- Challenges and Our Approach
- Design and Implementation
- Evaluation
- Discussion and Limitations
- Conclusion

Evaluation

- Port three applications
  - Memcached
  - Cherokee
  - FUSE
- Porting effort

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%)

Evaluation

- Protection effectiveness
  - Arbiter can defeat all the simulated attacks and counterattacks.

Application	Simulated Attack	Arbiter Protection
Memcached	Lack of user auth	√
	Buffer overflow	√
Cherokee	Format string	√
	Logic bug	√
FUSE	Logic bug	√
	Code injection	√

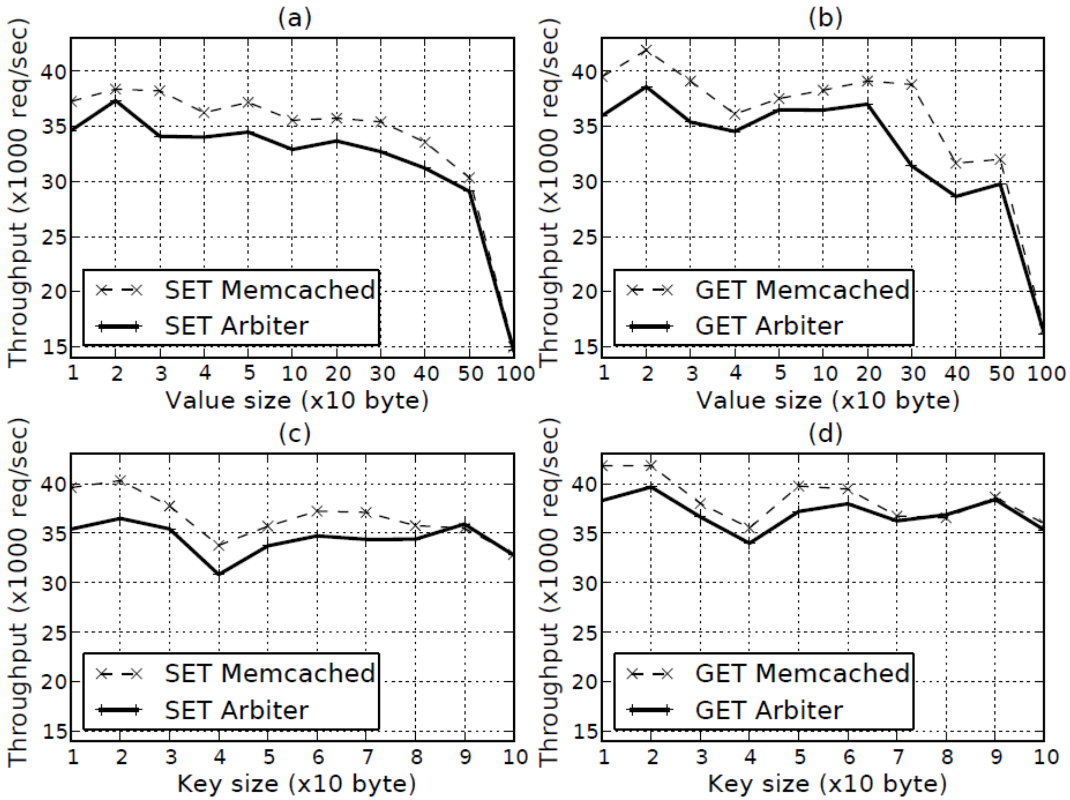
Evaluation

- Performance – microbenchmarks

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

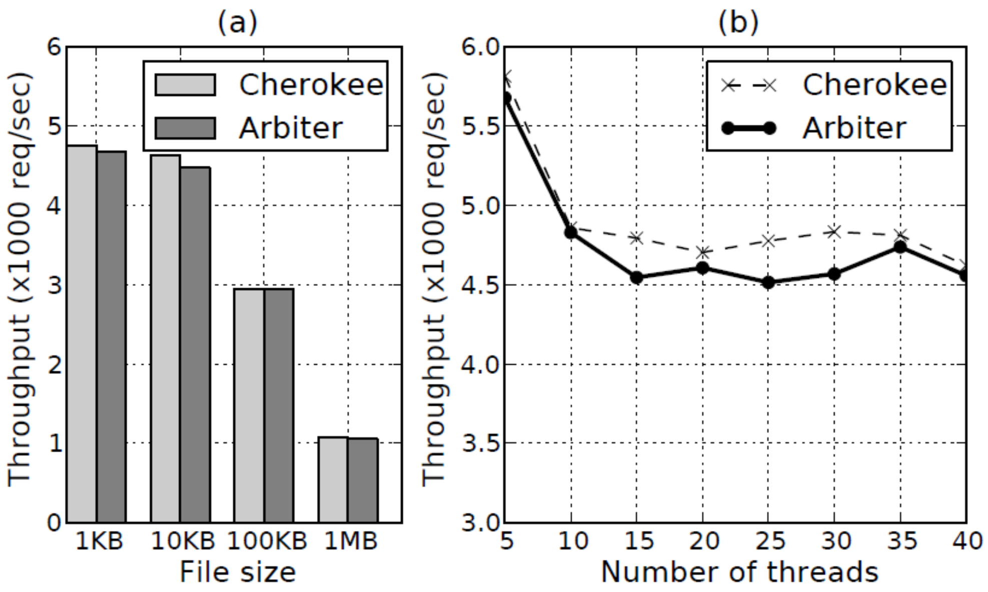
Evaluation

- Application performance – Memcached
  - Average throughput decrease ~5.6%



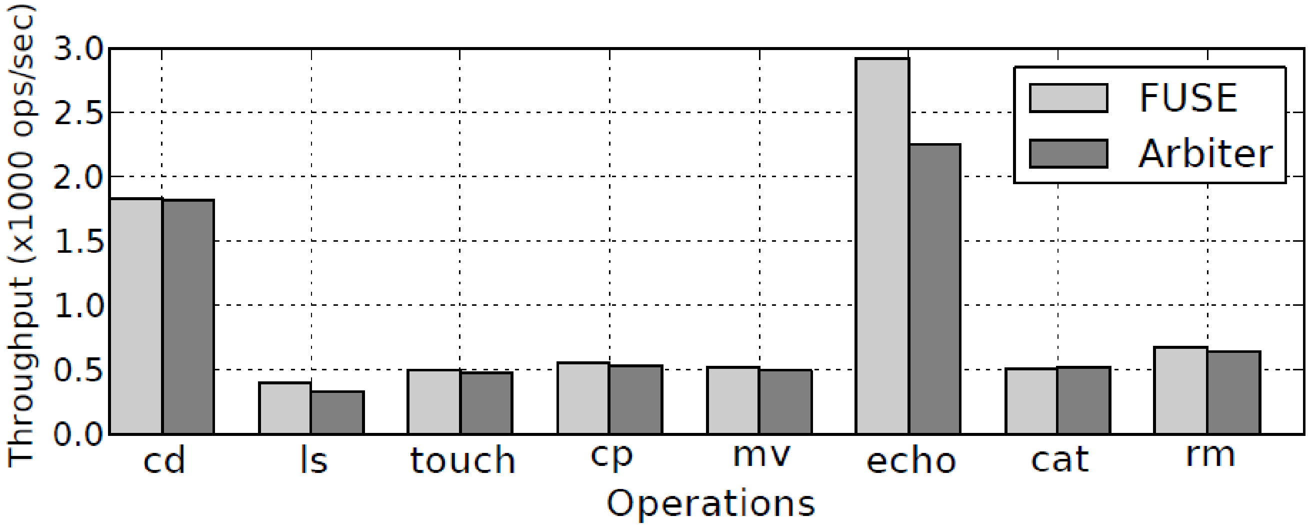
Evaluation

- Application performance – Cherokee
  - Average slowdown ~1.8% (file size), ~3.0% (# threads)



Evaluation

- Application performance – FUSE
  - Average slowdown ~7.4%



Evaluation

- Application performance much better than microbenchmarks
  - Extra cost of Arbiter API is amortized by other operations of the application.
- RSS Memory overhead

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%

Outline

- Motivation
- Challenges and Our Approach
- Design and Implementation
- Evaluation
- Discussion and Limitations
- Conclusion

Discussion and Limitations

- Two users served by the same thread
  - Per-user “virtual” thread?
- Lock granularity of malloc()
  - Potential to adopt per-label lock
- Annotation effort
  - How to ensure policy correctness and avoid misconfiguration?

Conclusion

- Threads not always mutually trusted: needs privilege separation
- Page table protection bits to achieve efficient fine-grained reference monitoring with proper memory management
- Design and implementation of Arbiter system
- Retrofitting and evaluation of three real world applications
- Ease of adoption, effectiveness of protection, and reasonable performance overhead


Thank you