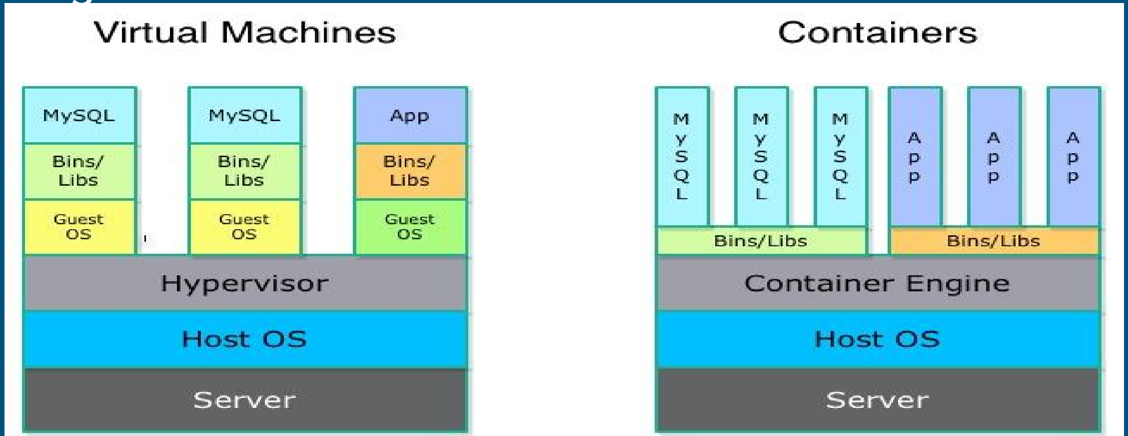
SCONE: Secure Linux Containers with Intel SGX

- Harsh Arya

Containerisation vs VM

- VMs represent hardware -level virtualization, containers represents operating system virtualization . Containers use same kernel and resources are shared among them .
- Containers have minimal OS overhead and are small in size and thus easier to ship and migrate.



Container Isolation

- From Provider's Perspective
  - User's application is not trusted
  - Container should not get access for other containers or privileged environment of the host.
- From Tenant's Perspective
  - Confidentiality and integrity of software and application
    - From other containers
    - From privileged host

Intel SGx - Enclaves

- Trusted execution environment provided with the help of hardware
- Enclave Page Cache(EPC) : (64MB - 128MB)
    - Enclave code and memory in protected physical memory
    - Integrity protected
    - Non-enclave code cannot access it
- Cache guarded by CPU access controls
- Memory Encryption Engine
    - On chip
    - Encrypts and decrypts data of cache line

Intel SGX - Enclave Life Cycle - ring0

- ECREATE
  - Finds a free EPC page and makes it the Enclaves SECS
  - Stores enclave initial attributes (mode of operation, debug, etc.)
- EADD
  - Commits information (REG) or TCS as a new enclave EPC entry pages (4KB at a time).
  - Ensures security properties
    - Maps page at accessed virtual address
- EINIT
  - ensures only measured code has enclave access
  - Creates a cryptographic measurment

Intel SGX - Enclave Life Cycle - ring3

- EENTER
  - Gets enclave TCS address as parameter
  - Verifies validity of enclave entry point
  - Check that TCS is not busy and marked it as “BUSY”.
  - Change CPU mode of operation to “enclave mode”.
- EEXIT
  - TCS is marked as “FREE”
  - Jumps out of enclave flow back to OS instruction address
  - Enclave developer is responsible for clearing registers state.

Linux Containers

- Kernel features are used for isolating applications
- Namespaces
  - PID — Process IDs
  - Network — Network devices, stacks, ports, etc.
  - Mount — Mount points
  - UTS — Hostname and NIS domain name
  - IPC — System V IPC, POSIX message queues
  - User — User and group IDs
  - Cgroup — Cgroup root directory

Linux Containers

- Cgroups
  - Resources like memory, CPU and IO
  - Resource limiting - limits resources
  - Prioritization - prioritizes requests
  - Accounting - accounts usage
  - Control
- Overlay FS(in xenial+) or aufs (in trusty)
  - Layered File system
  - Layers can be shared across the system

SCONE - Desirable

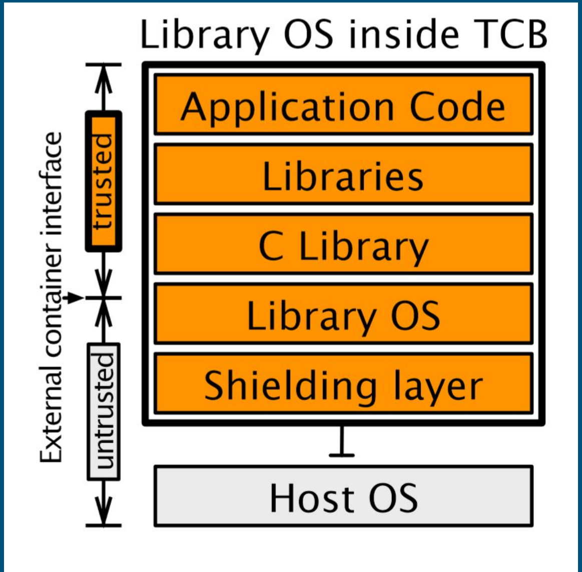
- Small TCB
  - Trusted Code Base
  - Offload system calls
  - Small code will have less attack surface
- Low Overhead
  - Reduce costly enclave transitions
- Transparent to Docker
  - Behave like regular containers

Threat Model

- Powerful and active adversary
  - Superuser access
- Compromise Data integrity or confidentiality by trusting OS
- Programming bugs or inadvertent design bugs of application are not included
- CPU stack is intact
- No denial-of-service or side-channel attacks

Design TradeOffs-Interface Design

- Haven - placing entire library OS inside the enclave
- Large TCB - more vulnerable
- Small Interface - 22 system calls
- Shields protect the interface
- Performance Overhead - extra abstractions



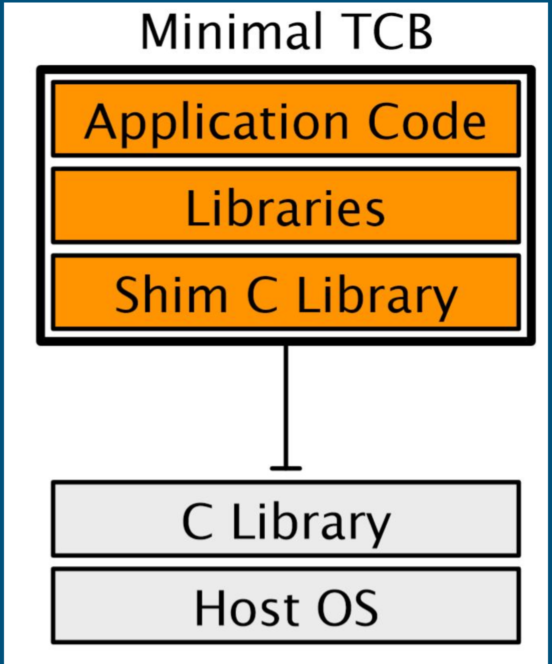
Design TradeOffs-Interface Design

- Similar design - 28 external calls
- Deployed with Linux Kernel library, musl libc library
  - Nginx , Redis - less system calls
  - Sqlite - More system calls , High IO

Service	TCB size	No. host system calls	Avg. throughput	Latency	CPU utilization
Redis	6.9×	<0.1×	0.6×	2.6×	1.1×
NGINX	5.7×	0.3×	0.8×	4.5×	1.5×
SQLite	3.8×	3.1×	0.3×	4.2×	1.1×

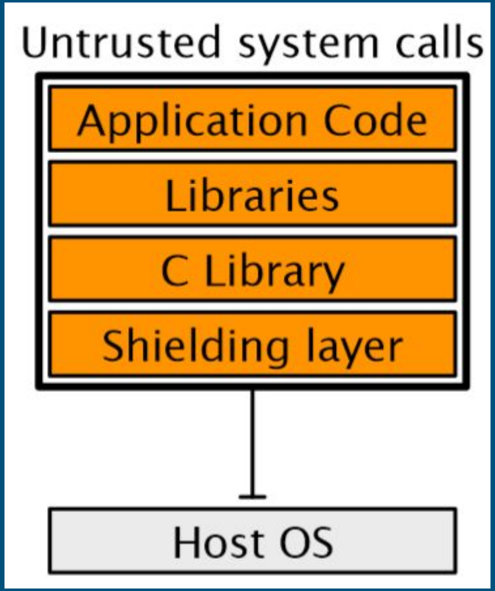
Design TradeOffs -Interface Design

- All calls via external interface
- Small TCB
  - Shim library for relaying libc calls
- Complex C library interface
  - Challenge to protect confidentiality and integrity



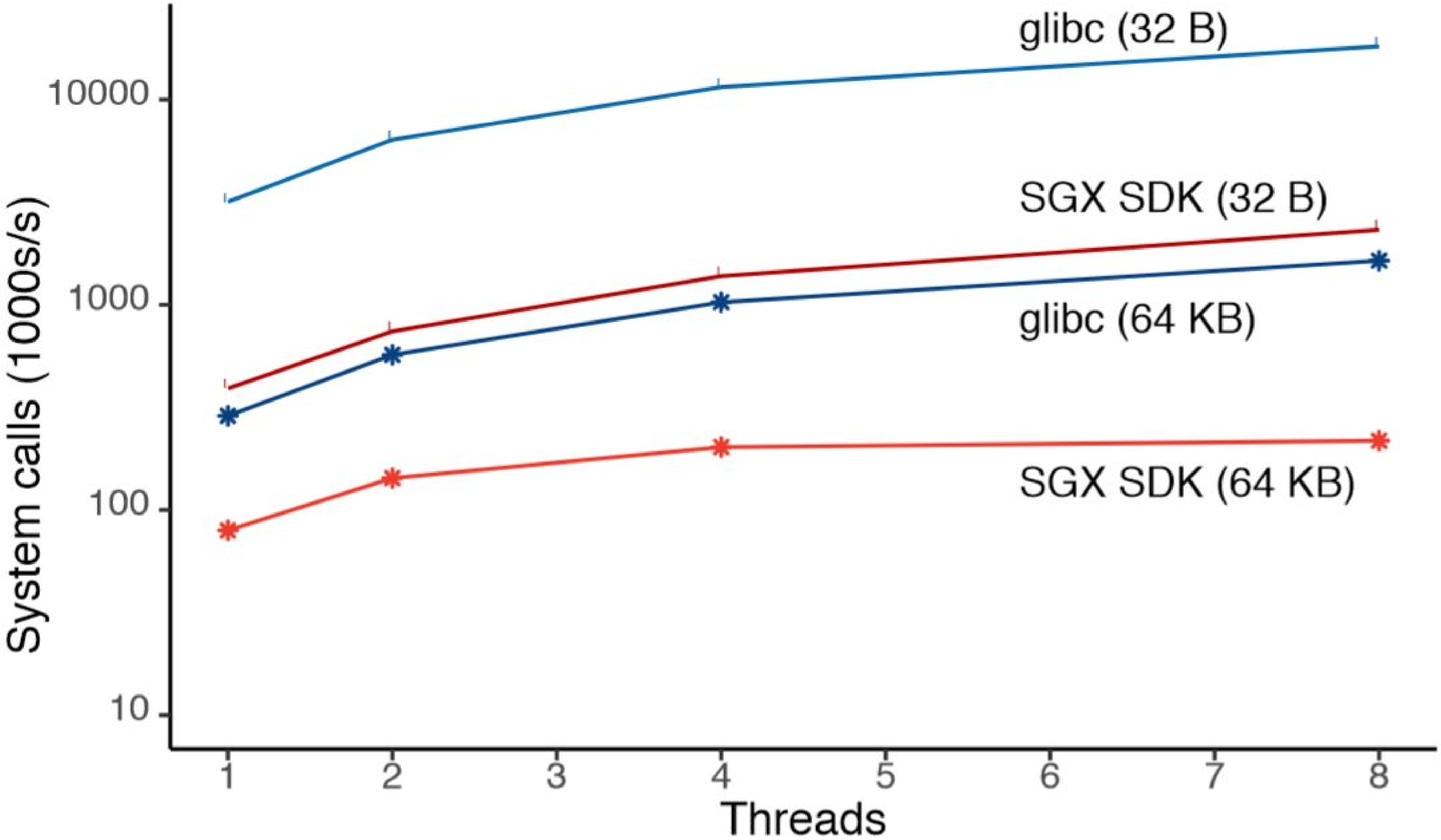
Design TradeOffs -Interface Design

- External interface at the level of system calls
- System calls - already privileged
- Shield libraries to protect security sensitive system calls
    - Read, write, send, recv
- Fork, exec , clone - are not supported
    - Enclave memory is tied to specific process
    - Allocation, attestation and initialisation of independent enclave memory is costly

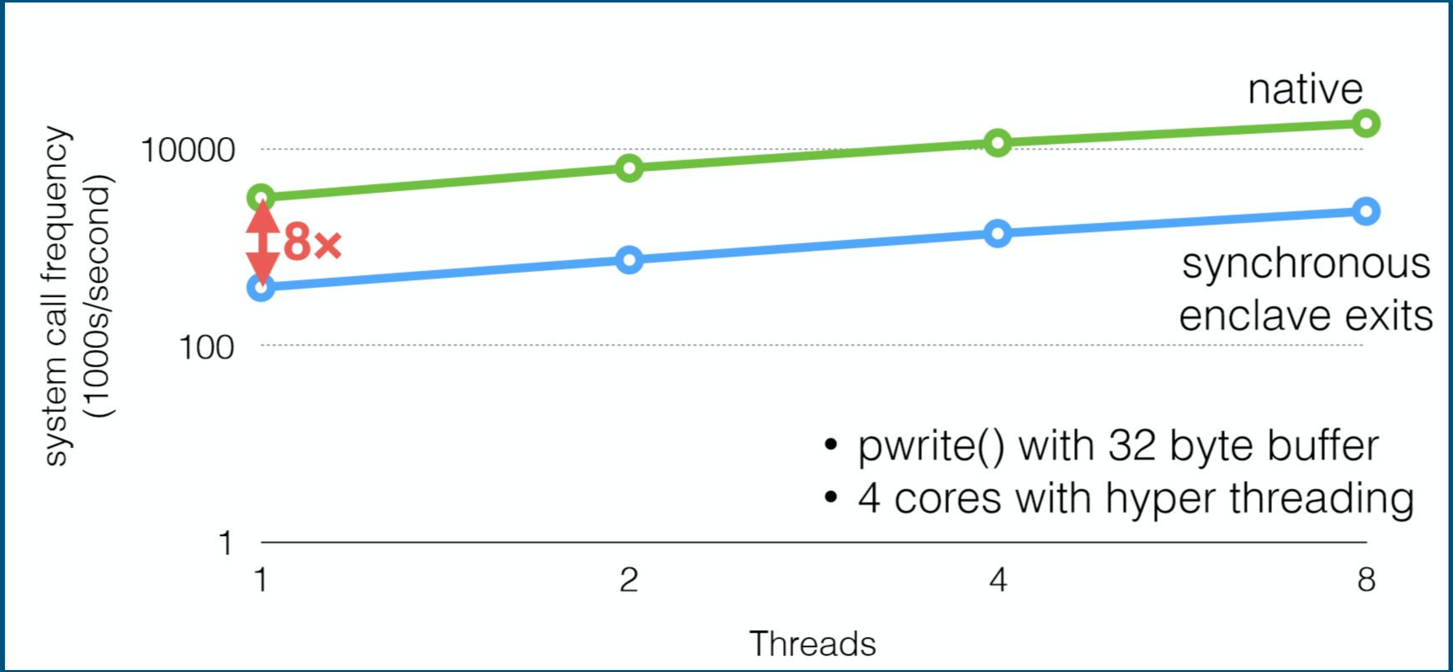


Design TradeOffs - System Calls

- Executing system calls outside the enclave
- Overheads
  - Copy overhead of memory based arguments
  - Leaving enclave is costly as it involves saving and restoring enclave execution state
- Benchmark
  - Synchronous system calls leaving the enclave
  - Overhead of order of magnitude


glibc (32 B)
SGX SDK (32 B)
glibc (64 KB)
SGX SDK (64 KB)
10000
1000
100
10
System calls (1000s/s)
1
2
3
4
5
6
7
8
Threads

Design TradeOffs - System Calls

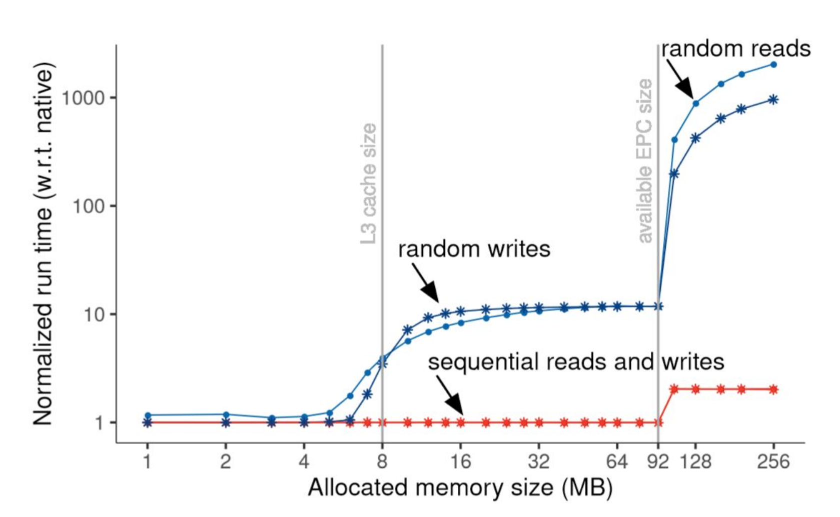


Design TradeOffs - Memory Access

- Overheads of accessing Enclave Pages
  - Penalty for writes to memory and cache misses
    - MEE must encrypt and decrypt cache lines
  - If memory requirements exceeds EPC size, eviction cost
    - Encrypt and integrity protect pages before swapping to DRAM
    - Interrupts all enclave threads
    - Flushes TLB
- Ideal Application
  - Reduce access to enclave memory

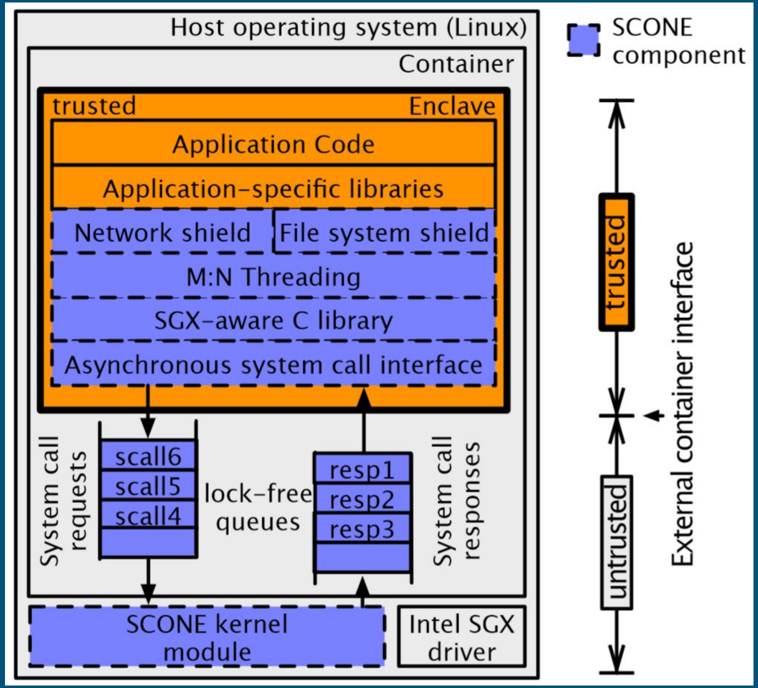
Design TradeOffs - Memory Access

- Micro benchmark - upto 256MB
  - Sequential or random read/write
- 8MB L3 Cache
  - Negligible overheads
- L3 Cache Miss
  - 12x for random operations
  - Negligible for sequential operations
    - Cpu prefetching
- Beyond EPC Size
  - Overhead of 1000x for random
  - Overhead of 2x for sequential



SCONE - Design Architecture

- Enhanced C library → small TCB
  - External calls limited to system calls
- Asynchronous system calls and user space threading reduce number of enclave exits
  - M:N multiplexing threading
  - System calls executed by separate threads
- Network and file system shields actively protect user data
  - Transparent encryption/decryption
- Integration with existing docker
  - compatibility



External Interface Shielding

- Prevent low-level attacks like OS kernel controlling pointer and buffers
- Ensures confidentiality and integrity of the application data
- Supports
    - Transparent Encryption of files
    - Transparent Encryption of Communication Channels
    - Transparent Encryption of Console Streams
- Can associate file descriptor with shield while opening it
- May maintain integrity of file metadata

File System Shield

- Three disjoint set of file path prefixes
    - Protected files - encrypted
    - Encrypted and authenticated
    - Authenticated
- When a file is opened, it matches longest prefix
- Files divided into fixed blocks
    - Authentication tag and none in metadata
    - Metadata also authenticated
    - Keys are part of configuration parameter passed at startup

File System Shield : Ephemeral FS

- Read only container image and thin ephemeral writable layer
- Docker tmpfs - a costly interaction with kernel and file system
- Ephemeral filesystem
  - Maintains state of modified files in non-enclave memory
  - Shield maintain integrity and confidentiality
  - Performance better than tmpfs
  - Resilient to rollback attack
    - No intermediate state is exposed
    - FS returns to previously validated state

Network Shield

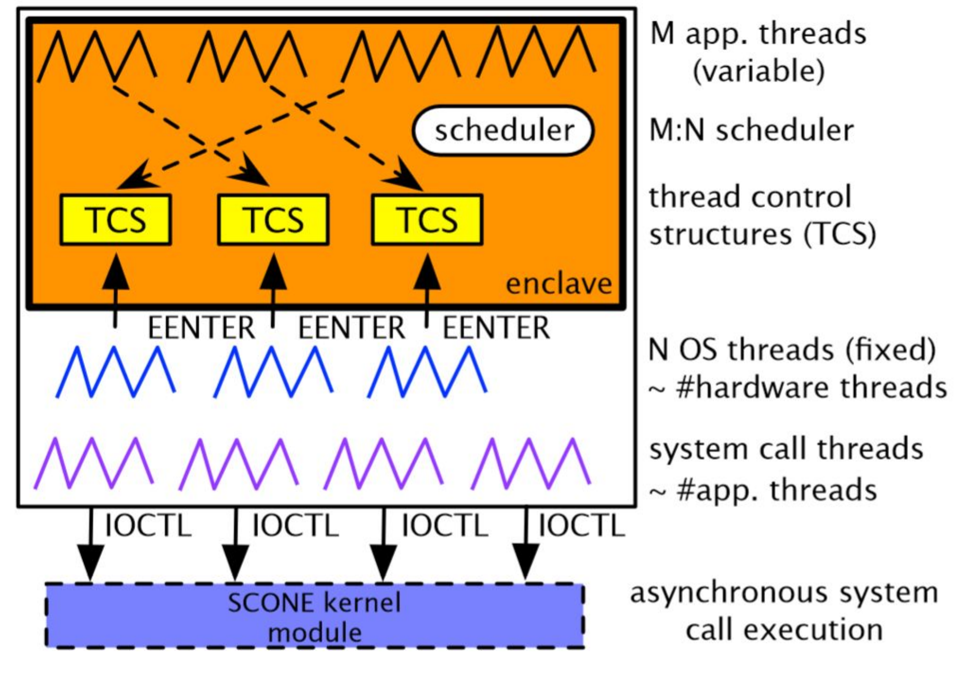
- Establish secure tunnels to container service using TLS
- On new Connection
  - Performs TLS Handshake
  - Encrypts outgoing traffic and decrypts incoming traffic
- Can be activated without server/client side modification
- Private key and certificate available in container FS
  - Guarded by file system shield
- Bidirectional Traffic

Console shield

- Supports transparent encryption of stdout, stdin, stderr streams
- Uses symmetric encryption key between scone client and container exchanged at runtime
- Unidirectional Traffic
- Stream splitted into variable sized blocks
- Resilient to replay and reordering attack
  - Each block has unique identifier checked by SCONE Client

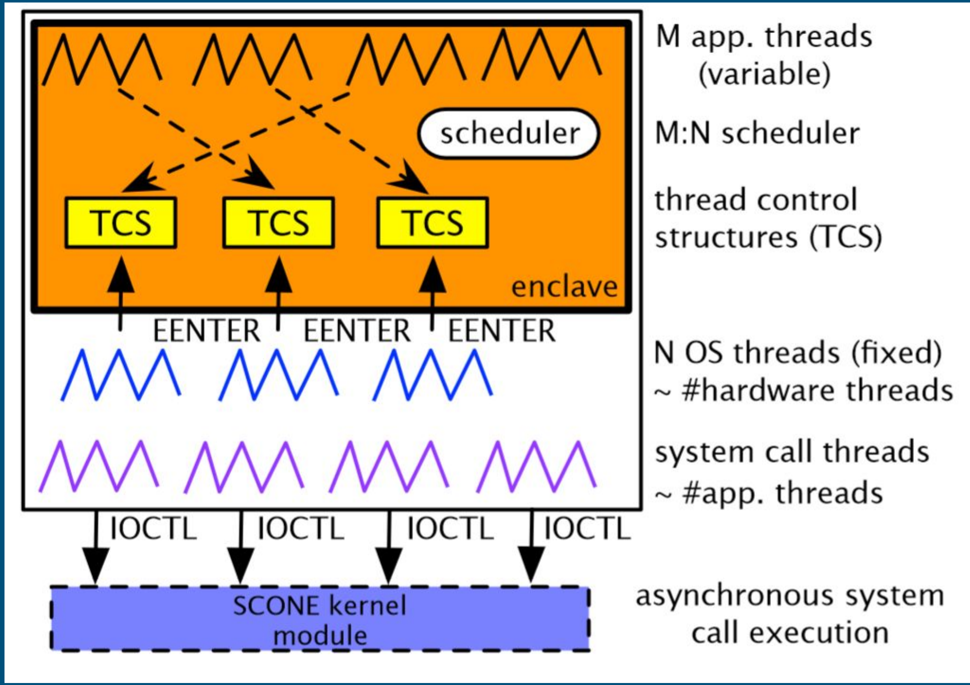
Threading Model

- M:N threading model
  - M application threads multiplexed with N OS threads
- OS threads enter enclave
  - Execute scheduler for checking
    - Application thread need to be woken
    - Application thread for scheduling
  - Scheduler executes the application thread
  - If no application thread, exp backoff
- Number of OS thread in enclave is bound by CPU cores



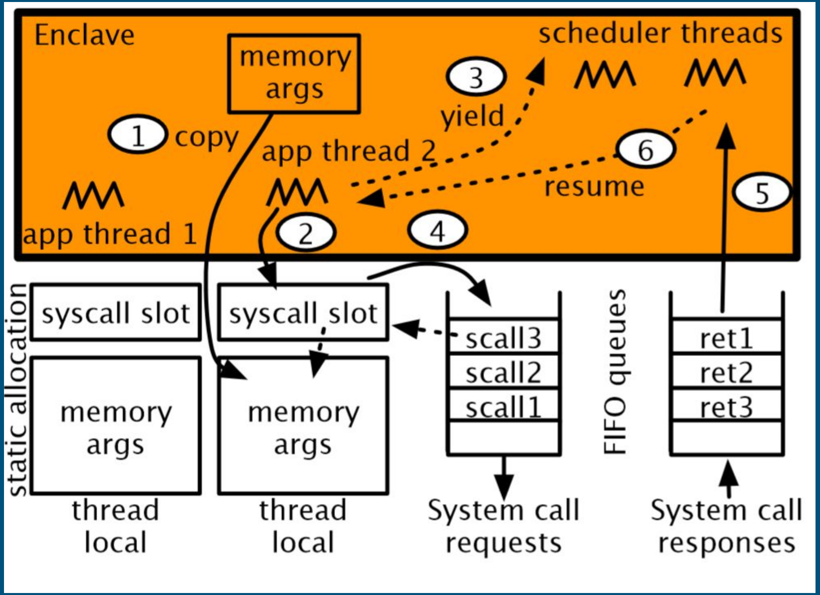
Threading Model

- No preemption
  - Application thread yield on system calls or synchronization primitives
- System call threads reside indefinitely in SCONE kernel to prevent switching overheads
- No. of syscall threads should be higher than application threads
- Periodically, threads left kernel for linux housekeeping



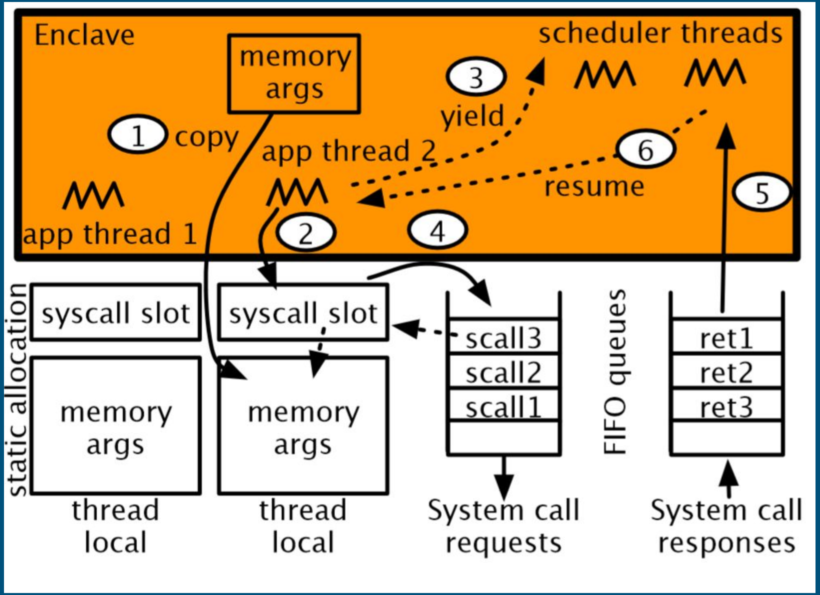
Anatomy of Asynchronous System Calls

1. Memory-based arguments are copied outside the enclave
2. Adds description of system call to syscall_slot data structure
   a. Syscall_slot and memory args reside in thread local memory and are reused
3. Application Thread yields to the scheduler
   a. Scheduler executes other application until response comes
4. Issues syscall by passing reference of syscall_slot to the request queue



Anatomy of Asynchronous System Calls

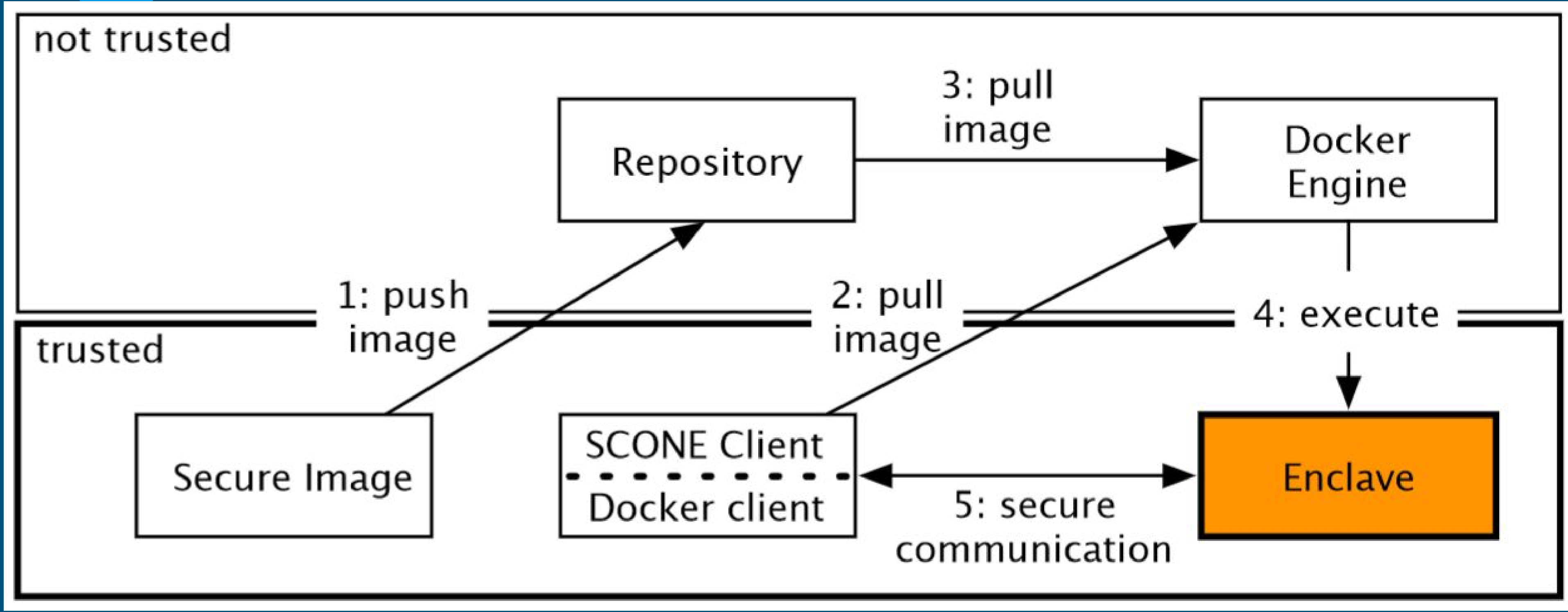
4. OS threads in scone kernel executes syscall and place response in response queue
5. Buffers are copied to inside of enclave and pointers are updated to enclave memory
   a. Checks on buffer size
   b. Checks for no malicious pointers outside enclave memory reach to the application
   c. No pointers passed by OS points to enclave memory - Iago Attack
6. Scheduler resumes the operation of application thread



Docker Integration

- SCONE integrated with docker
- SCONE containers have only single linux process protected by enclave
    - Docker containers can have many processes
- Trusted docker image
- SCONE Client (patched docker client)
    - Create configuration files
    - Launch containers in untrusted environment

Docker Integration



Docker Integration : Image Creation

- Image Creator need to create secured docker image catered to the needs of SCONE

1. Build SCONE executable of the application
    a. Statically compile application with its library dependencies and scone library
2. Use SCONE client to create required metadata
    a. Encrypts specified files
    b. Creates file system protection file (MACs for file chunks and keys)
    c. Encrypts FS protection file and adds to the image
3. Publish secured image to docker repository

Docker Integration : Container Startup

- Startup Configuration File
  - Keys to encrypt standard I/O Streams
  - Hash and encryption key of FS protection File
  - Application arguments
  - Environment Variables
- Verified enclave could only access the SCF
  - Not enforced by SGX
  - Sends it by TLS protected network connection established during startup
  - Container owner first validate proper setup of enclave and then send it to container

Evaluation - Benchmarks

- Work System
  - Intel Xeon E3-1270 v5 CPU with 4 cores at 3.6GHz and 8 hyper-threads (2 per core) and 8MB cache
  - 64 GB Memory
  - Ubuntu 14.04 linux kernel - 4.2
  - Disabled dynamic frequency scaling
- Workload Generator
  - Two 14-core Intel Xeon E5-2683 v3 CPUs at 2GHz with 112GB of RAM and Ubuntu 15.10
  - 10Gb Ethernet NIC with dedicated switch

Evaluation - Benchmarks

- Glibc
- Glibc + Stunnel
  - Encrypt communication for applications like memcached,redis in glibc variant
- SCONE-sync
  - No dedicated system call threads
  - Enclave threads synchronously exit enclave to perform syscall
- SCONE-async
  - Uses scone kernel module to capture syscall in threads

Evaluation - Benchmarks

- Worker Threads
  - Created by application using pthread_create()
  - Glibc - real OS threads
  - SCONE - user space threads
- Enclave threads
  - OS threads permanently in enclave
- System Call threads
  - OS threads permanently outside

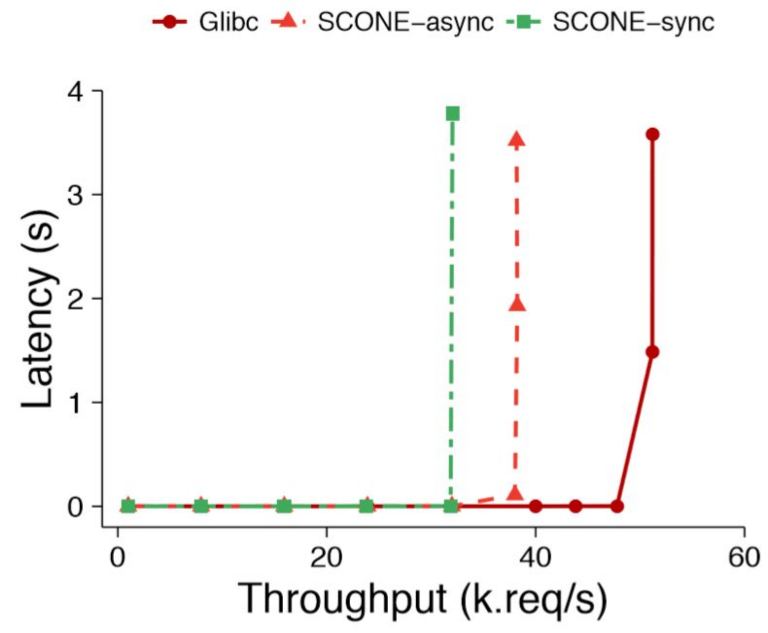
Application	Worker threads		Enclave threads		Syscall threads	
	async	sync	async	sync	async	sync
Apache	25	25	4	8	32	-
NGINX	1	1	1	1	16	-
Redis	1	1	1	1	16	-
Memcached	4	8	4	8	32	-

Evaluation - Benchmarks

- Apache
  - No stunnel is used
- Redis
  - Deployed solely in memory
  - Forking not supported by enclave - Single application thread
- Memcached
  - Application fits in memory
  - Multiple application threads
- Nginx
  - Single worker process

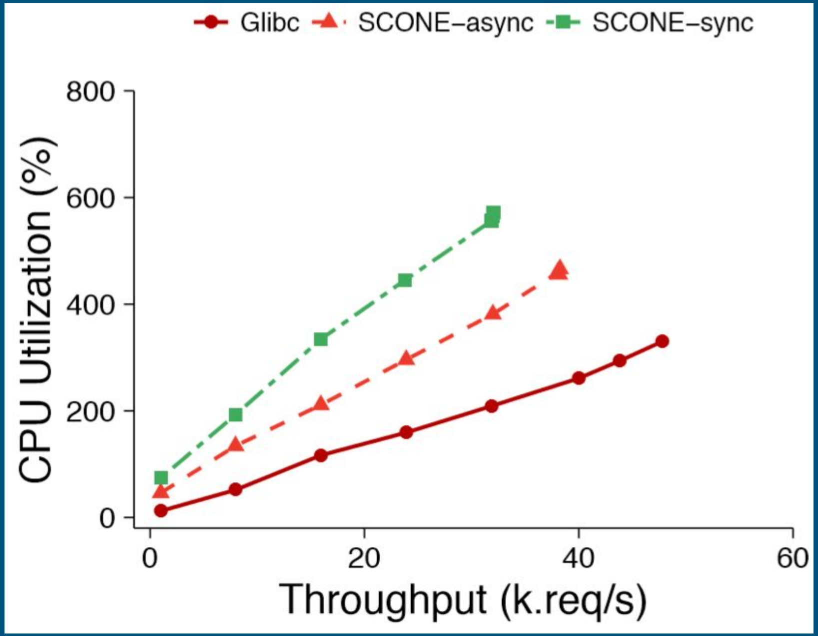
Evaluation - Apache

- SCONE -sync
  - 32,000 requests per second
- SCONE -async
  - 38,000 requests per second
- Glibc
  - 48,000 requests per second



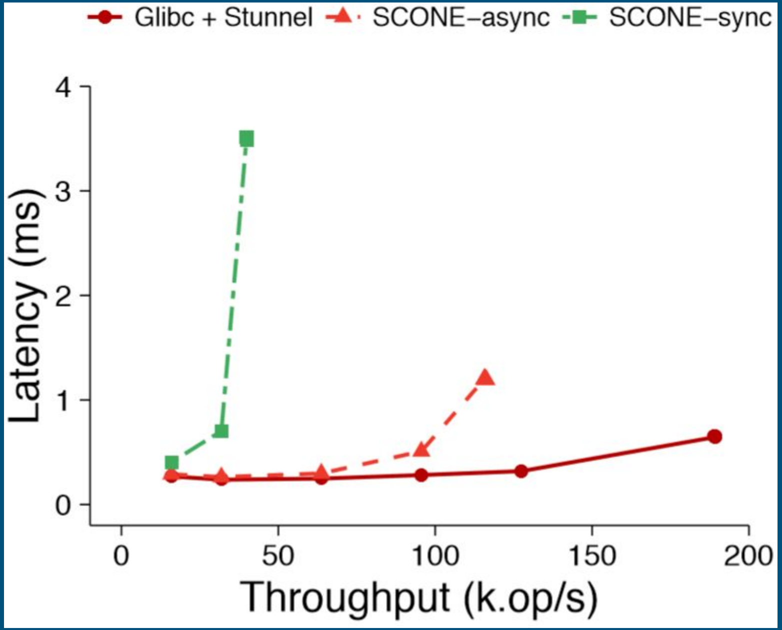
Evaluation - Apache

- Scone -sync
  - Synchronous call interface has less performance
  - More CPU despite async has extra threads
- Scone-async
  - Higher utilisation
    - Slower execution in apache inside enclave
    - Extra threads



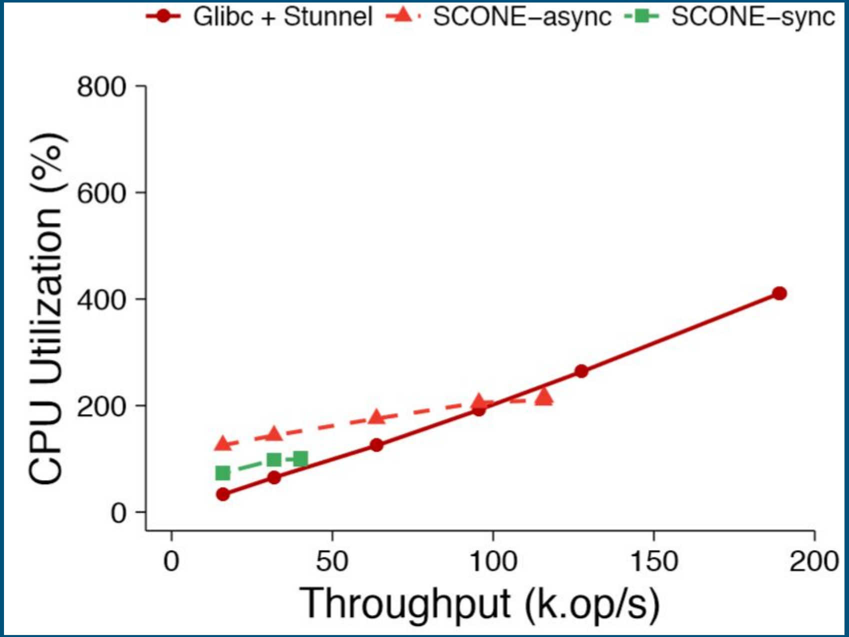
Evaluation - Redis

- Glibc
  - 189,000 requests per second
- SCONE-async
  - 116,000 request per second
  - 61% of glibc
- SCONE - sync
  - 40,000 request per second
  - 21% of glibc



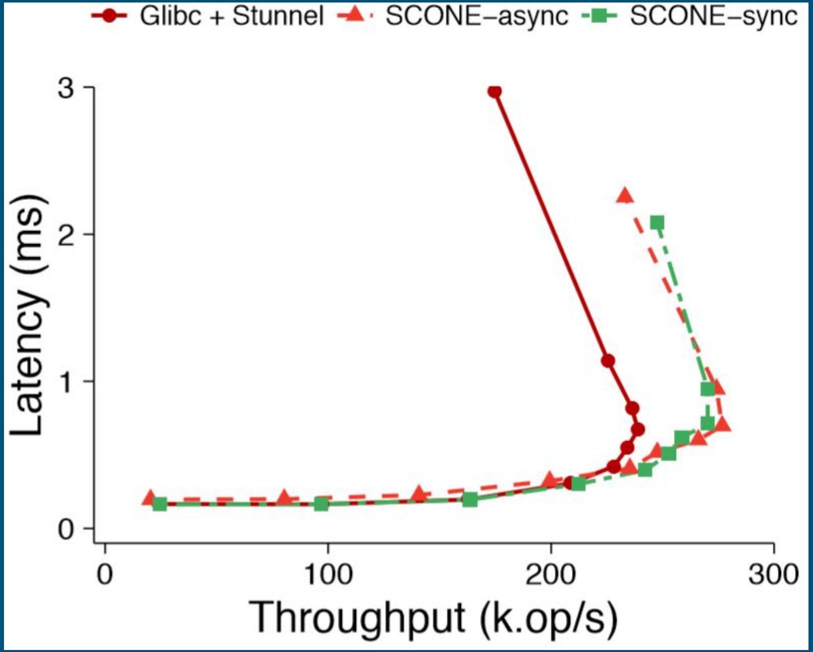
Evaluation - Redis

- Glibc
  - Bounded by 400% CPU utilisation
  - 1 hyperthread - redis
  - 3 hyper-threads - stunnel
- SCONE-sync
  - Perform encryption as part of network shield
  - Cannot use hyper-threading
  - Bounded by 100% CPU
- SCONE-async
  - Limited by single redis application thread
  - Higher than sync due to separate thread for syscall
  - Less than glibc as no separate thread for TLS



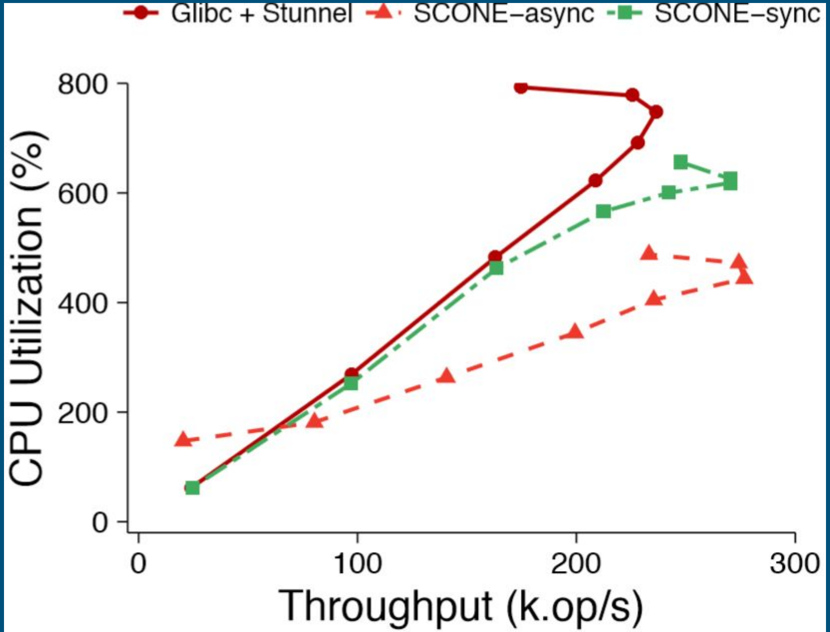
Evaluation - Memcached

- Glibc
  - 230,000 rps
- SCONE-async
  - 277,000 rps
- SCONE-sync
  - 270,000 rps



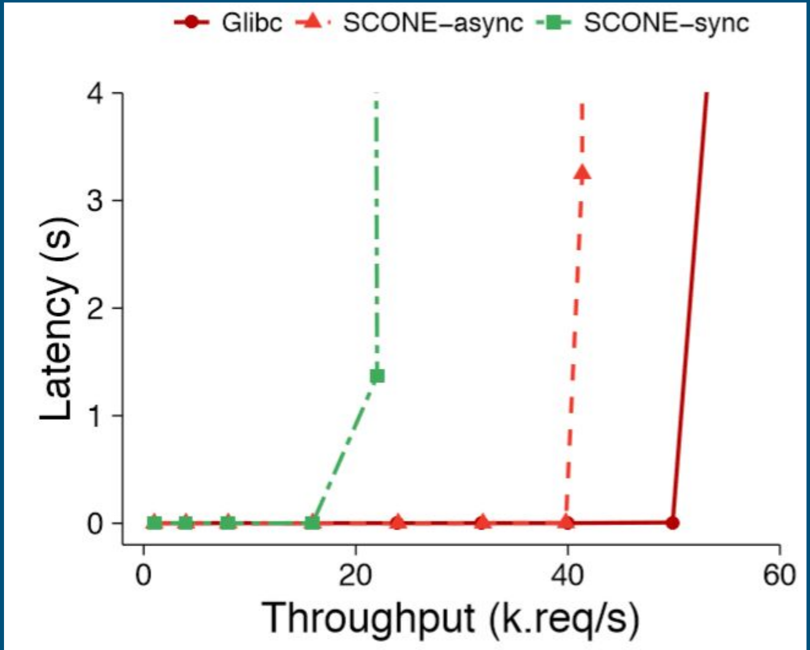
Evaluation - Memcached

- GLibc
  - Memcached competes with stunnel for CPU cycles
- SCONE
  - Network shield is more efficient



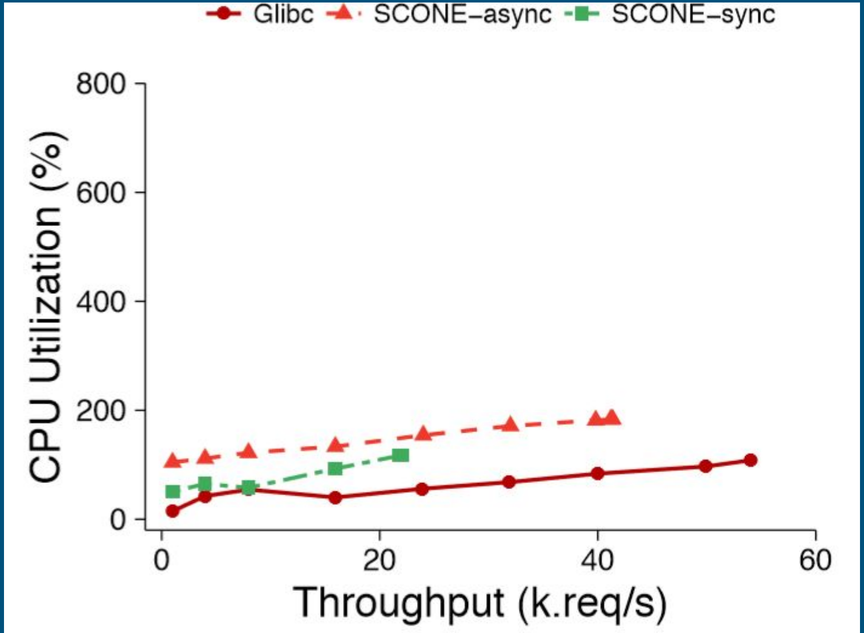
Evaluation - Nginx

- Glibc
  - 50,000 rps
- SCONE-sync
  - 18,000 rps
- SCONE - async
  - 40,000 rps



Evaluation - Nginx

- Lower CPU utilisation than Apache



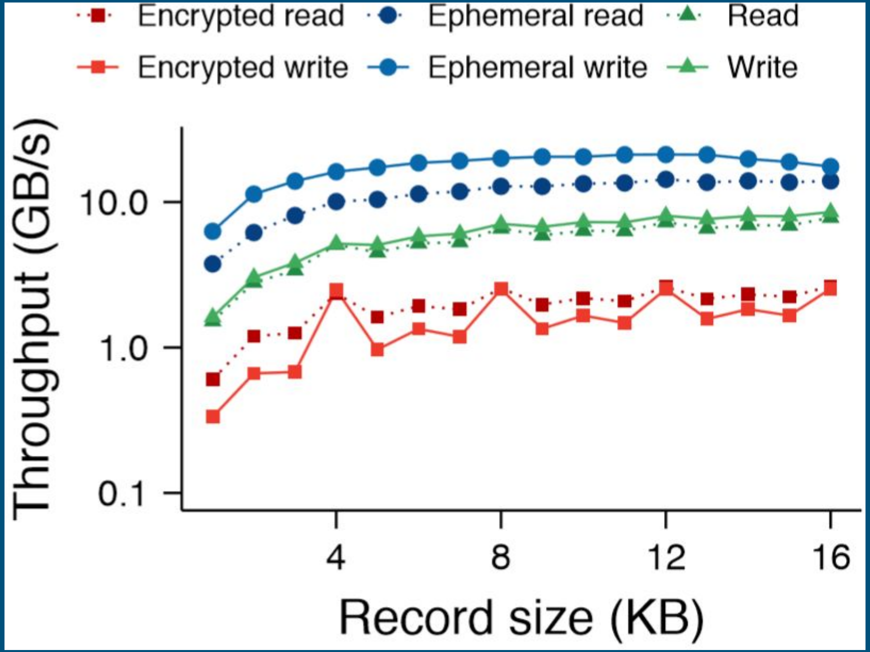
Evaluation - Normalised Application

- Apache
  - SCONE-async as par with native
- SCONE-async faster than SCONE-sync
- For Single threaded applications, benefit from async calls is limited
  - Faster response time compared to sync syscalls
  - No other application thread to run
- Nginx
  - Not scalable as apache
- Codesize - range from 0.6x-2x
  - Musl C library + shield code

Application	SCONE-async	
	T'put	CPU util.
Apache	0.8×	1.4×
Redis	0.6×	1.0×
Memcached	1.2×	0.6×
NGINX	0.8×	1.8×

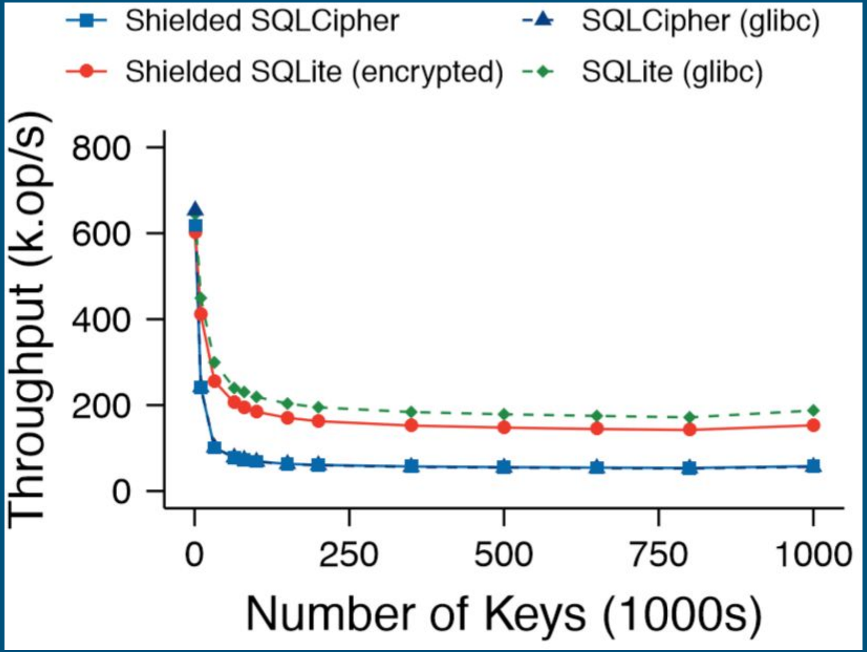
Evaluation - FileSystem Shield

- Ephemeral FS achieved higher performance
  - No syscall while accessing data from ephemeral FS
  - Accesses untrusted memory directly without leaving enclave
- Encryption on ephemeral FS reduces performance



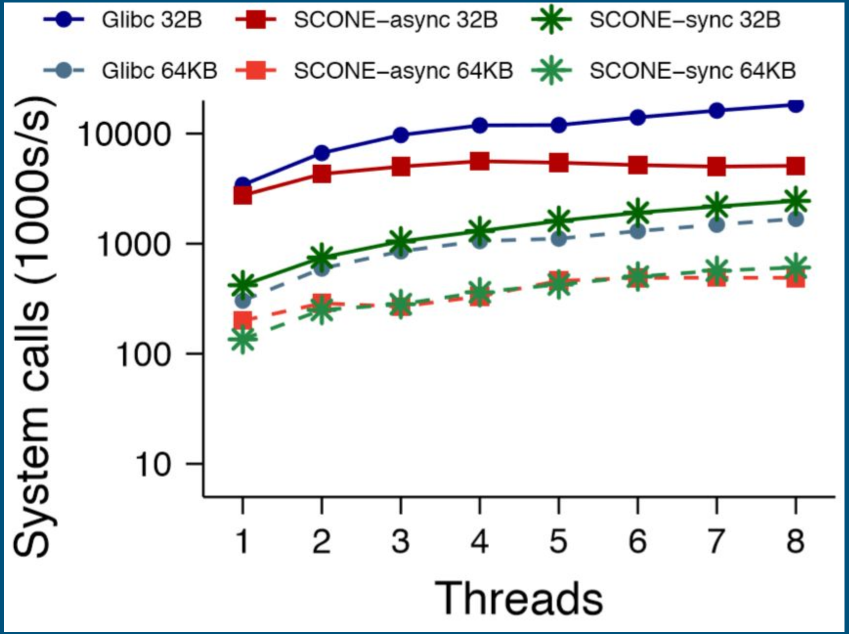
Evaluation - File System Shield

- SQLCipher - application level encryption
- Small Datasets
  - Everything in memory - no encryption/decryption
- SCONE FS shield has higher performance
  - Shield uses AES-GCM encryption faster than AES-CBC of SQLCipher



Evaluation - Async syscalls

- Large Buffers
  - Copy Overheads
  - SCONE-async and SCONE-sync have nearly equal performance
- Small Buffers
  - SCONE-async perform better than SCONE-sync
  - Less than glibc due to stress on shared memory queues



Conclusion

- Security
  - Confidentiality and Integrity of Containers using Intel-SGX
- Codebase Overheads
  - TCB of size = 0.6x-2x of original application code
- Performance Overheads
  - Perform at least 60% of the native
  - For memcached, it perform even faster