CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves

Meni Orenbach, ***Technion;*** **Yan Michalevsky,** ***Anjuna Security;***
Christof Fetzer, ***TU Dresden;*** **Mark Silberstein,** ***Technion***

https://www.usenix.org/conference/atc19/presentation/orenbach

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves

Meni Orenbach
Technion

Yan Michalevsky
Anjuna Security

Christof Fetzer
TU Dresden

Mark Silberstein
Technion

Abstract

Hardware secure enclaves are increasingly used to run complex applications. Unfortunately, existing and emerging enclave architectures do not allow secure and efficient implementation of custom page fault handlers. This limitation impedes in-enclave use of secure memory-mapped files and prevents extensions of the application memory layer commonly used in untrusted systems, such as transparent memory compression or access to remote memory.

CoSMIX is a **Co**mpiler-based system for **S**ecure **M**emory **I**nstrumentation and e**X**ecution of applications in secure enclaves. A novel *memory store* abstraction allows implementation of *application-level secure page fault handlers* that are invoked by a lightweight enclave runtime. The CoSMIX compiler instruments the application memory accesses to use one or more memory stores, guided by a global instrumentation policy or code annotations without changing application code.

The CoSMIX prototype runs on Intel SGX and is compatible with popular SGX execution environments, including SCONE and Graphene. Our evaluation of several production applications shows how CoSMIX improves their security and performance by recompiling them with appropriate memory stores. For example, unmodified Redis and Memcached key-value stores achieve about 2× speedup by using a self-paging memory store while working on datasets up to 6× larger than the enclave's secure memory. Similarly, annotating a single line of code in a biometric verification server changes it to store its sensitive data in Oblivious RAM and makes it resilient against SGX side-channel attacks.

1 Introduction

Virtual Memory is integral to modern processor architectures. In addition to its primary role in physical memory management, it empowers developers to *extend* the standard memory layer with custom data storage mechanisms in software. For example, the memory-mapped file abstraction, which is broadly used, e.g., in databases [10, 5], relies on the OS's page fault handler to map a frame and populate it with the contents of a file. Replacing accesses to physical memory with file accesses requires no application code changes. Therefore, the ability to override page fault behavior has been essential for implementing a range of system services, such as memory compression [44], disaggregation [39, 75], distributed shared memory [36, 46] and heterogeneous memory support [37].

With the emergence of Software Guard Extensions (SGX) for Trusted Execution in Intel CPUs [16, 55], applications are increasingly ported to be entirely executed in *hardware-enforced enclaves* [58, 45, 23, 25]. The enclave hardware protects them from attacks by a powerful privileged adversary, such as a malicious OS or a hypervisor. A number of recent systems facilitate the porting to SGX by shielding unmodified applications in an enclave [21, 81, 18]. Unfortunately, these systems do not allow secure overriding of page fault handling in enclave applications. This drawback complicates porting a large class of applications that use memory-mapped files to SGX. Further, it prevents SGX applications from using security and performance enhancements, such as efficient memory paging [61] and Oblivious RAM (ORAM) side-channel protection [67, 11, 88] without intrusive application modifications. Our goal is to eliminate these constraints.

For example, consider the task of running an SQLite database that uses memory-mapped files in the enclave. The database file must be encrypted to ensure data confidentiality. Enabling in-enclave execution of SQLite therefore requires support for encrypted memory-mapped files, which in turn implies that the *page fault handler must be executed securely* as well. Unfortunately, hardware enclaves available today do not support secure page faults. Instead, existing solutions use workarounds, such as eagerly reading and decrypting the whole mapped file region into trusted enclave memory [18]. This solution does not scale to large files and lacks the performance benefits of on-demand data access.

We argue that the problem is rooted in the fundamental limitation of SGX architecture, which does not provide the mechanism to define *secure* page fault handlers. The upcoming SGX-V2 [54, 86, 43] will not solve this problem either. Moreover, we observe that existing and emerging secure enclave architectures [28, 4, 34] suffer from similar limitations(§2).

In this work, we build **CoSMIX**, a compiler and a lightweight enclave runtime that overcomes the SGX architectural limitations and enables secure and efficient extensions to the memory layer of *unmodified* applications running in enclaves. We introduce a *memory store, (*m*store)*, a programming abstraction for implementing custom memory management extensions for enclaves. The CoSMIX compiler automatically instruments application code to allocate the selected

variables and memory buffers in the *m*store, replacing the accesses to enclave memory with the accesses to the *m*store. The *m*store logic runs in the enclave as part of the application. The CoSMIX runtime securely invokes the *m*store memory management callbacks, which include custom page fault handlers. The page faults are semantically equivalent to hardware page faults yet are triggered by the CoSMIX runtime.

An *m*store can implement the missing functionalities that require secure page fault handlers. For example, it may provide the secure `mmap` functionality by implementing the page fault handler that accesses the file and decrypts it into the application buffer. A more advanced *m*store may add its own in-memory cache analogous to an OS page cache, to avoid costly accesses to the underlying storage layer. CoSMIX supports several types of *m*stores, adjusting the runtime to handle different *m*store behaviors while optimizing the performance.

CoSMIX allows the use of *multiple* m*stores* in the same program. This can be used, for example, to leverage both secure `mmap` *m*store and an ORAM *m*store for side-channel protection. Additionally, CoSMIX supports *stacking* of multiple *m*stores to enable their efficient composition and reuse. We design and prototype three sophisticated *m*stores in §3.2.5, and demonstrate the benefits of stacking in §4.5.

CoSMIX's design focuses on two primary goals: (1) minimizing the application modifications to use *m*stores and (2) reducing the instrumentation performance overheads. We introduce the following mechanisms to achieve them:

Automatic inference of pointer types. CoSMIX *does not* require annotating every access to a pointer. Instead, it uses inter-procedural pointer analysis [17] to determine the type of the *m*store (or plain memory) to use for each pointer. When the static analysis is inconclusive, CoSMIX uses *tagged pointers* [47, 74, 13] with the *m*store type encoded in the unused Most Significant Bits, enabling runtime detection (§3.3.1).

Locality-optimized translation caching. The *m*store callbacks interpose on memory accesses, which are part of the performance-critical path of the application. To reduce the associated overheads, we employ static compiler optimizations to reduce the number of runtime pointer type checks and *m*store accesses. These include loop transformations and a software *Translation Lookaside Buffer (TLB)* (§3.3.4). These mechanisms reduce the instrumentation overheads by up to two orders of magnitude (§4.2).

Our prototype targets existing SGX hardware and is compatible with several frameworks for running unmodified applications in enclaves [81, 1, 18]. However, CoSMIX makes no assumptions about enclave hardware. The CoSMIX compiler is implemented as an extension of the LLVM framework [48].

We prototype three *m*stores: Secure User Virtual Memory (SUVM) for efficient paging in memory-intensive applications [61], Oblivious RAM [78] for controlled side-channel protection, and a secure `mmap` *m*store that supports access to encrypted/integrity-protected memory-mapped files. We evaluate CoSMIX on the Phoenix benchmark suite [66], as well as on unmodified production servers: `memcached`, Redis, SQLite, and a biometric verification server [61]. The compiler is able to correctly instrument all of these applications, some with hundreds of thousands of lines of code (LOC), *without the need to manually change the application code*.

Our microbenchmarks using Phoenix with SUVM and secure `mmap` *m*stores show that CoSMIX instrumentation results in a low geometric mean overhead of 20%.

For the end-to-end evaluation, we run `memcached` and Redis key value stores on 600 MB datasets – each about 6× the size of the secure physical memory available to SGX enclaves. In this setting, SGX hardware paging significantly affects the performance. The SUVM [61] *m*store aims to optimize exactly this scenario. To use it, we only annotate the item allocator in `memcached` (a single line of code) and compile it with CoSMIX. Redis is compiled *without* adding annotations. The instrumented versions of both servers achieve about 2× speedup compared to their respective vanilla SGX baselines.

In another experiment, we evaluate a biometric verification server with a database storing 256 MB of sensitive data. We use the ORAM *m*store to protect it from SGX controlled side-channel attacks [87] that may leak sensitive database access statistics. We annotate the buffers containing this database (one line of code) to use ORAM. The resulting ORAM-enhanced application provides security guarantees similar to other ORAM systems for SGX, such as ZeroTrace [67], yet without modifying the application source code. ORAM systems are known to result in dramatic performance penalties of several orders of magnitude [26]. However, our hardened application is only 5.8× slower than the vanilla SGX thanks to the benefits of selective instrumentation enabled by CoSMIX.

To summarize, our contributions are as follows:

- Design of a compiler and an *m*store abstraction for transparent secure memory instrumentation (§3.2).
- Loop transformation and loop-optimized caching techniques to reduce the instrumentation overheads (§3.3.4).
- Seamless security and performance enhancement for unmodified real-world applications running in SGX, by enhancing them with the SUVM, ORAM and secure `mmap` *m*stores (§4).

2 Motivation

Enabling the use of custom page fault (PF) handlers in enclaves would not only facilitate porting of existing applications that rely on such functionality, but also enable a range of unique application scenarios, as we discuss next.

SUVM. The authors of Eleos [61] proposed Software User-space Virtual Memory (SUVM), which implements *exit-less memory paging in enclaves* and significantly improves the performance of memory-demanding secure applications. It keeps the page cache in the enclave's trusted memory, while the storage layer resides in untrusted memory whose contents are encrypted and integrity-protected.

I like their writing style. It's clear and well cited.

ORAM. Oblivious RAM (ORAM) obfuscates memory access patterns by shuffling the physical data locations and re-encrypting the contents upon every access. As a result, an adversary observing the accessed locations learns nothing about the actual access pattern to the data [38]. Multiple ORAM schemes have been proposed over time [38, 84, 78, 88, 67, 33], and, ORAM was recently used to manually secure applications executing in SGX enclaves against certain side-channel attacks [88, 67, 33].

Both ORAM and SUVM are generic mechanisms that could be useful in many applications. Unfortunately, integrating them with the application logic requires intrusive code modifications. With the support for efficient and secure in-enclave PF handlers, we could add these mechanisms to existing unmodified programs, as we show in the current work.

Other applications include transparent compression for in-enclave memory, `mmap` support for encrypted and integrity-protected files, and inter-enclave shared memory, as well as various memory-management mechanisms [39, 75, 37].

Unfortunately, existing enclave hardware provides no adequate mechanisms to implement efficient and secure user-defined PF handlers, as we describe next.

2.1 Background: page-faults in enclaves

There are several leading enclave architectures: Intel SGX [16, 43], Komodo for ARM Trust Zone [34, 15], Sanctum [28], and Keystone [4]. Among these, only Intel SGX and Sanctum published support for paging. We briefly describe them below.

Intel SGX [16, 43, 55] supports on-demand paging between secure and untrusted memory. SGX relies on Virtual Memory hardware in X86 CPUs. When a PF occurs, the enclave *exits* to an *untrusted* privileged OS which invokes the SGX PF handler. The enclave execution resumes (via `ERESUME`) after the swapping is complete.

Since the PF handler is untrusted, the SGX paging is secured via SGX paging instructions. Specifically, `EWB` encrypts and signs the page when swapping the page out, whereas `ELDU` validates the integrity and decrypts when swapping it in. These instructions cannot be modified to perform other operations. They cannot change the internal SGX encryption key or modify the swapped page. In other words, they cannot act as a general-purpose secure PF handler.

Sanctum [28] supports per-enclave page tables and secure PF handlers. It uses a security monitor that runs at a higher privilege level than the OS and the hypervisor. Upon a PF, the enclave exits to the security monitor, which triggers the in-enclave secure PF handler.

2.2 Limitations of existing enclaves

Signal handling in SGX. Page fault handlers can be customized in userspace by registering a signal handler. SGX supports signal handlers in enclaves and works according to the following scheme (Figure 1): when an interrupt occurs, the enclave exits to the OS ❶. The OS takes control and performs an up-call to an untrusted user-space trampoline in the enclave's hosting process ❷. The trampoline re-enters the enclave by invoking the in-enclave signal handler ❸. After the signal handler terminates, the enclave exits ❹ and resumes execution of the last instruction ❺ via `ERESUME`.

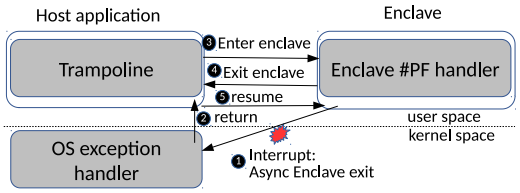


Figure 1: Execution of a signal handler in SGX

SGX: No secure page fault handlers. The SGX signal handling mechanism cannot guarantee secure execution of the handler itself. When the enclave is resumed after the PF, `ERESUME` replays the faulting memory access. Therefore, the enclave *cannot validate* that the signal handler was indeed executed correctly, or was executed at all. To the best of our knowledge, this problem will not be resolved in the next version of SGX [54, 86, 43].

This is a problem for me as well

SGX: Performance overheads. Even with hardware support for the secure signal handler, SGX has inherent architectural properties that will lead to significant performance penalties, rendering this mechanism unsuitable for customized application memory management. The architecture relies on the OS to manage the enclave's virtual memory. Furthermore, SGX may only run userspace code. Since the OS is untrusted, any secure page fault handling would inevitably follow the signal handling scheme depicted in Figure 1, namely, double enclave transition between the trusted and untrusted contexts.

We measure the latency of an empty SGX PF handler (access to a protected page) to be *11*μsec, which is more than 6$\times$ the latency of a signal handler outside SGX. For comparison, CoSMIX's software page fault handler is only 0.01 μ second, the cost of a single function call, which is three orders of magnitude faster than in SGX.

Further analysis shows that the signal latency is dominated by the latency of enclave transitions, which we measure to be 5.3μsec each [1] and stems from costly validation, register scrapping, TLB and L1 cache flushes [82, 43].

Other enclave architectures. Enclave transition overheads are pertinent to other enclave architectures. Komodo reports exit latency of 0.96μs [34]. Sanctum and Keystone do not disclose their enclave transition penalties, yet they describe similar operations performed when such transitions occur.

We conclude that *secure page fault handlers are not supported in SGX, and are likely to incur high-performance costs in other enclave architectures due to transition overheads.*

[1] This value is almost double the one reported in prior works [61, 85] because of the firmware update to mitigate the Foreshadow [82] bug.

2.3 Code instrumentation for enclaves

Instrumenting application memory accesses with the desired software PF handling logic is a viable option to achieve the functionality equivalent to the hardware-triggered PF handlers. Unfortunately, existing instrumentation techniques are not sufficient to achieve our goals, as we discuss below.

Binary instrumentation. Dynamic binary instrumentation tools [51, 57, 24, 52, 71], such as Intel PIN [51], enable instrumentation of memory accesses. Unfortunately, these tools have significant drawbacks in the context of in-enclave execution. For example, for PIN to work, all its libraries should be included in the enclave's Trusted Computing Base (TCB). Moreover, PIN requires JIT-execution mode to instrument memory accesses. Therefore, the enclave code pages should be configured as writable, which might lead to security vulnerabilities. Removing the write access permission from enclave pages will be supported in SGX V2, but doing so will require costly enclave transitions [43].

Static binary instrumentation tools do not suffer from these shortcomings. However, compared to the compiler-based techniques we propose in this work, they do not allow using comprehensive code analysis necessary for performance optimizations. Therefore, we decided against the binary instrumentation design.

Compiler-based instrumentation. The main advantage of this method is the ability to aggressively reduce the instrumentation overheads by using advanced source code analysis. On the other hand, the source code access requirement limits the applicability of this method. However, this drawback is less critical in the case of SGX enclaves because many SGX execution frameworks, such Panoply [77] and SCONE [18], require code recompilation anyway. Therefore, we opt for compiler-based instrumentation in CoSMIX.

3 CoSMIX Design

CoSMIX aims to facilitate the integration of different *m*stores efficiently into SGX applications. Our design goals are:

- **Performance.** Low overhead memory-access and software address translation.
- **Ease-of-use.** Annotation-based or automatic instrumentation without manual application code modification.
- **General memory extension interface.** Easy and modular development of application-specific memory instrumentation libraries.
- **Security.** Keep SGX security guarantees and small TCB.

Threat Model. CoSMIX is designed with the SGX threat model, where the TCB includes the processor package and the enclave's code. Additionally, we assume that the code running in an enclave does not contain memory vulnerabilities.

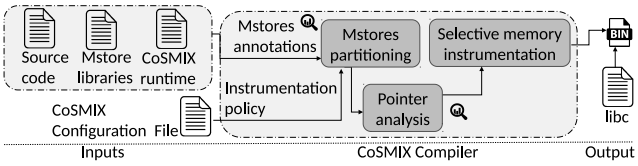


Figure 2: CoSMIX compilation overview. The compiler is guided by code annotations and global instrumentation policy.

3.1 Design overview

Compiler-based instrumentation. CoSMIX enables SGX developers to build custom *memory stores,* m*stores* that redefine the memory layer functionality of an instrumented program. To integrate one or more *m*stores into an application, the CoSMIX compiler automatically instruments the program (Figure 2). The developer may selectively annotate static variables or/and memory allocations to use different *m*stores, or define a global instrumentation policy. The compiler automatically instruments the necessary subset of memory accesses with the accesses to the corresponding *m*stores, and statically links *m*store libraries with the code.

The CoSMIX configuration file defines the instrumentation behavior. It specifies annotation symbols per *m*store, *m*store type (§3.2) and the instrumentation policy(§3.3).

3.2 *M*store abstraction

At a high level, an *m*store implements another layer of virtual memory on top of an abstract *storage layer*. An *m*store operates on pages, m*pages*, and keeps track of the *m*page-to-data mappings in its internal *m*page table. When an application accesses memory, the runtime invokes the *m*store's software *page fault handler*, retrieves the contents (e.g., for the secure `mmap` *m*store, it would read data from a file and decrypt), and makes it accessible to the application.

We distinguish between *cached* and *direct-access* *m*stores. A cached *m*store maintains its own *m*page cache to reduce accesses to the storage layer, whereas a direct-access *m*store does not cache the contents.

Figure 3 shows the execution of an access to a cached *m*store. The pointer access ❶ triggers the runtime call, which chooses the appropriate *m*store ❷ and checks the translation cache ❸. If the translation is not in the cache, the runtime invokes the *m*page fault handler ❹. The *m*store translates the pointer ❺, and either fetches the referenced *m*page from the page cache ❻, or retrieves it from the storage layer and updates the *m*page and translation caches ❼.

3.2.1 *M*store callbacks

Table 1 lists the callback functions *m*stores must implement.
Initialization/teardown. The *m*store is initialized at the beginning of the program execution and torn down when the pro-

Callback	Purpose
`mstore_init(params)/mstore_release()`	Initialize/tear down
`void* alloc(size_t s, void* priv_data)/free(void* ptr)`	Allocate/free buffer
`size_t alloc_size(void* ptr)`	Allocation size
`size_t get_mpage_size()`	Get the size of the *m*page
Direct-access *m*store	
`mpf_handler_d(void* ptr, void* dst, size_t s)`	*m*page fault on access to `ptr`, store the value in `dst`
`write_back(void* ptr, void* src, size_t size)`	Write back value in `src` to `ptr`
Cached *m*store	
`void* mpf_handler_c(void* ptr)`	*m*page fault on access to `ptr`, return pointer to *m*page
`flush(void* ptr, size_t size)`	Write the *m*pages in the range `ptr:ptr+size` to *m*store
`get_mstorage_base()/get_mpage_cache_base()`	Gets the base address of mstorage/*m*page cache
`notify_tlb_cached(void* ptr) / notify_tlb_dropped(void* ptr, bool dirty)`	The runtime cached/dropped the `ptr` translation in its TLB

Table 1: Compulsory *m*store callback functions

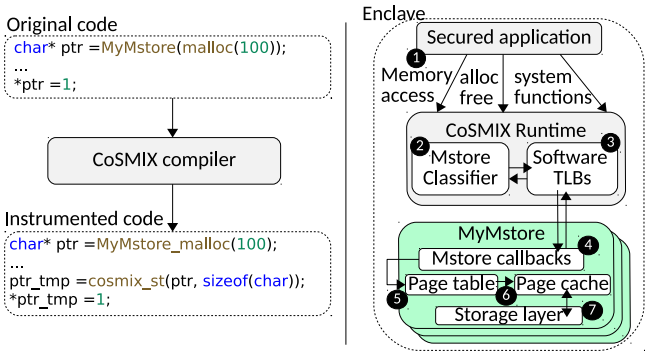


Figure 3: CoSMIX: code transformation and execution flow of access to a cached *m*store. See the text for explanation.

gram terminates. Importantly, the runtime flushes the *m*page cache when tear-down of cached *m*stores is called.

Memory allocation. The runtime delegates the memory allocation calls of the original program to the *m*store `alloc`.

3.2.2 Pointer access and *m*page faults

When the instrumented code accesses the *m*store, the runtime incurs an equivalent of a page fault, and invokes the respective callback in the *m*store, as discussed below.

Cached *m*stores. A cached *m*store *translates* the pointer to the *m*page inside its cache. `mpf_handler_c` returns the pointer into the *m*page that holds the requested data, allowing direct access from the code. For cross-page misaligned accesses, the runtime creates a temporary buffer and populates it by calling the `mpf_handler_c` for every page separately. For store access, the updates are written to both pages.

Impressive, but complicated

When the runtime determines that the code is accessing the same *m*page multiple times, it may *cache* the pointer to that *m*page in its private *TLB*. This avoids the address translation overheads for all but the first access to the page. To ensure that the *m*page is not swapped out by the *m*store logic, the runtime notifies the *m*store to pin the *m*page in the *m*page cache via `notify_tlb_cached`. The page is unpinned by `notify_tlb_dropped` when its translation is evicted from the TLB (see §3.3.4 for more details).

There can be multiple cached *m*stores in the same program, each with its own *m*page size. The runtime can query an *m*store page size using the `get_mpage_size` call, for example, to determine accesses to the same *m*page in the TLB.

A cached *m*store must implement the `flush()` callback to synchronize its *m*page cache with the storage layer. This callback is used, for example, to implement the `msync` call.

Direct-access *m*stores. Direct-access *m*stores have no cache and thus are easier to implement. The input pointer provided to the `mpf_handler_d` callback may be used without address translation, and at finer granularity not bound to the *m*page size. The runtime provides a thread-local intermediate buffer to store the accessed data. For loads, the program uses this buffer. For stores, the runtime writes the updated contents back to the *m*store using the `write_back` callback.

3.2.3 Thread safety and memory consistency

CoSMIX allows multiple threads to access the same *m*store, as long as the *m*store implementation is thread-safe.

For cached *m*stores, CoSMIX does not change the *m*store memory consistency model as long as the accesses are inside the same *m*page, and the *m*page size is 4KB or larger. The CoSMIX runtime does not introduce extra memory copies for such accesses and effectively inherits the *m*store memory consistency. In addition, the *m*store itself must ensure that the storage layer is accessed only via its *m*page cache, thereby preventing different threads from seeing inconsistent data.

This guarantee does not hold for direct-access *m*stores and misaligned cross-*m*pages. The primary implication is that hardware atomics will not work for such accesses.

We believe that this limitation does not affect most practical cases. The lack of cross-*m*page atomics support does not affect race-free programs synchronized via locks. This is because the intermediate buffer is written back to the *m*store immediately after the original store operation and thus will be protected by the original lock. We observed no cases of cross-*m*page atomics in the evaluated applications.

Today, the problem of misaligned cache accesses deserves special handling in compilers. For example, LLVM translates such accesses into the `XCHG` instruction [43]. CoSMIX must rely on a software solution, e.g., using readers-write locks per-*m*page. We defer this to future work.

3.2.4 Memory vs. file-backed *m*stores

The *m*store abstraction described so far instruments memory accesses alone. However, it is insufficient to enable implementation of memory-mapped files. For example, consider a program that uses `mmap` to access a file, and then calls `fsync`. It will not work correctly because `fsync` is not aware of the *m*store's internal cache. Specifically, all the I/O operations on files managed by a file-backed *m*store must interact with its *m*page cache to avoid data inconsistency.

We define a *file-backed* cached *m*store type, which implements all the callbacks of memory-backed *m*stores, but additionally overrides the POSIX file API that interacts with the page cache, e.g., `open, close, read, write, msync` (§3.3). Direct-access *m*stores do not have internal caches; thus they can be used with files without overriding file I/O operations other than `mmap` itself.

3.2.5 *M*store examples

CoSMIX provides a set of reusable building blocks for *m*stores, such as a slab memory allocator, spinlocks, a generic *m*page table, and an *m*page cache, all with multi-threading support, which we use to implement the *m*stores below.

SUVM *m*store. SUVM allows exit-less memory paging for enclaves by keeping a page table, page cache and a fault handler as part of the enclave. We implement SUVM from scratch as a cached memory-backed *m*store, using CoSMIX's generic *m*page table and *m*page cache. The `alloc` function returns a pointer to the storage layer in untrusted memory. Upon `mpf_handler_c`, the *m*store checks whether the needed *m*page is already cached in the *m*page table. If not, it reads the *m*page's contents from the storage layer, decrypts and verifies its integrity using a signature maintained for every *m*page, and finally copies it to the *m*page in the page cache. When the *m*page cache is full, the *m*store evicts pages back into the storage layer.

Secure `mmap` *m*store. This *m*store enables the use of memory-mapped encrypted and integrity-protected files in enclaves. We support private file mapping only and leave cross-enclave sharing support for future work. This is a cached file-backed *m*store that maintains its own *m*page table and *m*page cache.

The `alloc` callback is invoked by the runtime upon the `mmap` call in original code. `alloc` records the mapping start offset in the file and the access flags in an internal table. It then returns a pointer with the requested offset from a unique file base address. This address is internally allocated by the *m*store and used as isolated address space for each file.

The `mpf_handler_c` callback translates the given pointer to the respective file. If the contents are not available in the *m*page cache, the data is fetched from the file using a `read` system call, followed by decryption and integrity check.

Oblivious RAM *m*store. ORAM obfuscates memory access patterns by shuffling and re-encrypting the data upon every data access. The ORAM *m*store streamlines the use of ORAM in enclaves. It allows the developer to allocate and access the buffers that store sensitive data in an ORAM, thereby protecting the program against controlled side-channel attacks [87].

We implement a direct-access memory-backed ORAM *m*store. This is because if it were cached, the accesses to the cache would be visible to the attacker, compromising the ORAM security guarantees. Our ORAM *m*store addresses a threat-model similar to ZeroTrace [67], yet without leakage protection in the case of enclave shutdowns. Specifically, all the instrumented memory accesses destined to the ORAM *m*store become oblivious, such that an adversary cannot learn the actual memory access pattern. We implement the Path ORAM algorithm [78] and ensure oblivious accesses to its position map and stash using the *cmovz* instruction.

We store the Path-ORAM tree in a contiguous buffer within the enclave trusted memory. This eliminates the need to implement block encryption and integrity checks as part of the ORAM *m*store since SGX hardware does exactly that.

The `alloc` function allocates the requested number of blocks in ORAM and registers them with the ORAM module. It returns an address from a linear range with a hard-coded base address, which is used only to compute the block index.

The `mpf_handler_d` callback translates the address to the requested block index and invokes the ORAM algorithm to obliviously fetch the requested memory block to a temporary buffer. Loads are issued from this buffer and stores are appended with the `write_back` callback.

3.2.6 Stacking *m*stores

The *m*store abstraction makes it possible to *stack* different *m*stores. Stacking allows one *m*store to invoke another *m*store recursively. We denote by A→B an *m*store A that internally accesses its memory via *m*store B.

Consider, for example, ORAM→SUVM. The motivation to stack is when the ORAM *m*store covers a region that is larger than the enclave's physical memory. Since SUVM optimizes the SGX paging mechanism, stacking ORAM on top of SUVM improves ORAM's performance (§4.5).

To create a new A→B *m*store, the developer simply annotates A's storage layer allocations with B's annotations. CoSMIX instruments all these accesses appropriately.

Stacking the ORAM *m*store on top of any *m*store that maintains data confidentiality does not compromise the ORAM access pattern obfuscation guarantees, as ORAM protocols consider the backing store to be untrusted [78]. Therefore ORAM→SUVM would maintain data-oblivious access.

However, the stacking → operator is *not commutative* from the security perspective: SUVM→ORAM would result in the SUVM *m*store caching *m*pages fetched obliviously by the ORAM *m*store, thereby leaking the access patterns to these *m*pages and compromising ORAM's security guarantees.

3.3 CoSMIX compiler and runtime

The instrumentation compiler modifies the application to use *m*stores and is guided by code annotations or/and a global instrumentation policy. The compiler needs to instrument four different types of code: (1) memory accesses; (2) memory management functions; (3) file I/O operations for file-backed *m*stores; (4) `libc` library calls.

Instrumentation policy. A developer may annotate any static variable declaration or memory allocation function call. Annotations allow instrumentation of a subset of the used buffers to reduce instrumentation overheads. Alternatively, a global instrumentation policy specifies compile-time rules applied to the whole code base (e.g., instrument all calls to `malloc`), or run-time checks injected by the compiler (e.g., using *m*store for large buffers above a certain threshold). A global policy serves for bulk operations on large code bases, such as adding SUVM *m*store to Redis sources with over 130K LOC (§4.4).

Similarly, for file-backed *m*stores, a global policy may limit the use of the *m*store to specific files or directories.

3.3.1 Pointer access instrumentation

Static analysis. The compiler uses static build. Therefore, it can conservatively determine the subset of operations that must be replaced with *m*store-related functions at compile time and eliminate the instrumentation overhead for such cases. Trivially, the compiler may replace an annotated call to `malloc` with the `alloc` callback of the requested *m*store. A much more challenging task, however, is to determine *the type of the* m*store (if any) to use for every pointer in the program*.

For this purpose, we use Andersen's analysis [17] to generate inter-procedural point-to information. In a nutshell, CoSMIX first parses all instructions and generates a graph with pointers as nodes and their constraints (e.g., assignment or copy) as edges. The graph is then processed by a constraint solver which outputs the set of points-to-information.

When instrumenting memory accesses, CoSMIX can use this information to determine whether the pointer may alias to a specific *m*store pointer.

Runtime checks and tagged pointers. CoSMIX's pointer analysis is sound but incomplete; therefore it requires runtime decisions for ambivalent cases. We use tagged pointers [47, 13, 74, 31] to determine pointer type at runtime. Each *m*store is assigned a unique identifier, stored in unused most significant bits of the pointer virtual address. For instrumented allocations, the runtime adds this identifier to the returned address from the *m*store allocation. For external function calls and memory accesses, the runtime checks the tag, strips it from the pointer, and invokes the callback of the respective *m*store if necessary.

Tagged pointers vs. range checks. One known limitation of tagged pointers is that the application code might reset the higher bits of a pointer. Prior work [47] and our own experience suggest that this is rarely the case in practice. An alternative approach is to differentiate between *m*stores by assigning a unique memory range to each. Using tagged pointers turned out to be faster in our experience because the range check requires additional memory accesses.

3.3.2 Memory management and file I/O calls

The compiler replaces all the memory management operations selected by the instrumentation policy with the calls to the runtime that invokes the appropriate *m*store callbacks.

Similarly, file I/O operations are replaced with runtime wrappers. On `open`, the runtime determines whether to use an *m*store with the current file and registers its file descriptor. An I/O call using this file descriptor will be redirected to the respective *m*store.

3.3.3 `libc` support

Invoking an uninstrumented function on an *m*store pointer would result in undefined behavior. We assume that most application libraries are available at compile time. However, `libc` is not instrumented and we provide wrappers instead, similarly to other works [47, 59].

There are two main reasons to not instrument `libc`. First, doing so would create a bootstrapping problem since *m*stores might use `libc` calls. Second, SGX runtimes such as SCONE [18] use proprietary, closed-source versions of `libc`. Wrapping `libc` functions allows CoSMIX to be portable across multiple enclave execution frameworks.

CoSMIX provides wrappers for most popular `libc` functions (about 80), which suffices to run the large production applications we use in our evaluation. Adding wrappers for more functions is an ongoing effort.

In addition to stripping the pointer tag, these wrappers must guarantee access to virtually contiguous input/output buffers from uninstrumented code, instead of using *m*store *m*pages. Thus, where necessary, the wrappers use a temporary buffer in regular memory to stage the *m*store buffer before the library call and write it back to the *m*store after the call.

3.3.4 Translation caching

Minimizing the instrumentation overhead is a fundamental requirement for CoSMIX. The overheads are caused mainly by runtime checks of the pointer type and invocation of the *m*store logic on memory accesses.

To reduce these overheads, CoSMIX first runs aggressive generic code optimizations, reducing memory accesses. It

also avoids invoking *m*store page fault handlers for recurrent accesses to the same *m*page.

Opportunistic caching. We introduce a small (one cache line) TLB stored in the thread-local memory. This TLB is checked upon each access to the *m*store. The runtime pins the page in the *m*store while the *m*page translation is cached, and unpins when it is flushed. To support multiple threads, the TLB notification callbacks use a reference counter for each *m*page. The *m*page can be evicted if the counter drops to zero, eliminating the need for explicit TLB invalidation. We choose small TLB size (5) for its low lookup times. Increasing the size did not improve performance in our workloads.

Translation caching in loops. The TLB captures the locality of accesses quite effectively, but in loops, the performance can be further improved by transforming the code itself to use the *m*page base address without checking the TLB.

For example, in the case of an array allocated in an *m*store and sequentially accessed in a tight loop, most accesses to the *m*store are performed within the same *m*page. Therefore, replacing the code in the loop to check the TLB only at m*page boundaries* would result in near-native access latency.

To perform this optimization, CoSMIX has to (a) determine the iterations in which the same *m*page gets accessed, and (b) determine the pointer transformation across the iterations. For (b), CoSMIX uses the scalar evolution analysis [83] in the compiler to find predictable affine transformations for the pointers used in the loop. For (a) it injects a code that determines the number of iterations where the cached translation hits the same *m*page, recomputing the new base pointer and dropping the translation from the TLB when crossing into a new *m*page. Finally, it replaces the original accesses to the *m*store with the accesses to the *m*page's base pointer with the offset, which is updated across the loop iterations according to the determined transformation.

3.4 Discussion

Security guarantees. CoSMIX itself does not change the security of SGX enclaves. Its runtime neither accesses untrusted memory nor leaks secret information from the enclave. However, the security of *m*stores depends on their implementation: SUVM has the same security as SGX paging [61] and the ORAM *m*store introduces a controlled side-channel [87] protection mechanism not available in the bare-metal SGX [67]. We note, however, that CoSMIX does not guarantee that the code using an *m*store will indeed maintain that *m*store's security properties. For example, in case of the ORAM *m*store, user code must not use sensitive values read from ORAM in a data/control dependent manner because doing so might break the data pattern obfuscation [65].

Other *m*store applications. *M*stores are general and can be used to implement many other useful extensions. For example, implementing bounds checking for 32-bit address space enclaves as in SGXBounds [47] becomes easy. All it takes is adding an extra 4 bytes for each buffer allocation to store the lower bounds of the object and tag the pointer's highest 32 bits with its lower bounds' address. Then, every memory access is instrumented to check these bounds. The SGXBounds *m*store can implement this logic in its callback functions.

Another useful application is transparent inter-enclave shared memory, which may enable execution of multi-socket enclaves and support for secure file sharing.

Eleos vs. CoSMIX. The starting point of our design was Eleos [61]. There, the authors introduce *s*pointers, which are similar to C++ smart pointers. In Eleos all necessary memory accesses are replaced with *s*pointers and translations are cached in the *s*pointers themselves. On every access, *s*pointers perform bound checking, to make sure that pointer arithmetic on the *s*pointer did not cross to a new page. However, the *m*page bound check impacts performance greatly, even when the caching is limited to the scope of a function. Second, maintaining static translation for *m*store pointers complicated the design. For example, pointer-to-integer casts had to be invalidated, forcing a reverse mapping in *m*stores.

A key lesson from CoSMIX is that *caching the translations only in cases of high access locality is enough to leverage the performance benefits and simplify the design.*

Limitations. Inline assembly snippets, while quite rare, cannot be easily supported. CoSMIX considers them as an opaque function call. It injects code to check whether passed arguments are *m*store pointers. If so, CoSMIX aborts the program and notifies that manual instrumentation is necessary.

Hardware extensions. We hope that CoSMIX will motivate hardware developers to support *secure in-enclave fault handling*. This functionality would allow enclaves to control the execution flow of the page faults. For example, an enclave might refuse to resume execution after a fault unless a correct secure fault handler has been invoked. As a result, secure page faults would allow extending enclaves with cached *m*store functionality, such as the secured `mmap` provided by CoSMIX, albeit at a significantly higher performance costs due to transitions to/from untrusted mode. Moreover, hardware support for secure fault handlers would enable paging of code pages not supported by CoSMIX.

However, direct *m*stores such as ORAM cannot be supported in the same way, since they invoke the fault handler for every memory access. Therefore, good performance could be achieved only by much more intrusive modifications that would avoid enclave mode transitions. Additionally, enclave hardware evolves slowly. For example, the SGX2 specification was published in 2014, yet is still not publicly available in mainstream processors [20]. CoSMIX on the other hand, can be used to enhance enclaves' functionality today.

4 Evaluation

Implementation. CoSMIX implementation closely follows its design [2]. The compiler prototype is based on the LLVM 6.0 framework and is implemented as a compile-time module pass, consisting of 1,080 LOC. We compile applications using the Link-Time Optimization (LTO) feature of LLVM and pass them as inputs to the compiler pass.

CoSMIX uses the SVF framework [79, 80] to perform Andersen's pointer analysis with type differentiation. CoSMIX runtime is written in C++ and consists of 1,600 LOC. CoSMIX's configuration file is `JSON` formatted and CoSMIX uses JsonCpp [3] to parse it. All *m*stores are written in C++. Their implementation follows the design described in Section 3. The implementations of SUVM, ORAM and `mmap` *m*stores are 935 LOC, 551 LOC, and 1,108 LOC respectively. `mmap` *m*store leverages the SCONE file shields, which override the `read/write` calls in `libc` to implement integrity checks and encryption for file I/O operations. This is one of the examples when our design choice to use `libc` wrappers rather than `libc` instrumentation pays off.

Setup. Our setup comprises two machines: server and client. The client generates the load for the server under evaluation. The client and the server are connected back-to-back via a 56Gb Connect-IB operating in IP over Infiniband [27] (32Gbps effective throughput) to stress the application logic and avoid network bottlenecks.

For the server, we use Dell OptiPlex 7040, with Intel Skylake i7-6700 4-core CPU with 8 MB LLC, 16 GB RAM, and 256 GB SSD drive, Ubuntu Linux 16.04 64-bit, Linux 4.10.0-42, and Intel SGX driver 2.0 [2]. We use LLVM 6.0 to compile the source code. As recommended by Intel, we apply L1TF microcode patches [9]. The client runs on a 6-core Intel Xeon CPU E5-2620 v3 at 2.40GHz with 32GB of RAM, Ubuntu 16.04 64-bit, Linux 4.4.0-139.

In my papers, be sure to mention the microcode firmware.

Methodology. Unless otherwise specified, we run all the workloads in SGX using SCONE [18]. We run each test 10 times and report the mean value, with the standard deviation below 5%. We compile all workloads as follows: (1) compile to LLVM IR with full optimizations (-O3) and invoke LLVM's IR linker to link all IR files; (2) invoke CoSMIX LLVM pass for code instrumentation; (3) use the SCONE compiler to generate an executable binary linked with its custom libc. We skip step (2) when compiling the baseline.

Summary of workloads. We evaluated several production applications and benchmarks, detailed in Table 2. CoSMIX successfully instruments large code bases using only *a few or no code annotations, and no source code changes*.

[2] CoSMIX source code is publicly available at https://github.com/acsl-technion/cosmix.

Workload	LOC	Changed LOC	*m*store
`memcached` [35]	15,927	1	SUVM
Redis [8]	123,907	0	SUVM
SQLite [62]	134,835	2	secure `mmap`
Phoenix suite [66]	1,064	1/bench	SUVM ‖ secure `mmap`
Face verification [61]	700	1	SUVM → ORAM

Table 2: Summary of the evaluated workloads. ‖ - side-by-side, → - stacked. LOC includes all statically linked libraries.

Data size	Baseline	SUVM	ORAM
16 MB	0.7μsec	0.9μsec (−1.28×)	32.3μsec (−46.1×)
256 MB	14.4μsec	1.5μsec (9.6×)	590.6μsec (−41×)
1GB	19.9μsec	1.6μsec (12.4×)	1.23*m*sec (−61.8×)

Table 3: *m*store latency to fetch a 4 KB page. Baseline is native memory access.

4.1 *M*store performance

First, we measure the latency of random accesses to *m*stores and compare them to native memory accesses in the enclave. We evaluate scenarios with small and large memory usage where the latter causes SGX page faults. We report the results for SUVM and ORAM *m*stores and exclude the `mmap` *m*store because it is similar to SUVM. We measure the average latency to access random 4 KB pages over 100k requests.

Table 3 shows the results. For small datasets, SUVM incurs low overhead compared to regular memory accesses. However, for large data sets which involve SGX paging, it is about 10× faster. This is because SUVM optimizes the enclave paging performance by eliminating enclave transitions [61]. As expected, ORAM *m*store access is between 30× to 60× slower than the baseline, even when covering a small range of 16 MB. This result indicates that selective instrumentation for ORAM is essential to achieve practically viable systems.

4.2 Instrumentation and *m*store overheads

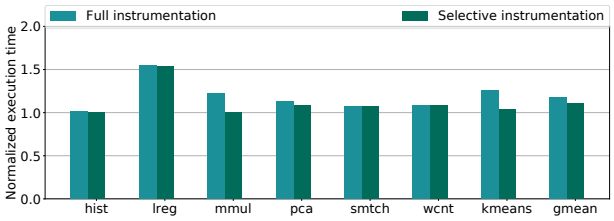
We instrument all seven benchmarks in the Phoenix suite [66] to measure CoSMIX's instrumentation overheads. Each benchmark is small, making the results easier to analyze.

We evaluate 4 configurations: (1) Full automatic instrumentation using SUVM and `mmap` *m*store. Both *m*stores are run side-by-side because Phoenix uses both dynamic allocations and memory-mapped files. (2) Same but with an *empty* *m*store. All the pointers are instrumented, but the *m*store logic is not invoked. (3) Selective instrumentation where we manually annotate only the inputs. (4) Same but with an empty *m*store. For benchmarks that use `mmap`, the baseline reads the entire input file to memory.

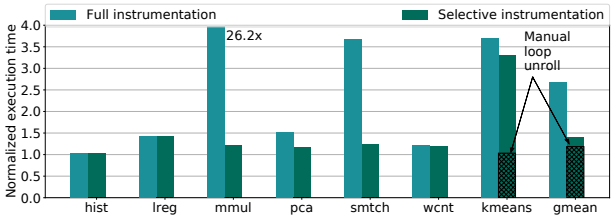
Good analytical process

Measuring an empty *m*store allows us to distinguish between overheads of pointer instrumentation and *m*stores.

To focus solely on the CoSMIX overheads, we use the small dataset shipped with Phoenix so no paging operation will occur. The only exception is the histogram benchmark, for which we synthetically resize the dataset to 25 MB. We exclude *m*store initialization and preload all the datasets into



(a) Instrumentation-only cost (empty mstores).



(b) Full cost with mstores.

Figure 4: CoSMIX instrumentation overheads for the Phoenix suite running in the enclave normalized to the execution of non-instrumented binaries. Lower is better.

SGX memory, both in the baseline and in CoSMIX measurements, to stress the runtime components of the system.

Often *m*stores can be tuned to reduce the translation overhead, by increasing the *m*store page size. As a result, the accesses in a loop might touch fewer pages, enabling more efficient use of CoSMIX's TLB. Therefore, we manually tune the page size, setting it to 256 KB for kmeans, 16 MB for word count, 64 MB for lreg, and 4 KB for the rest of the benchmarks, as in all the other experiments.

Figure 4 shows the results. Figure 4b shows the overhead for both CoSMIX's instrumentation and *m*store logic. Figure 4a excludes the *m*store logic. In each figure, the rightmost bar refers to the selective instrumentation of accesses to the input data alone, and the other bar refers to full instrumentation of all dynamic allocations and `mmap` calls.

Instrumentation overheads. The runtime overheads excluding *m*stores are relatively small, with an average (geomean) of 17% for full instrumentation and 10% for the input instrumentation alone, with the worst case of 50% in lreg.

Full instrumentation. With the full instrumentation, *m*store logic dominates the runtime overheads, ranging from almost none for histogram to 26× for matrix multiplication. Such variability stems from the different ways memory is accessed. Specifically, if the program exhibits poor access locality, or the CoSMIX compiler fails to capture the existing locality in a loop, the runtime will not be able to optimize out the calls to the *m*store inside the loop, resulting in high overheads.

Selective instrumentation. Instrumenting only the input buffers results in dramatically lower overheads, ranging from 5% to 15%. The only pathological case is kmeans, where the CoSMIX compiler fails to optimize accesses to the multi-dimensional input array because the inner array is too short. Unrolling this loop reduces the overhead to about 5%. We plan to add automatic optimizations for this case in the future.

	CoSMIX secure `mmap`	`read` **no cache**	`read` **1 MB cache**	`read` **16 MB-60 MB cache**
Query latency	2.4μsec	10.7μsec	4.5μsec	1.7μsec
Speedup		4.4×	1.8×	−1.4×

Table 4: SQLite performance with secure `mmap` *m*store.

Contribution of CoSMIX optimizations. We measure the performance of the *selective* instrumentation while disabling the runtime TLB and compiler loop optimizations (§3.3.4). We find that these two features *are essential* to make CoSMIX practical and keep the instrumentation overheads low. The slowdown ranges from 4× for word count and kmeans to 55× for histogram and 197× for lreg, making the system unusable. In comparison, the optimized version brings the overheads down to 4% and 44% for histogram and lreg. The geomean of the unoptimized selective instrumentation is 16.4× compared to 20% with the optimizations enabled.

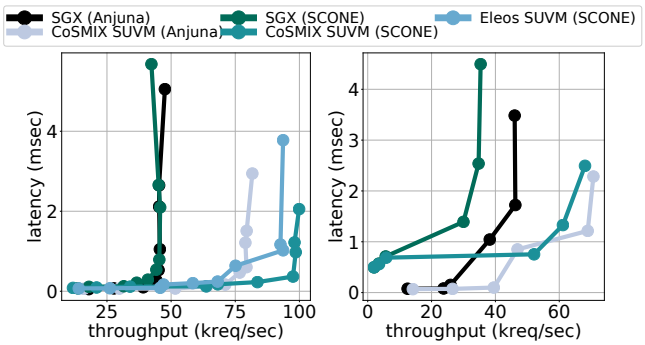
CoSMIX overheads for non-enclave execution. For completeness we perform the same experiment with Phoenix but now outside the enclave. As expected, the relative overheads increase as compared to the in-enclave execution, with up to 50% geometric mean slowdown when using selective instrumentation with SUVM and `mmap` *m*stores and up to 25% when using an empty *m*store. We attribute this discrepancy to SGX's memory encryption engine [40], which provides confidentiality and integrity to memory accesses and therefore offsets the CoSMIX instrumentation overheads.

4.3 Secure `mmap` with SQLite

SQLite is a popular database, but running it in SGX with `mmap` while providing encryption and integrity guarantees for the accessed files is not possible today. To run SQLite with `mmap` support, we use the secure `mmap` *m*store. We configure the *m*page size to be 4 MB. We use SQLite v3.25.3, and evaluate it with kvtest [10], shipped with SQLite to test read access latency to DB BLOBs. We use a database stored in a 60 MB file holding 1 KB BLOBs. The database is sized to fit in SGX physical memory. This allows us to focus on the evaluation of the file access logic rather than SGX paging (refer to §4.4 for paging evaluation).

As a baseline, we evaluate SQLite with its internal backend that uses `read/write` calls instead of memory-mapped files. In this configuration, SQLite implements its own optimized page cache for data it reads from files. In the evaluation, we vary the SQLite page cache size from disabled (1 KB) to 60 MB (no misses). We measure the average latency of 1 KB random read requests over 1 million requests.

Results. CoSMIX enables execution of an unmodified SQLite server that uses `mmap` to access encrypted and integrity-protected files. Such execution was not possible without CoSMIX. Moreover, as we see in Table 4, the secure *m*store enables 4.4× faster queries compared to the SQLite without



(a) Memcached. 10% SET 90% GET for 1 KB items.

(b) Redis. 100% GET for 1 KB items.

Figure 5: Performance improvement using the SUVM *m*store in production key-value stores, each using a 600 MB database (6× the size of SGX secure memory)

page cache. This case illustrates the CoSMIX performance benefits for applications that do not implement their own optimized page cache.

On the other hand, enabling the SQLite page cache allows us to evaluate the instrumentation overheads. This is, in fact, a conservative estimate, because the baseline is hand-optimized and implements the necessary functionality by itself, versus the general instrumentation and generic CoSMIX's page cache. Even in this worst-case scenario, CoSMIX is only about 40% slower than SQLite.

4.4 Optimizing memory-intensive workloads with the SUVM *m*store

To demonstrate CoSMIX's support for different SGX execution frameworks, we run the following experiments both in SCONE [18] and Anjuna [1].

We use CoSMIX to accelerate SGX execution of applications with a large memory footprint. We choose Redis and `memcached` key-value stores as representatives. Both run with data sets of 600 MB – about 6× larger than the SGX enclave-accessible physical memory. These applications experience a significant slowdown due to SGX paging overheads. The goal is to reduce these overheads using the SUVM *m*store.

Memached. `memcached` uses a slab allocator to manage its memory. We annotate a *single line of code* where the memory is allocated, making SUVM *m*store manage all items.

We evaluate `memcached` v1.4.25 [35] using the memaslap load generator shipped with libmemcached v1.0.18 [6]. Our workload consists of 10% SET and 90% GET requests for 1 KB items (key+value) as used in prior works [61], with requests uniformly distributed to stress the memory subsystem.

We compare the instrumented `memcached` with SUVM to native SGX execution. In addition, we run a *manually optimized* version of `memcached` with the SUVM used in Eleos [61]. Notably, in Eleos, the authors changed `memcached` internals to create a shadow memory buffer for the slab allocator. This is an intrusive change that CoSMIX eliminates completely. All runs are performed on 4 cores. Figure 5a shows that SUVM *m*store boosts the throughput by 1.9× and 2.2× compared to native SGX execution in Anjuna and SCONE respectively. The difference between the frameworks correlates with the relative time each of them spends resolving page faults. Interestingly, the CoSMIX version is about 7% *faster* than Eleos thanks to its compiler optimizations.

Redis. Manually annotating Redis with its 130 KLOC would be too tedious. Further, its memory management involves too many allocations, making annotation challenging. Therefore, we use automatic instrumentation *without code changes*.

To achieve high performance, we leverage CoSMIX's ability to perform conditional instrumentation. Specifically, CoSMIX introduces *runtime checks* that determine whether to allocate a buffer in an *m*store or in regular memory based on the requested allocation size. In Redis, we configure the policy to redirect all allocations in the range of 1 KB-10 KB to the SUVM *m*store. The intuition is to use SUVM only for keys and values and keep native memory accesses for the rest.

We use Redis v5.0.2 [8], and evaluate it using the memtier v1.2.15 official RedisLab load generator [7]. We configure memtier to generate uniformly distributed GET requests for 1 KB items as used in prior works [18].

Figure 5b shows that Redis with CoSMIX achieves about 1.6× and 2× higher throughput compared to native SGX execution in Anjuna and SCONE respectively. *These results demonstrate the power of CoSMIX to improve the performance of an unmodified production system.*

4.5 Protecting data with the ORAM *m*store

Selective instrumentation capabilities in CoSMIX are particularly useful when using heavyweight *m*stores such as ORAM. ORAM is known to dramatically affect performance (Table 3).

We use ORAM to protect a face verification server [61] against controlled side-channel attacks [87] on its data store.

The server mimics the behavior of a biometric border control server. It stores a database with sensitive face images. When a person passes through border control, the client at the border kiosk queries the server whether the image of the person in the database matches the one taken at the kiosk.

The implementation stores the images in an array. The server fetches the image from the array and compares it with the input image, using the LBP algorithm [12]. This implementation is vulnerable to controlled channel attacks which leak the access pattern to SGX memory pages. Thus, an attacker may observe page access frequency and learn when a person passes through border control. Note that existing defenses against controlled channel attacks would be ineffective since they cannot handle legitimate demand paging [76, 60].

We use a database with 1,024 256 KB images from the Color FERET dataset [63], totaling 256 MB of sensitive data. We annotate the allocation of the database array to use ORAM

	Native SGX	ORAM	ORAM→SUVM
Throughput(req/sec)	203.1	23.4	34.7
Slowdown		8.6×	5.8×

Table 5: Selective instrumentation of face verification server.

(1 LOC) and compile with CoSMIX. As a result, the application accesses the sensitive data obliviously. In the experiments we configure the server to use 1 thread and the load generator to issue random requests, saturating the server.

We report the throughput achieved in Table 5. It shows that the ORAM *m*store introduces an 8.6× slowdown. We note that for a dataset of 256 MB, the ORAM *m*store overhead is about 41× more than native access, as reported in Table 3. This shows that selective instrumentation may make ORAM an attractive solution for some systems. However, with CoSMIX we can further reduce paging penalties.

ORAM→SUVM stacking. Although the application data is only 256 MB, PathORAM consumes about 860 MB due to its internal storage overheads. As a result, the SGX paging significantly affects the performance.

To optimize, we create a new ORAM→SUVM *m*store by stacking ORAM on top of SUVM. Only 1 LOC in the ORAM *m*store is annotated. The use of the combined ORAM→SUVM *m*store improves the overall performance by 1.5× compared to the ORAM *m*store alone (Table 5). Overall, selective instrumentation and *m*store stacking result in a relatively low, 5.8× slowdown of the oblivious system compared to native SGX execution. This performance might be acceptable for practical ORAM applications and requires no code changes.

5 Related Work

Enclave system support. Recent works proposed systems to protect unmodified applications using enclaves [18, 21, 77, 81]. Other works proposed enclave enhancements [72, 47, 49], such as memory safety, ASLR, and enclave partitioning. Complementary to these works, CoSMIX provides system support for modular extensions to unmodified enclave applications.

Trusted execution environments. Previous works proposed different systems to protect applications from a malicious OS [30, 29, 42, 16, 15]. InkTag [42], for example, offers secured `mmap` service to applications; however, it relies on a trusted hypervisor with para-verification. CoSMIX puts its root of trust in hardware enclaves to implement in-enclave secure fault handlers.

Controlled side-channel mitigation. Previous works suggested the use of ORAM in SGX to improve its security [65, 67, 33, 88, 11, 50]. We believe that CoSMIX will allow the use of ORAM in many more applications via lightweight annotations. More efficient systems for mitigating the controlled side-channel attack have been proposed [22, 60, 76, 30]. For example, Apparition [30] uses Virtual Ghost [29], a compiler-based virtual machine, to restrict OS access to the page table. However, these systems do not support demand paging. CoSMIX protects applications that rely on demand paging from both a malicious OS and physical bus snooping attacks using the ORAM *m*store.

Valuable

Customizing applications via paging mechanisms. Previous systems proposed using page faults to improve performance and enable quality of service functionality in OS, GPUs, and enclaves [41, 61, 74, 14]. CoSMIX enables using similar enhancements in secure enclave systems.

Recent works take advantage of the RDMA infrastructure to enable efficient and transparent access to remote memory by customizing the OS page fault handler [39, 75, 53, 37]. For example, LegoOS [75] uses paging to simulate an *ExCache* for disaggregated memory support. CoSMIX takes a similar approach for extending secure enclaves.

Software-based distributed shared memory systems were proposed to customize memory access logic across remote machines [68, 69, 70, 64, 19, 56, 32]. These systems either use the page fault handler, runtime libraries or instrumentation of applications. CoSMIX is inspired by these systems; however, the *m*store abstraction is more general and can support different memory access semantics, mixed or stacked in the same application.

Memory instrumentation. Much work has been done on instrumenting memory accesses using binary instrumentation tools [51, 24, 57, 52, 71] and compiler-based instrumentation [73, 47, 13, 31]. CoSMIX enables low-overhead selective memory instrumentation, specifically tailored for enclaves.

6 Conclusions

CoSMIX is a new system for modular extension of the memory layer in unmodified applications running in enclaves. It sidesteps the lack of hardware support for efficient and secure page fault handlers in existing architectures. CoSMIX enables low-overhead and selective instrumentation of memory accesses via a simple yet powerful *m*store abstraction.

We show how compilation with CoSMIX both speeds up execution and adds protection to production applications. We believe that *m*stores may become a useful tool for facilitating the development of new secured systems.

7 Acknowledgments

We would like to thank our shepherd John Criswell for his valuable feedback. We also gratefully acknowledge the support of the Israel Science Foundation (grant No. 1027/18), the Israeli Innovation Authority Hiper Consortium, the Technion Hiroshi Fujiwara Cybersecurity center, and Intel ICRI-CI Institute grant, and the feedback from the Intel SGX SEM group. Meni Orenbach was partially supported by HPI-Technion Research School.

References

[1] Anjuna. https://www.anjuna.io. Accessed: 2019-01-01.

[2] Intel SGX Linux Driver. https://github.com/intel/linux-sgx-driver. Accessed: 2018-12-06.

[3] JsonCpp. https://github.com/open-source-parsers/jsoncpp. Accessed: 2019-01-09.

[4] Keystone: Open-source Secure Hardware Enclave. https://keystone-enclave.org. Accessed: 2019-01-09.

[5] Lightning Memory-Mapped Database Manager. http://www.lmdb.tech/doc/. Accessed: 2018-12-06.

[6] memaslap: Load Testing and Benchmarking a Server. http://docs.libmemcached.org/bin/memaslap.html. Accessed: 2018-12-06.

[7] memtier benchmark: A High-Throughput Benchmarking Tool for Redis and Memcached. https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/. Accessed: 2018-12-06.

[8] Redis In-Memory Data Structure Store. https://redis.io/. Accessed: 2018-12-06.

[9] Resources and Response to Side Channel L1 Terminal Fault. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html. Accessed: 2018-12-31.

[10] SQLite Memory-Mapped I/O. https://www.sqlite.org/mmap.html. Accessed: 2018-12-06.

[11] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[12] T. Ahonen, A. Hadid, and M. Pietikainen. Face Description With Local Binary Patterns: Application to Face Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):2037–2041, 2006.

[13] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[14] H. Alam, T. Zhang, M. Erez, and Y. Etsion. Do-It-Yourself Virtual Memory Translation. In *44th Annual International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2017.

[15] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security. *ARM White Paper*, 3(4):18–24, 2004.

[16] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, volume 13, 2013.

[17] L. O. Andersen. *Program Analysis and Specialization for The C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[18] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703, 2016.

[19] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992.

[20] A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 132–137. ACM, 2017.

[21] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3), 2015.

[22] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A.-R. Sadeghi. DR. SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv preprint arXiv:1709.09917*, 2017.

[23] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, 2016.

[24] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[25] L. Bryant, J. Van, B. Riedel, R. W. Gardner, J. C. Bejar, J. Hover, B. Tovar, K. Hurtado, and D. Thain. VC3: A Virtual Cluster Service for Community Computation. In *Proceedings of the Practice and Experience on Advanced Research Computing, (PEARC)*, pages 30:1–30:8, 2018.

[26] Z. Chang, D. Xie, and F. Li. Oblivious RAM: A Dissection and Experimental Evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 9(12):1113–1124, 2016.

[27] J. Chu and V. Kashyap. Transmission of IP over InfiniBand (IPoIB). *RFC*, 4391:1–21, 2006.

[28] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium*, pages 857–874, 2016.

[29] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–96, 2014.

[30] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *27th USENIX Security Symposium (USENIX Security)*, pages 1441–1458, 2018.

[31] G. J. Duck, R. H. C. Yap, and L. Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *24th Annual Network and Distributed System Security Symposium, (NDSS)*, 2017.

[32] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-time/Run-time Software Distributed Shared Memory System. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 186–197. ACM, 1996.

[33] S. Eskandarian and M. Zaharia. ObliDB: Oblivious Query Processing using Hardware Enclaves. *arXiv preprint arXiv:1710.00458*, 2017.

[34] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware From Software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 287–305, 2017.

[35] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, 2004.

[36] B. Fleisch and G. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 211–223. ACM, 1989.

[37] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. J. Patel, and W. mei W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–358, 2010.

[38] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[39] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667. USENIX Association, 2017.

[40] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016.

[41] S. M. Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, 1999.

[42] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, 2013.

[43] Intel 64 and IA-32 Architectures. Software Developer's Manual. *Intel Corp*.

[44] S. Jennings. Transparent Memory Compression in Linux. LinuxCon, 2013.

[45] Joshua Lind and Oded Naor and Ittay Eyal and Florian Kelbert and Peter R. Pietzuch and Emin Gün Sirer. Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels. In *Proceedings of the 11th ACM International Systems and Storage Conference (SYSTOR)*, pages 125–125, 2018.

[46] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–132, 1994.

[47] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, pages 205–221, 2017.

[48] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, pages 75–, 2004.

[49] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 285–298. USENIX Association, 2017.

[50] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 87–101, 2015.

[51] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[52] J. Maebe, M. Ronsse, and K. D. Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *4th Workshop On Binary Translation (WBT)*, 2002.

[53] E. P. Markatos and G. Dramitinos. Implementation of a Reliable Remote Memory Pager. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 177–190. USENIX Association, 1996.

[54] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. V. Rozas. Intel®Software Guard Extensions (Intel®SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*, pages 10:1–10:9, 2016.

[55] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[56] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 291–305. USENIX Association, 2015.

[57] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003.

[58] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium*, pages 619–636, 2016.

[59] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28, 2018.

[60] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 227–240, 2018.

[61] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, pages 238–253. ACM, 2017.

[62] M. Owens and G. Allen. *SQLite*. Springer, 2010.

[63] P. J. Phillips, H. Wechsler, J. Huang, and P. J. Rauss. The FERET Database and Evaluation Procedure for Face-Recognition Algorithms. *Image Vision Comput.*, 16(5):295–306, 1998.

[64] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[65] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 431–446. USENIX Association, 2015.

[66] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24. IEEE, 2007.

[67] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[68] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th International Conference on Supercomputing*, pages 245–252. ACM, 1997.

[69] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP

Clusters. In *4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 125–136. IEEE, 1998.

[70] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *ACM SIGPLAN Notices*, volume 29, pages 297–306. ACM, 1994.

[71] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 36–47, 2003.

[72] J. Seo, B. Lee, S. M. Kim, M. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[73] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[74] S. Shahar, S. Bergman, and M. Silberstein. ActivePointers: A Case for Software Address Translation on GPUs. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 596–608, 2016.

[75] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87. USENIX Association, 2018.

[76] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[77] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[78] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, pages 299–310. ACM, 2013.

[79] Y. Sui and J. Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[80] Y. Sui, D. Ye, and J. Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[81] C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 645–658, 2017.

[82] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security)*, pages 991–1008, 2018.

[83] R. A. Van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *International Conference on Compiler Construction*, pages 118–132. Springer, 2001.

[84] X. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.

[85] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 81–93. ACM, 2017.

[86] B. C. Xing, M. Shanahan, and R. Leslie-Hurd. Intel®Software Guard Extensions (Intel®SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*, pages 11:1–11:9, 2016.

[87] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[88] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283–298, 2017.