SGXBOUNDS: Memory Safety for Shielded Execution

Dmitrii Kuvaiskii[†] Oleksii Oleksenko[†] Sergei Arnautov[†] Bohdan Trach[†]
Pramod Bhatotia[*] Pascal Felber[‡] Christof Fetzer[†]

[†]TU Dresden [*]The University of Edinburgh [‡]University of Neuchâtel

Abstract

Shielded execution based on Intel SGX provides strong security guarantees for legacy applications running on untrusted platforms. However, memory safety attacks such as Heartbleed can render the confidentiality and integrity properties of shielded execution completely ineffective. To prevent these attacks, the state-of-the-art memory-safety approaches can be used in the context of shielded execution.

In this work, we first showcase that two prominent software- and hardware-based defenses, AddressSanitizer and Intel MPX respectively, are impractical for shielded execution due to high performance and memory overheads. This motivated our design of SGXBOUNDS—an efficient memory-safety approach for shielded execution exploiting the architectural features of Intel SGX. Our design is based on a simple combination of tagged pointers and compact memory layout.

We implemented SGXBOUNDS based on the LLVM compiler framework targeting unmodified multithreaded applications. Our evaluation using Phoenix, PARSEC, and RIPE benchmark suites shows that SGXBOUNDS has performance and memory overheads of 17% and 0.1% respectively, while providing security guarantees similar to AddressSanitizer and Intel MPX. We have obtained similar results with SPEC CPU2006 and four real-world case studies: SQLite, Memcached, Apache, and Nginx.

1. Introduction

Software security is often cited as a key barrier to the adoption of cloud services [6, 7, 75]. In this context, trusted execution environments provide mechanisms to make cloud services more resilient against security attacks [1, 53].

In this work, we focus on Intel Software Guard Extensions (SGX) [53], a recently proposed set of ISA extensions for trusted execution. Intel SGX provides an abstraction of *secure enclave*— a memory region opaque to other software including the hypervisor and the OS—that can be used to achieve *shielded execution* for unmodified legacy applications on untrusted infrastructure.

Shielded execution aims to protect confidentiality and integrity of applications when executed in an untrusted environment [19, 22]. The main idea is to isolate the application from the rest of the system (including privileged software), using only a narrow interface to communicate to the outside, potentially malicious world. Since this interface defines the security boundary, checks are performed to prevent the untrusted environment from attacks on the shielded application in an attempt to leak confidential data or subvert its execution.

Shielded execution, however, does not protect the program against *memory safety* attacks [74]. These attacks are widespread, especially on legacy applications written in unsafe languages such as C/C++. In particular, a remote attacker can violate memory safety by exploiting the existing program bugs to invoke out-of-bounds memory accesses (aka buffer overflows). Thereafter, the attacker can hijack program control flow or leak confidential data [14, 42].

To validate our claim, we reproduced many publicly available memory safety exploits inside the secure enclave (see §7 for details), including the infamous Heartbleed attack in Apache with OpenSSL [14] as well as vulnerabilities in Memcached [12], Nginx [13], and in 16 test cases from the RIPE security benchmark [80]. These examples highlight that a single exploit can completely compromise the integrity and confidentiality properties of shielded execution.

To prevent exploitation of these bugs, a number of memory safety approaches have been proposed to automatically retrofit bounds checking in legacy programs [17, 20, 26, 35, 55, 58]. Among these, we experimented with two prominent software- and hardware-based memory protection mechanisms in the context of shielded execution: AddressSanitizer [69] and Intel Memory Protection Extensions (MPX) [9], respectively.

Unfortunately, these approaches exhibit high performance and memory overheads, thus rendering them impractical for shielded execution. For instance, consider the motivating example of SQLite evaluated against the `speedtest` benchmark (shipped with SQLite) with increasing working set items. Figure 1 compares the performance and memory overheads of SQLite hardened with AddressSanitizer and Intel MPX running inside an SGX enclave.

The experiment shows that Intel MPX performs so poorly that it crashes due to insufficient memory already after tiny working set of 100 (corresponding to memory consumption of 60MB for the native SGX execution). AddressSanitizer is more stable, but performs up to $3.1\times$ slower than

[Preprint. Original published by ACM.]

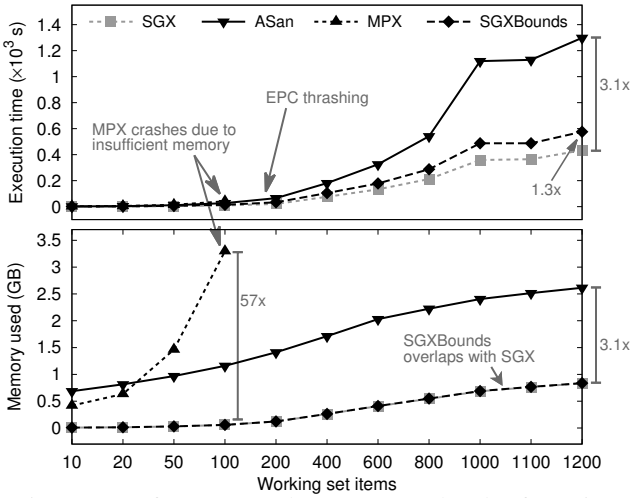


Figure 1: Performance and memory overheads of SQLite.

SGX on larger inputs (with virtual memory consumption of $700-800$MB for the native SGX execution). Additionally, AddressSanitizer consumes $3.1\times$ more virtual memory which can quickly exhaust available memory inside the enclave.

These overheads illustrate a drastic mismatch between memory needs of current memory-safety approaches and the architectural limitations of Intel SGX (high encryption overheads and limited enclave memory, as explained in §2.1). In particular, both AddressSanitizer and Intel MPX incur high memory overheads due to additional metadata used to track object bounds, which in turn leads to poor performance. (We detail the reasons behind the SQLite overheads in §2.3.)

In this paper, we present SGXBOUNDS—a memory-safety approach for shielded execution. Our design takes into account architectural features of SGX and reduces performance and memory overheads to the levels acceptable in production use. For instance, in the case of SQLite, SGXBOUNDS outperforms both AddressSanitizer and Intel MPX, with performance overheads of no more than 35% and almost zero memory overheads with respect to the native SGX execution.

The SGXBOUNDS approach is based on a simple combination of tagged pointers and efficient memory layout to reduce overheads inside SGX enclaves. In particular, we note that SGX enclaves routinely use only 32 lower bits to represent program address space and leave 32 higher bits of pointers unused.[1] We utilize these high bits to represent the upper bound on the referent object (or more broadly the beginning of the object's metadata area); the lower bound value is stored right after the object. Such metadata layout requires only 4 additional bytes per object and does not break cache locality—unlike Intel MPX and AddressSanitizer. Additionally, our tagged pointer approach requires no additional memory lookups for simple loop iterations over arrays—one of the most common cases for memory accesses [32].

Furthermore, we show that our design naturally extends for: (1) "synchronization-free" support for multithreaded applications, (2) increased availability instead of the usual fail-stop semantics by tolerating out-of-bounds accesses based on failure-oblivious computing [64, 65], and lastly, (3) generic APIs for objects' metadata management to support new use-cases.

[1] Current SGX implementations allow 36-bit address space. However, we believe that SGX enclaves spanning more than 4GB of memory are improbable.

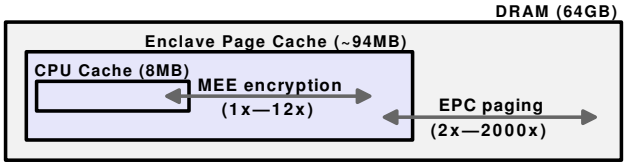


Figure 2: Memory hierarchy and relative performance overheads of Intel SGX w.r.t. native execution [19].

SGXBOUNDS is targeted but not inherently tied to SGX enclaves. Our approach is also applicable to programs that use 64-bit registers to hold pointers but can fit in 32-bit address space. However, as we show in our evaluation, SGXBOUNDS provides no tangible benefits in traditional, unconstrained-memory environments in comparison to other techniques.

We implemented SGXBOUNDS as an extension to the LLVM compiler with several optimizations for performance. Our compiler framework targets unmodified legacy multithreaded applications and thus requires no source code modifications. We evaluated SGXBOUNDS using two multithreaded benchmark suites, Phoenix and PARSEC, and four real-world applications: SQLite, Memcached, Apache, and Nginx. On this set of benchmarks, AddressSanitizer and Intel MPX exhibit high performance overheads of 51% and 75% respectively; memory consumption is $8.1\times$ and $1.95\times$ higher than native SGX. In contrast, SGXBOUNDS shows an average performance slowdown of 17% and an increase in memory consumption by just 0.1%. At the same time, it provides similar security guarantees. Additionally, we evaluated SGXBOUNDS on a CPU-intensive SPEC CPU2006 suite, both inside and outside SGX enclaves.

2. Background and Related Work

2.1 Shielded Execution

Our work builds on SCONE [19], a shielded execution framework to run unmodified applications. SCONE utilizes Intel SGX to provide confidentiality and integrity guarantees.

Intel SGX is a set of ISA extensions for trusted computing released with recent Intel processors [33, 53]. Intel SGX provides an abstraction of *enclave*—a memory region for which the CPU guarantees confidentiality and integrity.

A distinctive trait of Intel SGX is the use of a memory encryption engine (MEE). Enclave pages are located in the Enclave Page Cache (EPC)—a dedicated memory region protected by the MEE (Figure 2). While in main memory, EPC pages are encrypted. When such a page is accessed, the processor verifies that the access originates from the enclave code, fetches the requested data and copies it into the CPU cache. The MEE performs decryption and verifies the integrity of the data. This allows protecting enclaves from attacks launched by privileged software (e.g., by the OS or hypervisor) as well as from physical attacks (e.g., memory bus snooping), thus

	CF	DO	IL
Control Flow Integrity [27, 39, 52, 84]	✔	✘	✘
Code Pointer Integrity [46]	✔	✘	✘
Address Space Randomization [45, 48, 50, 68, 70]	✔*	✘	✘
Data Integrity [16]	✔	✔	✘
Data Flow Integrity [29]	✔	✔	✘
Software Fault Isolation [39, 79]	✔	✔	✔
Data Space Randomization [24, 28]	✔*	✔*	✔*
Memory safety [9, 17, 20, 26, 35, 55, 58, 69]	✔	✔	✔

*SGX enclaves do not provide sufficient bits of entropy in random offsets/masks

Table 1: Current defenses against attacks [74]. **CF** – control flow hijack, **DO** – data-only attack, **IL** – information leak.

reducing the Trusted Computing Base (TCB) to the enclave code and the processor.

The EPC is a limited resource and is shared among all enclaves. Currently, the size of the EPC is 128 MB. Approximately 94 MB are available to the user while the rest is reserved for the metadata. To enable creation of enclaves with sizes beyond that of the EPC, SGX features a paging mechanism. The operating system can evict EPC pages to an unprotected memory using SGX instructions. During eviction, the page is re-encrypted. Similarly, when an evicted page is brought back, it is decrypted and its integrity is checked. Paging incurs high overhead, from $2\times$ for sequential memory accesses and up to $2000\times$ for random ones [19].

SCONE is a shielded execution framework that enables unmodified legacy applications to take advantage of the isolation offered by SGX [19]. With SCONE, the program is recompiled against a modified standard C library (SCONE libc), which facilitates the execution of system calls. The address space of an application is confined to only enclave memory, and the untrusted memory is accessed only via the system call interface. Special wrappers copy arguments of system calls inside and outside the enclave and provide functionality to transparently cryptographically protect any data that might otherwise leave the enclave perimeter in plaintext (so-called *shields*).

Clearly, the combination of SCONE and SGX is not a silver bullet. As we showcase in §7, bugs in the enclave code itself can render these mechanisms useless: we reproduced bugs in Memcached, Nginx, and the infamous Heartbleed attack, all *inside* the SGX enclave and running under SCONE. Thus, it is necessary to defend against data leaks such that the attacker cannot reveal confidential information even in the presence of exploitable vulnerabilities.

To choose the right defense against information leaks, we first discuss the applicability of state-of-the-art defenses for shielded execution and SGX (based on the classification by Szekeres et al. [74]). Table 1 highlights that most state-of-the-art defenses target control-flow hijack attacks only. Even if a proposed defense claims to protect against information leaks, it usually implies that the attacker *can* obtain confidential data in plaintext but *cannot* launch a hijacking attack based on these leaks [21, 34, 70, 72, 73]. Also note that Address Space Randomization (ASR) and its fine-grained variants [31, 34, 41, 50] do not have sufficient bit entropy in SGX enclaves (recall that SGX restricts enclave address space to only 36 bits) and thus can be easily broken [70, 72]. Concurrent and independent from our work, SGX-Shield investigated the use of fine-grained ASR in the context of small enclaves [68].

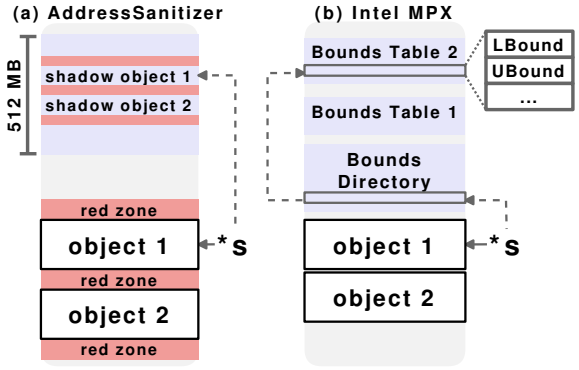


Figure 3: Memory protection mechanisms.

Most of the listed approaches do not prevent information leaks. The only exceptions are Software Fault Isolation (SFI) [39, 79], Data Space Randomization (DSR) [24, 28] and memory-safety techniques [17, 20, 35, 56, 58, 59, 67, 82]. Unfortunately, SFI requires manual separation of the enclave address space into fault domains and is too coarse-grained to guarantee high security (nevertheless, our preliminary evaluation using Intel MPX instructions indicates overheads of 3%, making it a viable low-cost alternative). DSR techniques rely on a simple XOR mask to obfuscate data, and a determined attacker can infer these masks by analyzing leaked data.

Therefore, we concentrate on memory-safety approaches proved to completely prevent data leaks and other attacks [74]. These approaches prevent the very first step in any attack—exploiting a vulnerability, such as overflowing a buffer or freeing an already freed object. We must note that even though we concentrate on memory safety, there are other, insider attack vectors (orthogonal to our work) where a malicious OS tries to deceive the shielded application [30, 60, 71, 83].

2.2 Memory Safety

The foundation of all memory attacks is getting access to a prohibited region of memory [54, 77]. Hence, memory safety can be achieved by enforcing a single invariant: memory accesses must always stay within the bounds of originally intended (referent) objects. For legacy applications written in C/C++, this invariant is enforced by changing (*hardening*) the application to perform additional bounds checks.

A number of memory-safety approaches have been implemented either in software [17, 38, 58, 59, 69] or in hardware [9, 10, 47, 81]. We analyze two open-source and stable approaches in order to put our own results into perspective: software-based AddressSanitizer and hardware-based Intel MPX.

AddressSanitizer is an extension to GCC and Clang/LLVM that detects the majority of object bounds violations [69]. It keeps track of all objects, including globals, heap, and stack variables, and checks whether the address is within one of the used objects on each memory access. For that, it utilizes

(a) Original

```
1  int *s[N], *d[N]
2
3
4  for (i=0; i<M; i++):
5    si = s + i
6    di = d + i
7
8
9
10   val = load si
11
12
13
14   store val, di
15
```

(b) AddressSanitizer

```
1  int *s[N], *d[N]
2  init_shadow(s, N)
3  init_shadow(d, N)
4  for (i=0; i<M; i++):
5    si = s + i
6    di = d + i
7    ssi = get_shadow(si)
8    if *ssi != 0:
9      crash(si)
10   val = load si
11   sdi = get_shadow(di)
12   if *sdi != 0:
13     crash(di)
14   store val, di
15
```

(c) Intel MPX

```
1  int *s[N], *d[N]
2  sbnd = bnd_create s
3  dbnd = bnd_create d
4  for (i=0; i<M; i++):
5    si = s + i
6    di = d + i
7
8    if bnd_check si, sbnd:
9      crash(si)
10   val = load si
11   val_bnd = bnd_load si
12   if bnd_check di, dbnd:
13     crash(di)
14   store val, di
15   bnd_store val_bnd, di
```

(d) SGXBounds

```
1  int *s[N], *d[N]
2  s = specify_bounds(s, s + N)
3  d = specify_bounds(d, d + N)
4  for (i=0; i<M; i++):
5    si = s + i
6    di = d + i
7    sp, sLB, sUB = extract(si)
8    if bounds_violated(sp, sLB, sUB):
9      crash(si)
10   val = load si
11   dp, dLB, dUB = extract(di)
12   if bounds_violated(dp, dLB, dUB):
13     crash(di)
14   store val, di
15
```

Figure 4: Memory safety enforcement of original code in (a) via: (b) AddressSanitizer, (c) Intel MPX, and (d) SGXBounds.

shadow memory – a separate memory region that stores metadata about main memory of an application (shown in pale-blue in Figure 3a). In particular, shadow memory shows which regions are allocated and used (i.e., safe to access) and which are not. AddressSanitizer does that by allocating *redzones* around all main memory objects and marking them inaccessible in the shadow memory. Hence, if an application tries to read or write out of object limits, this can be detected by checking the corresponding shadow address. On top of that, AddressSanitizer provides a quarantine zone for freed objects, thereby detecting temporal errors such as use-after-free and double free.

Execution of the hardened program is supported by a run-time library that initializes the shadow region and replaces memory management functions. It redefines memory-allocation functions (e.g., `malloc`) to allocate redzones and mark them unaddressable (poisoned) in shadow memory and memory-deallocation functions (e.g., `free`) to poison the whole object after it has been freed. The library also maps 1/8th of virtual address space for the shadow memory at startup.

The hardening is performed by a compile-time instrumentation pass. To understand how it works, consider an example in Figure 4a, which copies elements of one array ("s" for source) to another ("d" for destination). The first task of the pass is to set metadata for global, heap, and stack variables. In this example, it creates shadow objects for both arrays and sets the redzones by calling `init_shadow` (Figure 4b, lines 2–3). The pass also enforces the memory access correctness by computing the shadow addresses of all pointers (lines 7 and 11) and checking if they are within a redzone (lines 8 and 12). If a violation is detected, the application is crashed with a debugging message (lines 9 and 13).

Intel MPX is a recent set of ISA extensions of Intel x86-64 architecture for memory protection [9]. By design, Intel MPX detects all possible spatial memory vulnerabilities including intra-object ones (when one member in a structure corrupts other members). The approach to achieving this goal is different from AddressSanitizer—instead of separating objects by unaddressable redzones, Intel MPX keeps *bounds metadata* of all pointers and checks against these bounds on each memory access. Since metadata bookkeeping and checking is implemented partly in hardware, such protection is supposed to be highly efficient.

From the developer perspective, Intel MPX adds new 128-bit registers for keeping upper and lower addresses (bounds) of a referent object. It also provides instructions to check if a pointer is within these bounds, along with instructions to manipulate them. To illustrate how Intel MPX works in practice, consider an example in Figure 4c. After the objects are created (line 1), their bounds have to be stored for future checks (lines 2–3). Then, on each memory access, we check if the accessed address is within the bounds of the referent object (lines 8 and 12) and crash if the check fails (lines 9 and 13). Unlike AddressSanitizer, we have to copy not only the arrays' elements but also their bounds (lines 11 and 15), which causes additional performance overhead. Note that this copying of bounds is required because the elements of arrays are pointers themselves.

One major limitation of the current Intel MPX implementation is a small number of bounds registers. If an application contains many distinct pointers, it will cause frequent loads and stores of bounds in memory. To make this interaction more efficient, bounds are stored in tables with an index derived from the pointer address, similar to a two-level page table structure in x86: a 2GB intermediate table (Bounds Directory) is used as a mediator to the actual 4MB-sized Bounds Tables, which are allocated on-demand by the OS when bounds are created (see Figure 3b). Thus, the constant memory overhead is minimal and the total overhead depends mainly on the number of pointers in the application.

Other memory-safety approaches. Apart from AddressSanitizer and Intel MPX, relevant memory-safety approaches include Baggy Bounds [17] and Low Fat Pointers [37, 38].

Baggy Bounds [17] solves the problem of high memory consumption and broken cache locality by enforcing allocation bounds via buddy allocator. Thus, all objects become power-of-two aligned, allowing simple and efficient checks against the base and bounds. The approach maintains minimal metadata for the bounds table, and the authors introduce tagged pointers with 5 bits holding the size. However, even with tagged pointers Baggy Bounds incurs perceivable overheads: 70% performance and 12% memory (on SPECINT 2000) [17].

Low Fat Pointers [37, 38] are conceptually similar to Baggy Bounds: they also introduce a special allocator that divides the virtual address space in regions of fixed sizes and derive base and bounds from the unmodified pointer. Overheads are also comparable to Baggy Bounds: 54% performance and 12% memory [38]. Yet, to support sparse memory regions, Low Fat Pointers assume a complete 64-bit address space, incompatible with the current version of SGX. Also, the prototype of Low Fat Pointers protects only stack and heap but not globals.

Given their tagged-based nature and low memory consumption, Baggy Bounds and Low Fat Pointers seem proper candidates for usage in SGX enclaves. Unfortunately, neither of them are publicly available.

2.3 Memory Safety for Shielded Execution

Now that we have covered the necessary background, we explain the overheads for the SQLite case study introduced in §1.

In the normal environment—outside of the SGX enclave—Intel MPX exhibits performance overheads of up to $2.5\times$ and AddressSanitizer of up to $2.1\times$ (not shown in Figure 1). These are reasonable overheads expected from these approaches.

Inside the enclave the picture changes dramatically (Figure 1). Intel MPX crashes due to insufficient memory even on tiny input sizes. The cause for this behavior is the amount of bounds tables created to support pointer metadata ($800-900$ tables each 4MB in size), leading to memory exhaustion. We should note however that SQLite is a worst-case example for MPX since it is exceptionally pointer-intensive; pointerless programs, e.g., those using flat arrays, perform significantly better under MPX (see §6).

AddressSanitizer performs up to $3.1\times$ slower than the native SGX execution on bigger inputs. Performance deteriorates mainly due to the EPC thrashing caused by additional metadata accesses to shadow memory. Moreover, AddressSanitizer also has a constant memory overhead of 512MB for shadow memory plus some overhead for redzones around objects. This can lead to situations when the application prematurely suffers from insufficient memory.

For the same experiment, SGXBOUNDS shows performance comparable to native SGX ($30-35\%$ slower) with almost no memory overhead. This motivates our case for a specialized memory safety approach for shielded execution.

3. SGXBOUNDS

We built SGXBOUNDS based on the following three insights. First, as shown in §2.1, shielded application memory (more specifically, its working set) must be kept minimal due to the very limited EPC size in current SGX implementations. This is in sharp contrast to the usual assumption of almost endless reserves of RAM for many other memory-safety approaches [9, 17, 23, 38, 51, 56, 69]. Second, applications spend a considerable amount of time iterating through the elements of an array [32], and a smartly chosen layout of metadata can significantly reduce the overhead of bounds checking. Third, we rely on the SCONE infrastructure [19] with its monolithic build process: all application code is statically linked without external dependencies, which removes the requirements for compatibility and modularity. The first and second insights dictate the use of per-object metadata combined with *tagged pointers* [17, 26] to keep memory overhead minimal, and thanks to the monolithic application assumption, SGXBOUNDS avoids problems of interoperability with uninstrumented code [74].

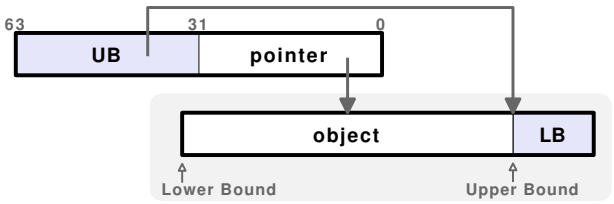


Figure 5: Tagged pointer representation in SGXBOUNDS.

3.1 Design Overview

All modern SGX CPUs operate in a 64-bit mode, meaning that all pointers are 64 bits in size. In SGX enclaves, however, only 36 bits of virtual address space are currently addressable [8], and even this amount of space is not likely to be used due to performance penalties. Thus, SGXBOUNDS relies on the idea of *tagged pointers*: a 64-bit pointer contains the pointer itself in its lower 32 bits and the referent object's upper bound in the upper 32 bits (Figure 5). Note that with SCONE, all application code and data are stored inside the enclave address space and thus all addressable memory is confined to 32 bits and all original pointers can be replaced by their tagged counterparts.

The value stored in the higher 32 bits (UB) serves not only for the upper-bound check, but also as a pointer to the object's other metadata (lower bound or LB). The metadata is stored right after the referent object.

This metadata layout has important benefits: (1) it minimizes amount of memory for metadata, (2) it requires *no* additional memory accesses while iterating over arrays with a positive increment, and (3) it alleviates problems of fat pointers concerning multithreading and memory layout changes (see §4.1).

Figure 4d shows how SGXBOUNDS instruments memory accesses. First, global arrays `s` and `d` are initialized with their respective bounds, and `s` and `d` pointers are transformed into tagged pointers (lines 2–3). For the sake of clarity, we show pointer increments on lines 5–6 uninstrumented (details are in §3.2). Next, before the first memory access at line 10, SGXBOUNDS inserts a bounds check. For this, the original pointer value and its upper bound are extracted from the tagged `si` as well as the lower bound, and the bounds check is performed (lines 7–9). The second memory access (line 14) is instrumented in the same way.

Looking at Figure 4, we can highlight the differences between SGXBOUNDS, AddressSanitizer and Intel MPX. Unlike AddressSanitizer, SGXBOUNDS does not rely on a vast amount of shadow memory, allocating only 4 additional bytes per object. Also, AddressSanitizer requires adjacent objects to

be separated by fixed-size unaddressable redzones and checks whether the memory access lands on one of these redzones. In contrast, SGXBOUNDS extracts pointer bounds and compares the current value of the pointer against them—similar to Intel MPX. But unlike Intel MPX, SGXBOUNDS does not maintain a bounds table and does not explicitly associate each pointer with its own bounds metadata: the newly created pointer implicitly inherits all associated metadata.

3.2 Design Details

Pointer creation. Whenever an object is created, SGXBOUNDS associates a pointer with the bounds of this object.

For global and stack-allocated variables, we change their memory layout so they are padded with 4 bytes and initialize them at run-time. More specifically, we wrap such variables in two-member structures, e.g., `int x` is transformed into `struct xwrap {int x; void* LB}` (similar to [69]). At program initialization, we set the lower and upper bounds with `specify_bounds(&xwrap, &xwrap.LB)`:

```
void* specify_bounds(void *p, void *UB):
    LBaddr = UB
    *LBaddr = p
    tagged = (UB << 32) | p
    return tagged
```

For dynamically allocated variables, SGXBOUNDS wraps memory-management functions such as `malloc`, `calloc`, etc. to append 4 bytes to each newly created object, initialize these bytes with the lower-bound value, and make the pointer tagged with the upper bound:

```
void* malloc(int size):
    void *p = malloc_real(size + 4)
    return specify_bounds(p, p + size)
```

Note that there is no need to instrument `free` as the 4 bytes of metadata are removed together with the object itself.

Lastly, a pointer can be assigned a value of another pointer. If we would use fat pointers or pointers with disjoint metadata, we would need to instrument such pointer assignments, as in Intel MPX (see Figure 4c). However, in SGXBOUNDS no instrumentation is needed, since the newly assigned pointer will also inherit the upper bound and thus all associated object metadata.

Run-time bounds checks. SGXBOUNDS inserts run-time bounds checks before each memory access: loads, stores, and atomic operations (we revise this statement in §4.4). For this, first the original pointer and the upper and lower bounds are extracted. To extract the original pointer, it is enough to use only the lower 32 bits:

```
void* extract_p(void* tagged):
    return tagged & 0xFFFFFFFF
```

Similarly, to extract the upper bound, the higher 32 bits of the tagged pointer must be extracted:

```
void* extract_UB(void* tagged):
    return tagged >> 32
```

If a check against a lower bound is also required then this bound is read from the memory at the upper-bound's address:

```
void* extract_LB(void* UB):
    return *UB
```

Finally, SGXBOUNDS adds the bounds check which crashes the application in case the bounds are violated (in the implementation, we take into account the size of the accessed memory while checking against the upper bound; here we omit it for clarity):

```
bool bounds_violated(void* p, void* LB, void* UB):
    if (p < LB or p >= UB):
        return true
```

Pointer arithmetic. There is a subtle issue with tagged pointers when it comes to pointer arithmetic. Take, for example, increment of a pointer as shown in Figure 4d, lines 5–6. In the ordinary case, pointer arithmetic affects only the lower 32 bits of a tagged pointer. However, it is possible that a malicious/buggy integer value overflows 32 bits and changes the upper bound bits. In this case, the attacker can manipulate the upper bound value and bypass the bounds check. To prevent such corner cases, SGXBOUNDS instruments pointer arithmetic so that only 32 low bits are affected:

```
UB = extract_UB(si)
si = s + i
si = (UB << 32) | extract_p(si)
```

Type casts. Pointer-to-integer and integer-to-pointer casts are a curse for fat/tagged pointer approaches. Some techniques break applications with such casts [17, 44], others suffer from worse performance or lower security guarantees [9, 26, 56]. Unfortunately, arbitrary casts are common in real-world [32].

SGXBOUNDS proved itself immune to arbitrary type casts. It does not perform any instrumentation on type casts and survives integer-to-pointer casts by design. Indeed, when a tagged pointer is casted to an integer, the integer inherits the upper bound. Unless the integer deliberately alters its high 32 bits, the upper bound will stay untouched and the later cast back to a pointer will preserve this bound.

Function calls. SGXBOUNDS does not need to instrument function calls or alter calling conventions. Unlike other approaches [9, 17, 26, 36, 56], SGXBOUNDS is not required to interoperate with possibly uninstrumented, legacy code: the only uninstrumented code is the standard C library (libc) for which we provide wrappers. This implies that any tagged pointer passed as a function argument will be treated as a tagged pointer in the callee. In other words, bounds metadata travels across function and library boundaries together with the tagged pointer.

As already mentioned, we leave libc uninstrumented and introduce manually written wrappers for all libc functions, similar to other approaches [9, 17, 56, 69]. Most wrappers follow a simple pattern of extracting original pointers from the tagged function arguments, checking them against bounds, and calling a real libc function. Others require tracking and extracting the pointers on-the-fly (e.g., the `printf` family), writing proxies for callbacks (`qsort`), or iterating through complex objects (`scandir`).

4. Advanced Features of SGXBOUNDS

4.1 Multithreading support

Bounds checking approaches usually hamper multithreaded applications. AddressSanitizer does not require any specific treatment of multithreading, but, as we illustrate in §6.4, it can negatively affect cache locality if a multithreaded application was specifically designed as cache-friendly (recall that AddressSanitizer inserts redzones around objects). On the other hand, current implementations of Intel MPX instrumentation may suffer from false positives and false negatives in multithreaded environments, introducing a possibility of false alarms or, even worse, of undetected attacks [32, 61].

In fact, all fat-pointer or disjoint-metadata techniques similar to Intel MPX suffer from multithreading issues [32, 57]. An update of a pointer and its associated metadata must be implemented as one atomic operation which requires some synchronization mechanism. This inevitably hampers performance as this is necessary for each pointer/metadata update.

For example, in Figure 4c, lines 10–11, the pointer `val` and its bounds metadata `val_bnd` are copied to `di`. After the first thread loaded `val` on line 10, the second thread can jump in and change `val` to point to some other object. This will also change `val_bnd`. Next, the first thread continues its execution and loads the wrong `val_bnd` on line 11. Now `val` and `val_bnd` do not match, which might result in a false positive. This is a realistic failure scenario for current implementations of Intel MPX since it does not enforce atomicity of metadata updates.[2]

SGXBOUNDS does not experience this problem. Indeed, the pointer and the upper bound are always updated atomically since they are stored in the same 64-bit tagged pointer. Additionally, the lower bound is written only once (at object creation) and is read-only for the whole object's lifetime.

4.2 Tolerating Bugs with Boundless Memory

Up to this point, we assumed that an application crashes with a diagnostic error whenever SGXBOUNDS detects an out-of-bounds access. This fail-fast strategy is simple and prevents hijacks and data leaks, but lowers availability of the system. Even in benign cases of off-by-one buffer overflows, the whole application is crashed and must be restarted.

To allow applications to survive most bugs and attacks and continue correct execution, SGXBOUNDS reverts to failure-oblivious computing [65] by using the concept of boundless memory blocks [64]. In this case, whenever an out-of-bounds memory access is detected, SGXBOUNDS redirects this access to a separate "overlay" memory area to prevent corruption of the adjacent objects, creating the illusion of "boundless" memory allocated for the object (see Figure 6).

This overlay area is implemented as a bounded least-recently-used (LRU) cache—a hash table that maps out-of-bounds memory addresses to spare chunks of memory (similar

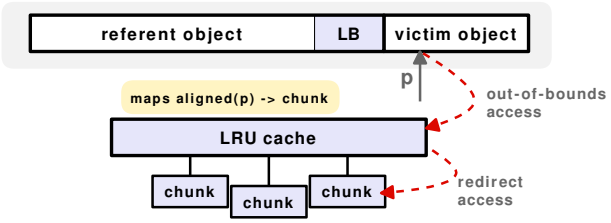


Figure 6: Boundless memory blocks for SGXBOUNDS.

to [64]). These chunks are allocated on-demand, each being 1KB in size. The whole LRU cache is bounded, i.e., it cannot grow more than a certain predefined size (in our implementation, 1MB). This is required to prevent bugs and attacks that span gigabytes of out-of-bounds memory—a frequent consequence of integer overflows due to negative buffer sizes.

Consider an example of a classical off-by-one bug from Figure 4d. If `M` is greater than `N` by one, the last iteration of the loop will trigger bound violations on lines 8 and 12.

With boundless memory feature enabled, SGXBOUNDS consults the LRU cache and redirects the load from `si` (line 10) to a load from an overlay address that corresponds to `si`. If there is no hit for `si` in the cache, SGXBOUNDS falls back on a failure-oblivious approach and simply returns zero values.

Additionally, SGXBOUNDS redirects the store to `di` (line 14) to a corresponding overlay address. If there is no overlay address in the LRU cache, then a new chunk of overlay memory is allocated and is associated with this address. If there is no space left for a new chunk in the LRU cache, the least recently used chunk is evicted (freed) and the new chunk is added instead.

4.3 Metadata Management Support

So far, we discussed only one metadata type kept per object—the lower bound (see Figure 5). However, our memory layout allows us to add arbitrary number of metadata items for each object to implement additional functionality.

All instrumentation in SGXBOUNDS is implemented as calls to auxiliary functions described in §3.2, which we refer to as instrumentation hooks. One can think of these hooks as a metadata management API (see Table 2). The API consists of only three functions: (1) `on_create` is called at run-time whenever a new object is created, either a global variable during program initialization or a local variable during stack frame creation or a dynamically allocated variable, e.g., via `malloc`. In the context of SGXBOUNDS, it corresponds to the `specify_bounds` function which initializes our only metadata (lower bound). (2) `on_access` is called at each memory access, be it a write, read, or both (for atomic instructions such as compare-and-swap). In SGXBOUNDS, the hook roughly corresponds to the `bounds_violated` function. (3) `on_delete` is called whenever an object is deallocated; we support this hook only for heap-allocated objects, since global variables are never deleted and there is no way to track deallocation of variables on stack. SGXBOUNDS does not use this hook because we do not focus on temporal safety (also note that the metadata is removed automatically with the object).

[2] We demonstrate how multithreaded code fails in MPX and discuss this and other issues in more detail in our technical report on Intel MPX [61].

`on_create(objbase, objsize, objtype)` called after object creation (globals, heap, or stack)
`on_access(address, size, metadata, accesstype)` called before memory access (read, write, or read-write)
`on_delete(metadata)` called before object destruction (only for heap)

Table 2: SGXBOUNDS metadata management APIs.

With this API, it is straightforward to implement additional functionality. For example, SGXBOUNDS can be expanded to probabilistically protect against double-free bugs using an additional metadata item acting as a "magic number" to compare with. Another example would be providing debug information about where a detected out-of-bounds access originates from.

4.4 Optimizations

Safe memory accesses. Many pointer arithmetic operations and memory accesses are always-safe. For example, the calculation of the member's offset in a structure is guaranteed to be in-bounds and never overflows 32 low bits. The memory access at a predefined index in a fixed-size array is also safe.

In these cases, there is no need for instrumentation of pointer arithmetic or bounds checks on memory accesses. We employ the built-in compiler analysis to detect all safe cases and do not instrument them. This is a standard optimization for many approaches [17, 36, 69] and yields significant performance gains for some applications, up to 20% (§6.5).

Hoisting checks out of loops. Many programs spend a lot of time iterating over arrays in simple loops. The array-copy example in Figure 4a is a good illustration.

The straightforward instrumentation with SGXBOUNDS, as depicted in Figure 4d, inserts bounds checks before each memory access (on lines 7–9 and 11–13). It is immediately obvious from the code that the lower-bound check is useless: `si` and `di` start from the base addresses of the corresponding arrays and increment on each iteration. Thus, it is safe to remove the check against the lower bound, which renders the extraction of the lower bound redundant. In the end, this optimization can save two memory accesses per iteration (to extract LBs).

The upper-bound check cannot be removed: in general case the value of `M` is unknown and can exceed the upper bound of the two arrays (`N`). But it is sufficient to perform only one check for each array outside of the loop, namely, the check of `s+M` and of `d+M` against their respective upper bounds.

Such optimization is applied only for loops with small increments (up to 1,024 bytes) – which is virtually all loops encountered in regular applications. We mark the last 4K page of an enclave as unaddressable, which protects from integer over- and underflows of the loop counter variable. These simple precautions protect against overflowing pointer arithmetic inside loops when lower- or upper-bound checks are hoisted out.

To perform these optimizations, we reused classical scalar evolution analysis. We observed performance gains of up to 22% in some cases (§6.5).

5. Implementation

5.1 SGXBOUNDS Implementation

SGXBOUNDS is a compile-time transformation pass implemented in LLVM 3.8. For greater modularity, we implement the functionality outlined in §3.2 as always-inlined functions in a separate C file. The pass inserts calls to these functions during instrumentation. We refer to this set of auxiliary C functions as the run-time for SGXBOUNDS.

We do not alter the usual build process of an application, but rather use the Link-Time Optimization feature of LLVM.

Compiler support. SGXBOUNDS compiler pass works under LLVM 3.8 [49] and was implemented in 951 lines of code (LOC). Its functionality closely follows the description in §3.

We treat inline assembly as an opaque memory instruction: all pointer arguments to inline assembly are bounds checked. To minimize the risk of misbehaving assembly, we disabled inline assembly in all tested applications which had such a flag.

To support C++, we opted to instrument the whole C++ standard library. We used libcxx (libc++) implementation for this purpose. SGXBOUNDS does not yet completely support C++ exception handling: it runs C++ applications correctly only if they do not throw exceptions at run-time.

Run-time support. Next we describe implementation details of the SGXBOUNDS auxiliary functionality. The complete implementation of the run-time functions spans 320 LOC, and the libc wrappers contain 4289 LOC.

We implemented boundless memory feature (§4.2) completely in the run-time support library in 68 LOC. It is based on uthash lists which we extend to a simple LRU cache [15]. To prevent data races, all read/update operations on the cache are synchronized via a global lock. Such implementation is slow, but since it is triggered on supposedly rare events of out-of-bounds memory accesses (and thus it lies on a slow path), we can ignore this possible performance bottleneck.

Furthermore, SGXBOUNDS does not fall back to a failure oblivious approach for libc function wrappers, but rather returns an error code through `errno` where applicable (e.g., `EINVAL` for the `recv` function). This allows applications to quickly drop offending requests.

For the tagged pointer scheme, SGXBOUNDS relies on SGX enclaves (and thus the virtual address space) to start from 0x0. To allow this, we set the Linux security flag `vm.mmap_min_addr` to zero for our applications. We also modified the original Intel SGX driver (5 LOC) to always start the enclave at address 0x0.

5.2 AddressSanitizer, Intel MPX, and SGX Enclaves

To integrate AddressSanitizer and Intel MPX into SGX enclaves, we had to solve three main issues. (1) SCONE disallows dynamic linking against shared libraries, so AddressSanitizer and Intel MPX must be compiled statically into the application. (2) The virtual address space is restricted to 32 bits. (3) The OS is not allowed to peek into the address space of the enclave.

Adapting AddressSanitizer for SGX enclaves. We had to solve issues (1) and (2) for AddressSanitizer. First, the current implementation of AddressSanitizer relies on libc being dynamically linked at application start-up (the usual function interposition scheme). Trying to statically link libc into the application would result in a compilation error due to multiple definitions of the same function.

Every function in SCONE libc has an alias (a second name which is used to denote the real function). We modified the interception layer of AddressSanitizer such that its wrapper functions call aliases (real libc functions), therefore solving the problem of multiple definitions. This is similar to SGXBOUNDS (see `malloc` in §3.2).

Second, by default AddressSanitizer is compiled in 64-bit mode and reserves $\sim$16TB of memory for its shadow space. Fortunately, it also has a 32-bit mode where only 512MB of memory is carved for shadowing. We changed the build system of AddressSanitizer to always use the 32-bit mode. Also, we disabled "leak detection" flag that broke SCONE.

Adapting Intel MPX for SGX enclaves. To put Intel MPX inside SGX, we solved issues (2) and (3). Intel MPX operates in the 64-bit mode, and this affects its address translation to store and load bounds (Figure 9 in [9]). In the 64-bit mode, Intel MPX allocates a 2GB Bounds Directory (BD) table at start-up and 4MB-sized Bounds Tables (BT) on-demand.

We discovered that this address translation also works with 32-bit addresses. In the 32-bit address case, only 12 bits are used for indexing in the BD table, and the rest for BT tables. Thus, we were able to restrict the size of BD to 32KB by changing the corresponding constants in the MPX compiler pass and run-time libraries. We did not change the address translation logic of BT allocation.

For issue (3), we had to move the kernel logic into the SGX enclave. In the normal case, on-demand allocation of BTs requires support from the Linux kernel. Whenever an application fires a "bounds store" exception (meaning the application needs to allocate a new BT to store some pointer metadata), the kernel handles it: it examines the pointer address that raised the exception, calculates the correct BT, and allocates it on behalf of the application. Then the execution of the application continues, and the metadata is stored in the newly allocated BT.

This kernel-application cooperation is impossible in SGX. The kernel cannot examine the failing pointer and cannot peek into or modify memory inside the SGX enclave. To alleviate this problem, we moved all the BT-allocation logic from the kernel into the Intel MPX run-time library. We also instructed the kernel not to try to cooperate with the application, but only to forward the exception to the application itself. At this point the enclave takes control and handles the exception. Note that this logic does not compromise security because SGX double-checks the exceptions forwarded by the kernel. Our adaptation also does not influence performance since BT-allocation is a rare event, and the kernel-to-application forwarding adds negligible overhead.

6. Evaluation

Our evaluation answers the following questions:

- What are the performance and memory overheads of SGXBOUNDS and how do they compare to AddressSanitizer and Intel MPX? (§6.2)
- How does the increasing working set affect the performance of SGXBOUNDS? (§6.3)
- How does multithreading affect the performance? (§6.4)
- How effective are the optimizations in improving the performance? (§6.5)
- What level of security is achieved by SGXBOUNDS according to the RIPE benchmark? (§6.6)
- How does the performance of SGXBOUNDS change outside of SGX enclaves? (§6.7)

6.1 Experimental Setup

Applications. We evaluated SGXBOUNDS using Fex [62] with applications from two multithreaded benchmark suites: Phoenix 2.0 [63] and PARSEC 3.0 [25], as well single-threaded SPEC CPU2006 [43]. We report results for all 7 applications in the Phoenix benchmark, 9 out of 13 applications in PARSEC, and 13 out of 19 in SPEC. The remaining applications are not supported for the following reasons: *raytrace* depends on the dynamic X Window System libraries not shipped together with the benchmark; *freqmine* is based on OpenMP, *facesim* and *canneal* fail to compile under SCONE due to position-independent code issues, *dealII*, *omnetpp*, and *povray* fail due to incomplete support of C++, *perlbench* triggered an unsupported corner case of a specific loop optimization, and *gcc* and *soplex* violate C memory model and cannot be protected via bounds-checking [61].

Methodology. In all experiments (except §6.3) the numbers are normalized against the native SGX version, i.e., a version compiled under the SCONE infrastructure and not instrumented with any memory-safety techniques. For all measurements, we report the average over 10 runs and geometric mean for the "gmean" across benchmarks. For memory measurements, since the Linux kernel does not provide statistics on the Resident Set Size inside SGX enclaves, we show the maximum amount of reserved virtual memory.

Testbed. We used the largest available datasets provided by Phoenix, PARSEC, and SPEC benchmark suites. The experiments were carried out on a machine with a 4-core (8 hyper-threads) Intel Xeon processor operating at 3.6 GHz (Skylake μarchitecture) with 64GB of RAM, a 1TB SATA-based SDD, and running Linux kernel 4.4. Each core has private 32KB L1 and 256KB L2 caches, and all cores share a 8MB L3 cache.

Compilers. We used LLVM 3.8 for native SGX, AddressSanitizer, and SGXBOUNDS versions and gcc 5.3 for the Intel MPX version. We use default options for AddressSanitizer but disable leak detection (see §5.2). We also disable "narrowing of bounds" feature in Intel MPX to remove false positives in some programs.

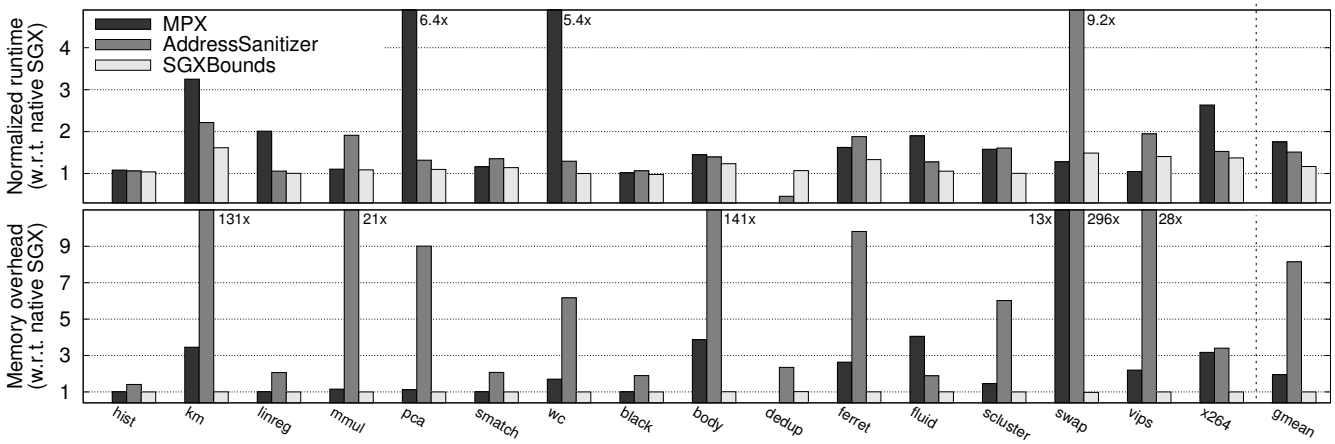


Figure 7: Performance (top) and memory (bottom) overheads over native SGX execution (with 8 threads).

6.2 Performance and Memory Overheads

Figure 7 shows performance and memory overheads of Intel MPX, AddressSanitizer, and SGXBOUNDS normalized against the uninstrumented SGX version. All benchmarks were run with 8 threads to fully utilize our machine.

Performance overheads of Intel MPX significantly vary across benchmarks, reaching up to $5-6\times$ in some cases. For example, consider *pca*. Its working set is 70MB (77MB for Intel MPX due to additional metadata), thus all data fits into EPC and performance is dominated by the on-die characteristics like CPU cache accesses and number of retired instructions. Indeed, Intel MPX leads to $10\times$ more instructions, $5\times$ more branches, and $25\times$ more L1 cache accesses (*pca* operates on a large array-of-pointers and is thus pointer-intensive). Together, this leads to an overhead of $6.3\times$. On the other hand, pointer-free benchmarks like *histogram* and *blackscholes* exhibit almost zero overhead (observe that memory overheads in these cases are also close to zero).

Memory overheads of Intel MPX also vary. For benchmarks working with large arrays and/or using no pointer-based structures (almost all Phoenix benchmarks), pointer bounds metadata occupies relatively small amount of space and overheads are negligible. However, for pointer-intensive cases like *bodytrack* and *fluidanimation*, Intel MPX allocates a lot of metadata, leading to $\sim 4\times$ memory overhead. In degenerate cases, overheads can reach up to $13\times$ (*swaptions*) or even crash the application (*dedup*, note the missing MPX bar).

AddressSanitizer has more reasonable and expected performance overhead of around 51%.[3] The *kmeans* benchmark has one of the highest overheads of $2.2\times$. Since the working set of *kmeans* is only 5MB (AddressSanitizer blows it up to 643MB but does not use most of it), the overhead is dominated by the CPU instructions and cache: $2.4\times$ more instructions, $2.6\times$ more branches, and $2.2\times$ more L1 cache accesses.

In terms of memory usage, AddressSanitizer is a poor choice for SGX enclaves. By reserving 512MB of memory for its shadow space, AddressSanitizer reduces the available memory to 3.5GB (§2.2). Moreover, AddressSanitizer pads objects with redzones and uses so-called "quarantine" which obstructs reuse of memory [69]. All this can lead to memory blow-ups of $50-100\times$.

The most dramatic example of memory overheads is *swaptions*. This benchmark has a working set of only 3.3MB, but it constantly allocates and frees tiny objects. For Intel MPX, it results in a flood of pointers and a constant need for more and more bounds tables (12 BTs or 48MB). For AddressSanitizer with its quarantine feature, the reuse of memory is restricted and new objects are allocated in more and more pages (103,250 pages or 413MB). Note that the excessive amount of metadata does not seriously hamper performance of Intel MPX because the working set still fits into EPC, but AddressSanitizer suffers from EPC thrashing and thus exhibits poor performance.

Finally, SGXBOUNDS performs the best, with an average performance overhead of 17% and average memory overhead of 0.1%. In comparison to Intel MPX, SGXBOUNDS does not choke on pointer-intensive programs (*pca*, *wordcount*, *x264*). In comparison to AddressSanitizer, SGXBOUNDS has much better memory consumption. It also does not exhibit corner-case performance drops like AddressSanitizer in *swaptions* and does not eat up all memory like Intel MPX in *dedup*.

6.3 Experiments with Increasing Working Set

To understand the behavior of different approaches with increasing sizes, we created five input sizes ranging from tiny (XS) to extra-large (XL) for several benchmarks (Figure 8). Note that we normalize against SGXBOUNDS for clarity; SGXBOUNDS itself performs $\sim 15\%$ worse than native SGX and has a maximum deviation of 2.1% across different sizes. We observed different patterns across approaches and benchmarks. In most cases, increasing the size did not influence the overheads of AddressSanitizer and Intel MPX in comparison to SGXBOUNDS, indicating no changes in memory access patterns due to CPU cache or EPC thrashing. Next, we elaborate on the patterns for some other cases.

[3] Except *dedup* which performs better than the baseline SGX version. Our investigation revealed that AddressSanitizer accidentally changes the memory layout of *dedup* such that it has much less LLC cache misses at runtime.

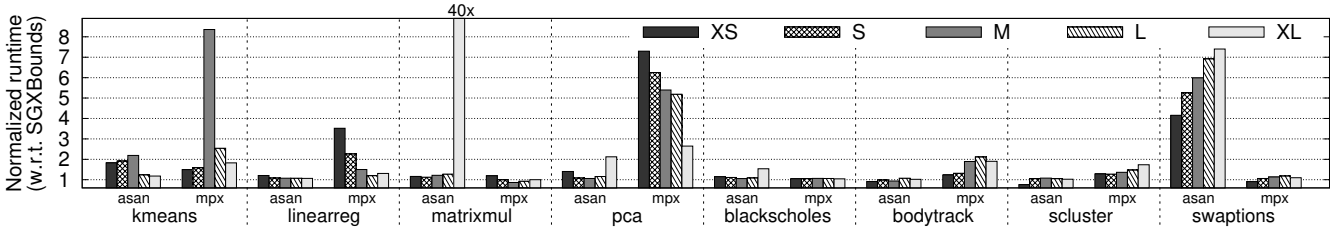


Figure 8: Performance overheads over SGXBOUNDS execution with increasing sizes of working sets (with 8 threads).

Working set (MB)		LLC misses (%) ASan	LLC misses (%) MPX	Page Faults (×) ASan	Page Faults (×) MPX	# of BTs
kmeans						
XS	(17)	5.8	-0.3	3.9	1.2	6
S	(34)	12.4	1.3	3.1	2.0	9
M	(68)	17.2	9.7	3.9	**44**	15
L	(**135**)	19.7	1.3	1.2	2.9	27
XL	(**270**)	11.3	1.5	1.2	1.9	52
matrixmul						
XS	(2)	1.7	1.0	9.4	1.5	1
S	(7)	-0.5	-1.2	5.8	1.4	1
M	(26)	-3.6	-13.8	2.9	1.2	1
L	(103)	125	-11.5	1.9	1.0	1
XL	(**412**)	**4367**	-0.1	1.2	1.0	1

Table 3: Overheads w.r.t. SGXBOUNDS for experiment of increasing working set size. Col. 4–5: page faults due to EPC thrashing. Col. 6: num. of bounds tables allocated in MPX.

Kmeans has the following pattern: the overheads over SGXBOUNDS grow until a certain point ("M"), reach a maximum and then drop. Looking at Table 3, we note that the working set fits completely in EPC at first and then spills out to RAM at large inputs. This means that before the "L" value, overheads are dominated by the on-die characteristics, and after it by the paging mechanism. In the case of *kmeans*—a benchmark which iteratively goes through its working set—the number of page faults explains the spikes and subsequent drops in both Intel MPX and AddressSanitizer.

Note the outlier number of page faults for "M" in Intel MPX: the working set increases to 127MB due to bounds tables. At the same time, the original SGX version and SGXBOUNDS both have the working set of 68MB. Thus, SGXBOUNDS fits completely into EPC while Intel MPX must evict and load-back pages (AddressSanitizer also has a working set that fits into EPC). Since such constant EPC thrashing is expensive (§2.1), performance of Intel MPX becomes 8.3× worse.

On "L" and "XL" sizes, all approaches do not fit into EPC and experience EPC thrashing, and this dominates the performance overheads of all of them. Note how the number of page faults from Table 3 correlates with the overhead in Figure 8.

Matrixmul performs a simple (cache-unfriendly) multiplication of two matrices and writes the result into a third matrix.

Intel MPX performs on par with SGXBOUNDS. Looking at the number of bounds tables allocated (Table 3), we see that only one table was enough for any input size. This is trivially explained by the fact that *matrixmul* requires only three bounds entries—one for each matrix. Moreover, Intel MPX holds these bounds in CPU registers such that there are *no* additional memory accesses and thus no overhead.

Note that *matrixmul* exhibits sequential pattern of memory accesses. This implies that even when the working set does not fit in EPC, there is no EPC thrashing (old EPC pages are evicted and never accessed again) – in other words, page faults do not dominate performance overheads. In this scenario, CPU cache misses play a major role. AddressSanitizer breaks cache locality since it inserts additional accesses to shadow memory. On "XL" size, this effect is exacerbated by matrices not fitting in EPC, leading to 44× more LLC cache misses. This explains the 40× spike in overhead in Figure 8.

6.4 Effect of Multithreading

As discussed in §4.1, SGXBOUNDS supports multithreading by design. To highlight the fact that SGXBOUNDS does not impose additional performance overhead with more threads, we conducted an experiment with one and four threads (Figure 9). Also, the overheads with 8 threads are shown in section 6.2. We compare SGXBOUNDS with AddressSanitizer which also has an efficient support for multithreading. We do not compare against Intel MPX since it lacks real support for multithreading; we believe that future versions of MPX might have deteriorated performance due to synchronization overheads.

On average, overhead of AddressSanitizer increases from 35% with one thread to 49% with four threads while overhead of SGXBOUNDS decreases from 17% to 16%. In most cases however, both SGXBOUNDS and AddressSanitizer do not exhibit any additional overhead. This is reasonable since both approaches do not require additional synchronization primitives and introduce lightweight wrappers around `pthreads`.

However, AddressSanitizer can break (1) memory layout due to redzones around objects, and (2) cache locality due to additional memory accesses to shadow memory. This happens in *matrixmul*: AddressSanitizer worsens cache locality on four threads and has 6.7× more LLC cache misses than SGXBOUNDS. Note that SGXBOUNDS adds only 12 bytes in *matrixmul* (4B for each matrix) which preserves the original memory layout. Thanks to this, SGXBOUNDS performs 70% better than AddressSanitizer on 4 threads. A similar explanation holds true for *swaptions*.

6.5 Effect of Optimizations

We evaluated gains of optimizations as detailed in §4.4. The results are shown in Figure 10. On average, applying all optimizations yields a modest performance improvement of 2%.

Unfortunately, our optimizations are limited in scope. Our implementation relies on Scalar Evolution and SizeOffsetVis-

Figure 9: Performance overheads of AddressSanitizer and SGXBOUNDS over native SGX with different number of threads.

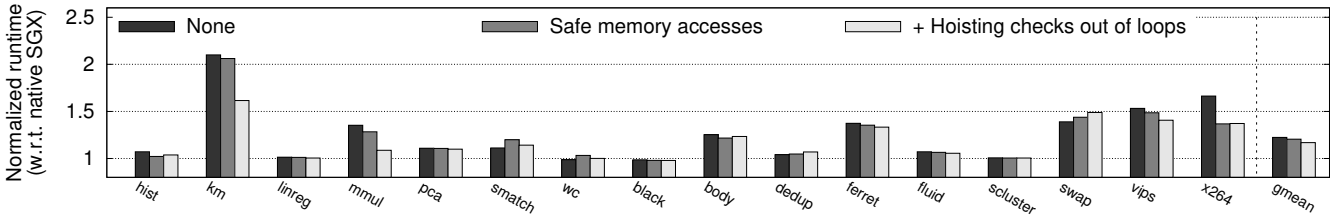


Figure 10: Performance overheads of SGXBOUNDS over native SGX execution with different optimizations (with 8 threads).

Approach	Prevented attacks
MPX	**2/16** (except return-into-libc on heap & data)
AddressSanitizer	**8/16** (except in-struct buffer overflows)
SGXBOUNDS	**8/16** (except in-struct buffer overflows)

Table 4: Results of RIPE security benchmark.

itor LLVM analyses. However, they do not yet support inter-procedural (whole-program) analysis. Therefore, the results turned out to be not as impressive as we originally hoped; we believe that enabling inter-procedural analysis in future implementations could greatly improve performance.

Nonetheless, our optimizations can give significant performance boost in some cases. For example, the hoisting checks optimization is helpful for *kmeans* and *matrixmul*, with performance improvements of up to 20%. Similar gains are seen for *x264* when the safe checks optimization is applied.

6.6 Security Benchmark (RIPE)

To evaluate security guarantees of SGXBOUNDS, we employed the RIPE security benchmark [80]. RIPE claims to perform 850 working buffer-overflow attacks. However, under our native configuration, only 46 attacks were successful: through the shellcode that creates a dummy file and through return-into-libc. When building RIPE under SCONE infrastructure, this number decreased to 16 attacks: the shellcode attacks failed because SGX disallows the `int` instruction used in shellcode.

Table 4 shows the security results of all approaches. Intel MPX could not detect 14 out of these 16 attacks: the two attacks detected were both stack-smashing attacks trying to overwrite an adjacent function pointer. AddressSanitizer detected 8 out of 16 attacks: the remaining 8 attacks were all in-struct buffer overflows, when the same object contained a vulnerable buffer and a target-of-attack function pointer. Finally, SGXBOUNDS showed the exact same results as AddressSanitizer. The in-struct overflows could not be detected because both AddressSanitizer and SGXBOUNDS operate at the granularity of whole objects.

6.7 SPEC CPU2006 Experiments

To facilitate comparison with other approaches, we also report the overheads of SGXBOUNDS over the SPEC CPU2006 benchmark suite. Note that all programs in SPEC are single-threaded and more CPU-intensive than Phoenix and PARSEC, such that the restrictions of SGX have less impact for SPEC. We performed two experiments to measure performance and memory consumption: inside of SGX enclaves (similar to previous evaluation) and outside them (to understand overheads in normal, unconstrained environments).

SGXBOUNDS, being a bounds-checking approach, has false positives in some legitimate programs that implement custom memory management. For example, we could not run *soplex* because it directly updates referent objects of pointers. SGXBOUNDS can also break on programs that manipulate high bits of pointers, e.g., *gcc* contains unions of pointers-ints and manipulates high bits. Note that other approaches have the same problems with these programs, e.g., MPX [61], Baggy Bounds [17], and Low Fat Pointers [47] – they all require manual modifications to misbehaving programs.

Figure 11 shows the results for our in-enclave scenario. In agreement with experiments on Phoenix and PARSEC (see Figure 7), SGXBOUNDS shows the lowest performance and memory overheads on average, 41% and 0.4% respectively. Again, SGXBOUNDS adds negligible overhead in memory consumption which in many cases leads to better cache and EPC locality. Consider *mcf*: AddressSanitizer exhibits performance overhead of $2.4\times$ whereas SGXBOUNDS–only 1%. This is explained by EPC thrashing: AddressSanitizer has $3{,}400\times$ more page faults than both original and SGXBOUNDS versions. Similar explanations hold for other extreme cases such as *milc*, *sjeng*, and *xalanc*.

Intel MPX performed slightly better than AddressSanitizer (52% performance and 110% memory overhead against 76% and $10\times$ respectively) but failed to finish on *astar*, *mcf*, and *xalanc*. Just like in cases of *SQLite* and *dedup*, these programs crash due to insufficient memory for MPX Bounds Tables.

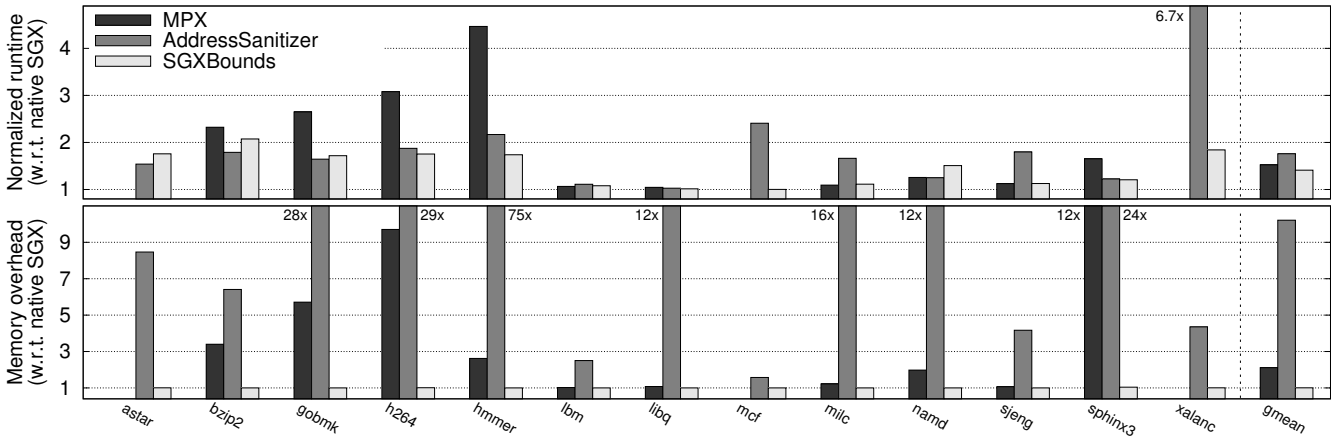


Figure 11: SPEC inside of SGX enclave: Performance (top) and memory (bottom) overheads over native SGX execution.

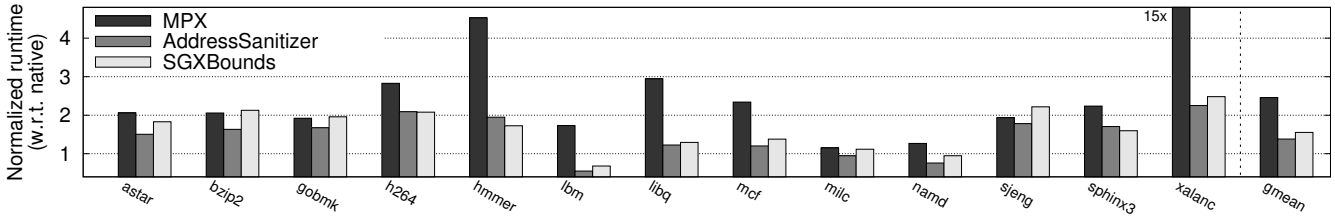


Figure 12: SPEC outside of SGX enclave (normal unconstrained environment): Performance overhead over native execution.

In addition, we show results for outside-enclave, unconstrained environment in Figure 12. As expected, SGXBOUNDS performs not that well outside of enclaves, with a higher average performance overhead (55%) than AddressSanitizer (38%).[4] In unrestricted-memory environments, the benefits of a cache-friendly layout of SGXBOUNDS are effectively wiped out, even though the memory consumption of SGXBOUNDS is only 0.1% in contrast to $2-4\times$ of MPX and AddressSanitizer (not shown on plots). Also, the 55% performance overhead of SGXBOUNDS is comparable to the ones incurred by Baggy Bounds (70%) and Low Fat Pointers (43%)[5]; see also §2.2.

7. Case Studies

In addition to SQLite, we evaluated three other applications. Our evaluation of the case-studies is based on: (1) performance and memory overheads; and (2) security guarantees. All applications were evaluated on the machine described in §6; clients connected via a 10Gb network.

Memcached. We evaluated Memcached v1.4.15 [40] using the memaslap benchmark shipped together with libmemcached v1.0.18 client [11]. Performance and memory overheads are shown in Figure 13. The uninstrumented SGX version performs significantly worse than the native version ($60-75\%$ throughput of native). This is due to the Memcached working set not fitting in the CPU cache; SGX spends some cycles on encrypting and decrypting data leaving the cache as well as checking its integrity. AddressSanitizer performs very close to SGX; even though it introduces additional memory accesses, the original memory latency is already high enough to hide this overhead. The performance of SGXBOUNDS can be explained similarly. Finally, Intel MPX has an abysmal drop in throughput: MPX bounds tables consume so much memory that the working set exceeds the EPC and requires paging (we observed $100\times$ more page faults than for SGXBOUNDS).

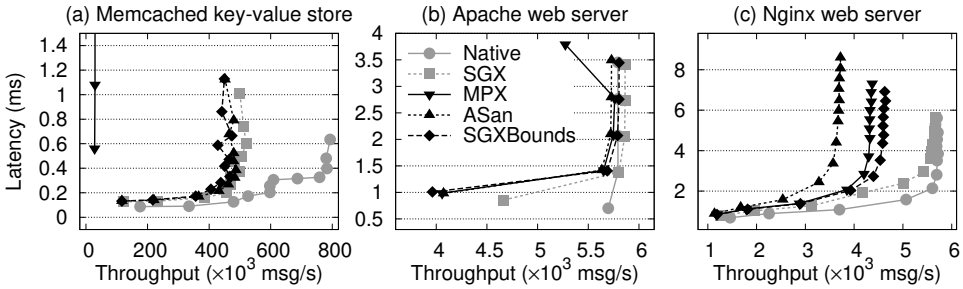
For security evaluation, we reproduced a denial-of-service attack, CVE-2011-4971 vulnerability [12], in the SGX environment. All approaches—AddressSanitizer, Intel MPX, and SGXBOUNDS—detected buffer overflow in the affected function's arguments. AddressSanitizer and Intel MPX halted the program, while SGXBOUNDS with its boundless memory feature discarded the overflowed packet's content but went into an infinite loop due to a subsequent bug in the program's logic.

Apache. We evaluated Apache v2.4.18 [5] with OpenSSL v1.0.1f using the ab benchmark [2]. The performance results are plotted in Figure 13b. The SGX version of Apache performs slightly and consistently better than the native version. We attribute this to the SCONE features of user-level scheduling and asynchronous system calls [19]. Intel MPX quickly deteriorates with more clients; looking at the number of page faults, we conclude that this is due to the increasing overheads of bounds tables. (In Apache, each new client requires around 1MB of memory which bloats the bounds metadata for Intel MPX.) AddressSanitizer performs 2% worse than SGX, and SGXBOUNDS—on par with SGX.

The unexpected 50% increase in memory use for SGXBOUNDS in comparison to SGX is due to the custom memory allocator of Apache. It allocates only page-aligned amounts

[4] *lbm* and *namd* under AddressSanitizer perform better than the native version. This is due to changes in memory layout and similar to *dedup*; also see [61].

[5] For Low Fat Pointers, we took the same subset of 13 programs as in our evaluation and calculated the geomean. For Baggy Bounds, we resorted to specifying the reported mean over SPEC2000.



Memory usage (MB) for peak t-put

	Memcached	Apache	Nginx
SGX	71.6	15.4	0.9
MPX	641	144	37.0
ASan	649	598	893
SGXBnd	71.8	23.2	1.0

Figure 13: Throughput-latency plots and peak memory usage of case studies: (a) Memcached, (b) Apache, and (c) Nginx.

of memory, and the additional 4B of metadata forces our `mmap` wrapper to allocate a whole additional page.

To evaluate security, we looked at the infamous Heartbleed bug [4, 14]. AddressSanitizer, Intel MPX, and SGXBOUNDS all detect Heartbleed attack. Additionally, SGXBOUNDS does not crash the application, but—thanks to boundless memory—copies zeros into the reply message in accordance to the failure-oblivious computing policy. Thus, SGXBOUNDS prevents confidential data leaks, at the same time allowing Apache to continue its execution.

Nginx. We evaluated Nginx v1.4.0 [13] using the ab benchmark. Figure 13c shows performance overheads. The $5-20\%$ difference in throughput between the native version and SGX is due to the overhead of copying the 200KB web page twice, first to the SCONE's syscall thread and then further to the socket. Note that this overhead was hidden by the overhead of thread synchronization in Apache (Apache uses 25 threads while Nginx is single-threaded).

AddressSanitizer performs the worst, achieving only $65-70\%$ throughput of that of SGX. In comparison to Apache, Intel MPX performs better than AddressSanitizer. The reason for this is a smarter memory management policy of Nginx, with as little memory copying as possible [13]. Because of this, Intel MPX does not spill bounds metadata as extensively as in Apache, and gains better performance as a result. Finally, SGXBOUNDS achieves $80-85\%$ throughput of SGX thanks to its efficient metadata scheme.

For security evaluation, the bug under test was a stack buffer overflow CVE-2013-2028 that can be used to launch a ROP attack [3]. All three approaches detect this bug. With SGXBOUNDS boundless memory feature, Nginx can drop the offending request and continue its execution.

8. Discussion and Concluding Remarks

In this work, we presented SGXBOUNDS—a memory-safety approach tailored to the specifics of Intel SGX. We conclude by discussing the limitations of our approach, future work, and peculiarities of SGX and MPX.

EPC Size. SGXBOUNDS mandates the use of a limited 32-bit address space. This is in accordance with current SGX implementations which allow only 36-bit address space. SGXBOUNDS could be refined to allow 36-bit pointers, hinged on the correct alignment of newly allocated objects (which is already provided by compilers and memory allocators).

It is possible that future SGX enclaves will have larger address spaces, decreasing the number of spare bits in pointers and negating the premise of SGXBOUNDS. We believe enclaves spanning more than 4GB of memory are doubtful as they will suffer huge performance penalty. In addition, SGX is best suited for programs with small TCB and working sets.

Limitation of static linking. SGXBOUNDS and the underlying SCONE infrastructure currently require the program to be statically linked. There is a decades-long debate on static vs dynamic linking [18, 66, 76, 78]. We strongly believe that dynamic linking is detrimental for security for a variety of reasons, including `LD_PRELOAD` issues, ldd and linker exploits. In addition, static linking enables powerful whole-program optimizations. Yet, SGXBOUNDS could be used with dynamic libraries, though it would require additional wrapper functions for interoperability with them.

Catching intra-object overflows. SGXBOUNDS keeps bounds for whole objects and therefore cannot detect intra-object overflows (similar to AddressSanitizer). We currently explore the ability to catch such overflows using narrowing of bounds: whenever SGXBOUNDS detects an access through a struct field, it updates the current pointer bounds to the bounds of this field. The main difficulty here is to keep additional lower-bound metadata for each object field; for this, we extend our metadata space and utilize metadata hooks.

Intel MPX. Considering that Intel MPX is a hardware extension, its low performance was surprising to us. Intel MPX performs well if the protected application works only with a small portion of pointers, but in the opposite case the overheads may get very high. To understand the underlying reasons of poor MPX performance, we conducted a more extensive and rigorous evaluation, results of which can be found in [61].

Software availability. Source code is publicly available under `https://github.com/tudinfse/sgxbounds`.

Acknowledgements. We thank our anonymous reviewers and our shepherd Herbert Bos for their helpful comments. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 645011 (SERECA) and the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed).

References

[1] Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO). https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf, 2015. Accessed: February, 2017.

[2] ab—Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2016. Accessed: February, 2017.

[3] Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028). http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html, 2016. Accessed: February, 2017.

[4] Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug. http://www.theregister.co.uk/2014/04/09/heartbleed_explained/, 2016. Accessed: February, 2017.

[5] Apache HTTP server project. http://httpd.apache.org/, 2016. Accessed: February, 2017.

[6] Cloud Computing - SME Survey. https://www.enisa.europa.eu/publications/cloud-computing-sme-survey, 2016. Accessed: February, 2017.

[7] CloudCamp: Five key concerns raised about cloud computing. http://www.itnews.com.au/news/cloudcamp-five-key-concerns-raised-about-cloud-computing-223980, 2016. Accessed: February, 2017.

[8] Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf, 2016. Accessed: February, 2017.

[9] Intel Memory Protection Extensions Enabling Guide (rev. 1.01). https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf, 2016. Accessed: February, 2017.

[10] Introduction to SPARC M7 and Silicon Secured Memory (SSM). https://swisdev.oracle.com/_files/What-Is-SSM.html, 2016. Accessed: February, 2017.

[11] libmemcached. http://libmemcached.org/, 2016. Accessed: February, 2017.

[12] Memcached bug: CVE-2011-4971. http://www.cvedetails.com/cve/cve-2011-4971, 2016. Accessed: February, 2017.

[13] nginx: The Architecture of Open Source Applications. http://www.aosabook.org/en/nginx.html, 2016. Accessed: February, 2017.

[14] The Heartbleed Bug. http://heartbleed.com/, 2016. Accessed: February, 2017.

[15] uthash: Hash Table for C Structures. https://troydhanson.github.io/uthash/, 2016. Accessed: February, 2017.

[16] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2008.

[17] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium (Sec)*, 2009.

[18] Anselm R Garbe. Static Linux. http://sta.li/faq. Accessed: February, 2017.

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

[20] T. M. Austin, S. E. Breach, G. S. Sohi, T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI)*, 1994.

[21] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[22] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[23] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[24] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[25] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009.

[26] M. Brunink, M. Susskraut, and C. Fetzer. Boundless memory allocations for memory safety and high availability. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, 2011.

[27] N. Burow, S. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-Flow Integrity: Precision, security, and performance. *arXiv preprint arXiv:1602.04056*, 2016.

[28] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[29] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[30] S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[31] X. Chen, H. Bos, and C. Giuffrida. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *Proceedings of the European Symposium on Security and Privacy (EuroS&P)*, 2017.

[32] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[33] V. Costan and S. Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/2016/086`.

[34] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2015.

[35] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceeding of the 28th international conference on Software engineering (ICSE)*, 2006.

[36] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[37] G. J. Duck and R. H. C. Yap. Heap bounds protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC'16)*, 2016.

[38] G. J. Duck, R. H. C. Yap, and L. Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*, 2017.

[39] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[40] B. Fitzpatrick. Distributed caching with Memcached. In *Linux Journal*, 2004.

[41] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceeding of the 21st USENIX Security Symposium (Sec)*, 2012.

[42] J. Graham-Cumming. Incident report on memory leak caused by Cloudflare parser bug. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`. Accessed: February, 2017.

[43] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[44] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG)*, 1997.

[45] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[46] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[47] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[48] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, 2014.

[49] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[50] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[51] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[52] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[53] 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.

[54] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[55] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *Leibniz International Proceedings in Informatics (LIPIcs)*, 2015.

[56] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[57] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, 2010.

[58] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2007.

[59] A. M. Niranjan Hasabnis and R. Sekar. Light-weight bounds checking. In *Proceedings of the 2012 ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2012.

[60] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in MapReduce. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[61] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. 2017, arXiv:1702.00719.

[62] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. Fex: A software systems evaluator. In *Proceedings of the 2017 IEEE/IFIP 47th International Conference on Dependable Systems & Networks (DSN)*, 2017.

[63] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[64] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, 2004.

[65] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[66] Roman Shaposhnik. What does dynamic linking and communism have got in common? `https://blogs.oracle.com/rvs/entry/what_does_dynamic_linking_and`. Accessed: February, 2017.

[67] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2004.

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

[69] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012.

[70] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004.

[71] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[72] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[73] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security (EUROSEC)*, 2009.

[74] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[75] J. Tang, Y. Cui, Q. Li, K. Ren, J. Liu, and R. Buyya. Ensuring security and privacy preservation for cloud data services. *ACM Computing Surveys*, 2016.

[76] Ulrich Drepper. Static Linking Considered Harmful. `https://www.akkadia.org/drepper/no_static_linking.html`. Accessed: February, 2017.

[77] V. van der Veen, N. Dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012.

[78] Various authors. Dynamic Linking. `http://harmful.cat-v.org/software/dynamic-linking/`. Accessed: February, 2017.

[79] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[80] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[81] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[82] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *ACM SIGSOFT Software Engineering Notes*, 2004.

[83] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[84] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2013.