Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack*

https://Intel-MPX.github.io/

OLEKSII OLEKSENKO and DMITRII KUVAISKII, TU Dresden, Germany
PRAMOD BHATOTIA, The University of Edinburgh, United Kingdom
PASCAL FELBER, University of Neuchâtel, Switzerland
CHRISTOF FETZER, TU Dresden, Germany

Memory-safety violations are the primary cause of security and reliability issues in software systems written in unsafe languages. Given the limited adoption of decades-long research in software-based memory safety approaches, as an alternative, Intel released Memory Protection Extensions (MPX)—a hardware-assisted technique to achieve memory safety. In this work, we perform an exhaustive study of Intel MPX architecture along three dimensions: (a) performance overheads, (b) security guarantees, and (c) usability issues.

We present the first detailed root cause analysis of problems in the Intel MPX architecture through a cross-layer dissection of the entire system stack, involving the hardware, operating system, compilers, and applications. To put our findings into perspective, we also present an in-depth comparison of Intel MPX with three prominent types of software-based memory safety approaches. Lastly, based on our investigation, we propose directions for potential changes to the Intel MPX architecture to aid the design space exploration of future hardware extensions for memory safety.

CCS Concepts: • **Security and privacy** → *Software security engineering*;

Additional Key Words and Phrases: Memory safety; ISA extensions; Intel MPX

ACM Reference Format:
Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack: https://Intel-MPX.github.io/. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 28 (June 2018), 30 pages. https://doi.org/10.1145/3224423

1 INTRODUCTION

The majority of critical software systems is written in low-level languages such as C or C++. These languages give programmers explicit and fine-grained control over memory, which is especially important for development of efficient software systems. Unfortunately, the ability to directly control memory often leads to violations of *memory safety* properties, i.e., illegal accesses to unintended memory regions [53].

*The paper presents only the summarized results—the detailed analysis is published on our website: https://Intel-MPX.github.io/. Clicking on most figures/plots and (sub-)section headings in the paper will open a corresponding webpage.

Authors' addresses: Oleksii Oleksenko; Dmitrii Kuvaiskii, TU Dresden, Dresden, Germany, {firstname.lastname}@tu-dresden.de; Pramod Bhatotia, The University of Edinburgh, Edinburgh, United Kingdom, pramod.bhatotia@ed.ac.uk; Pascal Felber, University of Neuchâtel, Neuchâtel, Switzerland, pascal.felber@unine.ch; Christof Fetzer, TU Dresden, Dresden, Germany, Christof.Fetzer@tu-dresden.de.

2476-1249/2018/6-ART28 $15.00
https://doi.org/10.1145/3224423

In particular, memory-safety violations emerge in the form of *spatial* and *temporal* errors. Spatial errors—buffer overflows and out-of-bounds accesses—occur when a program reads from or writes to a different memory region than the one expected by the developer. Temporal errors—wild and dangling pointers—appear when trying to use an object before it was created or after it was deleted.

These memory-safety violations are the root cause of most reliability and security vulnerabilities in legacy software systems [50]. Given its importance, over decades, a plethora of solutions have been proposed for enforcing memory safety in unsafe languages, ranging from static analysis to language extensions [1, 4, 12, 19, 28, 30, 32, 35, 37, 38, 40, 46, 58].

In this work, we concentrate on *deterministic dynamic bounds-checking* since it is widely regarded as the only way of defending against *all* memory safety attacks [34, 50]. Bounds-checking techniques augment the original unmodified program with metadata (bounds of live objects or allowed memory regions) and insert checks against this metadata before each memory access. Unfortunately, the state-of-the-art software-based approaches have seen limited adoption in practice, largely owing to high performance overhead (50–150%), incomplete security guarantees, and incompatibility with legacy libraries.

To overcome these limitations, Intel released Memory Protection Extensions (Intel MPX)—a set of new ISA extensions as part of the Skylake microarchitecture [22, 23]. Its underlying idea is to provide hardware assistance for enforcing memory safety, in the form of new instructions and registers, as an alternative to the software-based approaches. Through its cross-layer support, involving the hardware, operating system, compiler, and application levels—the Intel MPX architecture promises to address the performance, security, and compatibility issues of previous software-only approaches.

In this paper, we showcase that Intel MPX has flaws in all three important dimensions: (a) performance and memory overheads, (b) security guarantees, and (c) usability issues. Performance is important because only solutions with low (up to 10–20%) runtime overhead have a chance to be adopted in practice [50]. Security assessment of the available implementation on a diverse set of memory vulnerabilities is required to verify advertised security guarantees. And lastly, usability gives us insights on application-specific issues that arise when using the Intel MPX system stack and need to be manually fixed.

This work presents the first detailed cross-layer dissection of the Intel MPX system stack, comprising the hardware, operating system, compilers, and applications. Our work provides insights on the causes of overheads, security, and usability issues in both the Intel MPX architecture and its surrounding infrastructure. To fully explore Intel MPX's pros and cons, we put the results into perspective by comparing with existing software-based solutions. In particular, we compared Intel MPX with three prominent classes of memory safety: trip-wire — AddressSanitizer [46], object-based—SAFECode [12], and pointer-based—SoftBound [35]. Surprisingly, even though Intel MPX is a specially designed hardware-assisted approach, it is *not* faster than the software-based approaches.

We investigate Intel MPX and the aforementioned software-based approaches using a comprehensive range of micro-benchmarks and benchmark suites. Our investigation reveals that although Intel MPX strives to solve an important problem, it is not yet practical because of the following issues:

- New Intel MPX instructions are not as fast as expected and cause up to 4× slowdown in the worst case, although compiler optimizations amortize it and lead to runtime overheads of ~50% on average.
- In contrast to other solutions, Intel MPX provides no protection against temporal memory safety errors.

- Intel MPX does not support multithreading inherently, which can lead to unsafe data races in legacy threaded programs and if compilers do not synchronize bounds explicitly.
- Intel MPX does not support several common C/C++ programming idioms due to restrictions on the allowed memory layout. In our experiments, 8–13% programs did not run correctly without substantial code changes and additionally, 18% required non-intrusive manual fixes.
- Intel MPX is conflicting with some other ISA extensions resulting in performance issues. More specifically, we investigated the issues that arise when Intel MPX is used in combination with Intel TSX and Intel SGX.
- Lastly, MPX instructions incur significant performance penalty (15+%) even on earlier Intel CPU generations without MPX support (e.g., Haswell).

Note that some of these flaws could be fixed by making a few minor changes to Intel MPX. Specifically, there are relatively straightforward ways of implementing temporal safety, and the compatibility problems could probably be fixed too. Yet, most of the performance and usability issues are fundamental and would require substantial changes to the design of Intel MPX.

As of less critical issues, the supporting compiler infrastructure (compiler passes and runtime libraries) is not mature enough and has bugs, such that 3–10% programs cannot compile/run. Fortunately, these issues could be resolved by improving the toolchain, in contrast to the aforementioned fundamental issues that require hardware modifications.

All these issues created a growing trend of *re-purposing* Intel MPX to provide coarse-grained isolation of memory regions. In particular, out of the whole MPX stack, only two bounds-checking instructions are usually employed to provide efficient Software Fault Isolation [7, 26, 29, 33, 42]. Meanwhile, we know of *no* successful attempts to use MPX for the original purpose of complete memory safety. In fact, the interest of using Intel MPX for its direct purpose has become so little, that GCC is deprecating the feature in GCC 9 [41].

Nevertheless, there is an urgent need for a lightweight practical hardware-assisted memory-safety mechanism to end the eternal war in memory [50]. Based on our findings, we propose potential directions for extensions to the Intel MPX architecture to address three important issues: (1) performance and memory overheads, (2) security properties, and (3) transparent multithreading support. Our work seeks to help in paving the way for future correct-by-design hardware technologies.

To summarize, our paper makes the following contributions:

- We present the first detailed analysis of problems in the Intel MPX architecture through a cross-layer dissection of the entire MPX system stack (§3).
- To put our findings into perspective, we present a comparison of Intel MPX with three prominent types of software-based memory safety approaches (§4).
- Lastly, we suggest future directions for potential improvements to the MPX architecture (§6).

2 BACKGROUND

Before we delve into the details of the Intel MPX architecture, we first present a brief background on state-of-the art software-based memory safety approaches. We analyze and evaluate these approaches to put the design and results of MPX into perspective.

All spatial and temporal bugs, as well as memory attacks built on such vulnerabilities, are caused by an access to a prohibited memory region. Accordingly, to prevent such errors, *memory safety* must be imposed on the program, i.e., the following invariant must be enforced: memory accesses must always stay within the originally intended (referent) objects.

To this end, software-based runtime bounds-checking techniques are used, broadly classified as trip-wire, object-based, and pointer-based [34]. For comparison with Intel MPX, we chose a

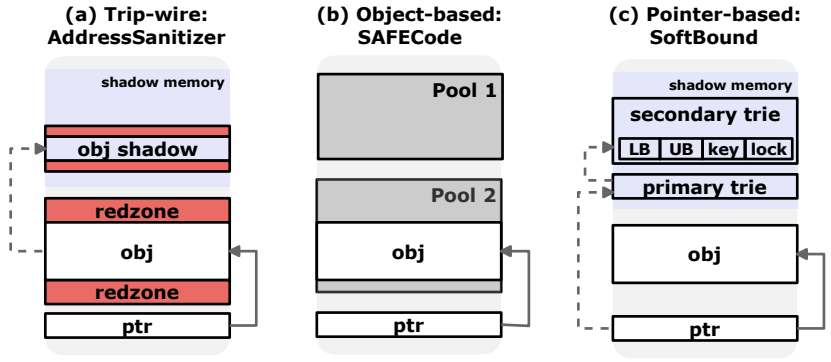

Fig. 1. Designs of three memory-safety classes.

prominent example from each of the three classes: AddressSanitizer, SAFECode, and SoftBound (Figure 1 highlights the differences between them).

Trip-wire approach: AddressSanitizer [46]. This class surrounds all objects with regions of marked (poisoned) memory called *redzones*, so that any overflow will change values in this—otherwise invariable—region and will be consequently detected [19, 20, 38, 46]. In particular, AddressSanitizer reserves 1/8 of all virtual memory for the *shadow memory* at program startup; the memory is accessed only by the instrumentation and not the original program. AddressSanitizer updates data in the shadow memory whenever a new object is created and freed, and inserts checks on shadow memory before memory accesses to objects. The check itself looks like this:

```
shadowAddr = MemToShadow(ptr)
if (ShadowIsPoisoned(shadowAddr)) Error()
```

In addition, AddressSanitizer provides means to probabilistically detect temporal errors via a *quarantine zone*: if a memory region has been freed, it is kept in the quarantine for some time before it becomes allowed for reuse.

AddressSanitizer was built for debugging purposes and is not targeted for security. It is sometimes used in this context for the lack of better alternatives [6, 34] but such use is discouraged [55] (e.g., because attackers may abuse the debugging features in AddressSanitizer's run-time library). For example, it may not detect non-contiguous out-of-bounds violations. Nevertheless, it detects many spatial bugs and significantly raises the bar for the attacker. It is also the most widely-used technique in its class, comparing favorably to other trip-wire techniques such as LBC [19], Purify [20], and Valgrind [38].

Object-based approach: SAFECode [11, 12]. This class's main idea is enforcing the intended referent, i.e., making sure that pointer manipulations do not change the pointer's referent object [1, 11, 12, 14–16, 44]. In SAFECode, this rule is relaxed: each object is allocated in one of several fine-grained partitions—*pools*—determined at compile-time using pointer analysis; the pointer must always land into the predefined pool. This technique allows powerful optimizations and simple runtime checks against the pool bounds:

```
poolAddr = MaskLowBits(ptr)
if (poolAddr not in predefinedPoolAddrs) Error()
```

In addition, we considered other object-based approaches. CRED [44] has huge performance overheads, mudflap [16] is deprecated in newer versions of GCC, and Baggy Bounds Checking [1] and Low-Fat Pointers [14, 15] are not open sourced.

(a) Original code

```
   struct obj { char buf[100];  int len }
1  obj* a[10]                                  ;; Array of pointers to objs
2  for (i=0; i<M; i++):                        ;; M may be greater than 10
3    ai = a + i                                ;; Pointer arithmetic on a
4    objptr = load ai                          ;; Pointer to obj at a[i]
5    lenptr = objptr + 100                     ;; Pointer to obj.len
6    len = load lenptr
```

(b) Intel MPX

```
1  obj* a[10]
2  a_b = bndmk a, a+79                         ;; Make bounds [a, a+79]
3  for (i=0; i<M; i++):
4    ai = a + i
5    bndcl a_b, ai                             ;; Lower-bound check of a[i]
6    bndcu a_b, ai+7                           ;; Upper-bound check of a[i]
7    objptr = load ai
8    objptr_b = bndldx ai                      ;; Bounds for pointer at a[i]
9    lenptr = objptr + 100
10   bndcl objptr_b, lenptr                    ;; Checks of obj.len ⎤
11   bndcu objptr_b, lenptr+3                                       ⎦
12   len = load lenptr
```

Fig. 2. Example of bounds checking using Intel MPX.

Pointer-based approach: SoftBound [35, 36]. Such approaches keep track of pointer bounds (the lowest and the highest address the pointer is allowed to access) and check each memory write and read against them [25, 32, 35–37, 47]. Note how SoftBound associates metadata *not* with an object but rather with a pointer to the object. This allows pointer-based techniques to detect intra-object overflows (one field overflowing into another field of the same struct) by *narrowing bounds* associated with the particular pointer.

For our comparison, we used the SoftBound+CETS version which keeps pointer metadata in a two-level trie—similar to MPX's bounds tables—and introduces a scheme to protect against temporal errors [36]. The checks are as follows:

```
LB,UB,key,lock = TrieLookup(ptr)
if (ptr < LB or ptr > UB or key != *lock) Error()
```

As for other pointer-based approaches, MemSafe [47] is not open sourced, and CCured [37], Cyclone [25], and CheckedC [32] require manual changes in programs.

3 ANALYSIS OF THE INTEL MPX STACK

Intel Memory Protection Extension (MPX) provides a hardware assisted pointer-based mechanism for memory safety. It is a cross-layer solution: *(i)* hardware layer introduces new instructions and registers to operate on pointer bounds, *(ii)* operating system layer provides support for memory management and exception handling, *(iii)* compiler and runtime layer adds instrumentation passes and wrappers, and *(iv)* application layer allows for MPX-specific changes in programs. In the following section, we separately analyze each layer of the Intel MPX system stack.

Instruction	Description	Latency	Throughput
`bndmk b,m`	create pointer bounds	1	2
`bndcl b,m`	check mem-operand against lower bound	1	1
`bndcl b,r`	check reg-operand against lower bound	1	2
`bndcu b,m`	check mem-operand against upper bound	1	1
`bndcu b,r`	check reg-operand against upper bound	1	2
`bndmov b,m`	move pointer bounds from memory	1	1
`bndmov b,b`	move pointer bounds to other register	1	2
`bndmov m,b`	move pointer bounds to memory	2	0.5
`bndldx b,m`	load pointer bounds from BT	4-6	0.4
`bndstx m,b`	store pointer bounds in BT	4-6	0.3

Note: bndcu has a one's complement version; we skip it for clarity

Table 1. Latency (cycles/instr) and throughput (instr/cycle) of Intel MPX instructions; b—MPX bounds register; m—memory operand; r—general-purpose register operand.

Before going into details, we give a brief overview of MPX on a simple example shown in Figure 2a. The original program allocates an array `a[]` with 10 pointers to objects of type `obj` (Line 1). Next, it iterates through the first `M` array items to read the objects' lengths (Lines 2–6). Since M is a variable, a bug may set M to a value that is larger than 10 and an overflow will happen.

Figure 2b shows the code with Intel MPX enabled. First, the bounds for the array `a[]` are created on Line 2 (the array contains 10 pointers each 8 bytes wide, hence the upper-bound offset of 79). Then in the loop, before the array item access on Line 7, two MPX bounds checks are inserted to detect if `a[i]` overflows (Lines 5–6).

Now that the pointer to the object is loaded in `objptr`, the program wants to load the `obj.len` subfield. By design, Intel MPX must protect this second load by checking the bounds of the `objptr` pointer. Thus, the bounds are first loaded via `bndldx` instruction (Line 8) and then the two bounds checks are inserted before the load of the length value on Lines 10–11 (narrowing of bounds is not shown for simplicity, see §3.3).

3.1 Hardware

At its core, Intel MPX provides 7 new instructions and a set of 128-bit bounds registers. The Skylake architecture provides four registers named `bnd0`–`bnd3`. Each of them stores a lower bound in bits 0–63 and an upper bound in bits 64–127.

Instruction set. The new MPX instructions are: `bndmk` to create new bounds, `bndcl` and `bndcu`/`bndcn` to compare the pointer value against the lower and upper bounds in `bnd` respectively, `bndmov` to move bounds from one `bnd` register to another and to spill them to stack, and `bndldx`/`bndstx` to load/store pointer bounds in special Bounds Tables respectively.

It is interesting to compare the benefits of hardware implementation of bounds-checking against the software-only counterpart—SoftBound [35, 36]. First, Intel MPX introduces separate bounds registers to lower register pressure on the general-purpose register file, something that software-only approaches suffer from. Second, dedicated `bndcl` and `bndcu` instructions substitute the software-based "compare and branch" instruction sequence, saving one cycle and exerting no pressure on branch predictor.

Storing bounds in memory. The current version of Intel MPX has only 4 bounds registers, which is clearly not enough for real-world programs. All additional bounds have to be stored (spilled) in memory, similar to spilling data out of registers. A simple and fast option is to copy them

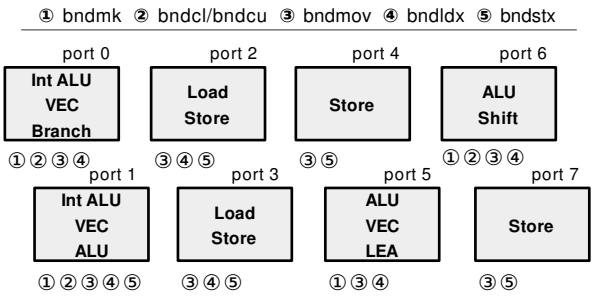

Fig. 3. Distribution of Intel MPX instructions among execution ports (Intel Skylake).

directly on stack with `bndmov`. However, it works only inside a single stack frame: if a pointer is later reused in another function, its bounds will be lost. To solve this issue, two instructions were introduced—`bndstx` and `bndldx`. They store/load bounds to/from a memory location derived from the address of the pointer itself (see Figure 2b, Line 8), thus making it easy to find pointer bounds without any additional information, though at a price of higher complexity.

When `bndstx` and `bndldx` are used, bounds are stored in a memory location calculated with two-level address translation scheme, similar to virtual address translation. In particular, each pointer has an entry in a Bounds Table (BT), which is allocated dynamically and is comparable to a page table. Addresses of BTs are stored in a Bounds Directory (BD). For a specific pointer, its entries in the BD and the BT are derived from the memory address in which the pointer is stored. In contrast to virtual address translation, no dedicated hardware like MMU or TLB cache is introduced, thus Intel MPX can experience performance degradation caused by cache thrashing (see §4.1).

Figure 4 shows pointer address translation on the example of `bndldx`. In the first stage, the CPU: ① extracts the offset of BD entry from bits 20–47 of the pointer address and shifts it by 3 bits (since all BD entries are 2^3 bits long), ② loads the base address of BD from the `BNDCFGx` register, and ③ sums the base and the offset and loads the BD entry from the resulting address. In the second stage, the CPU: ④ extracts the offset of BT entry from bits 3–19 of the pointer address and shifts it by 5 bits (since all BT entries are 2^5 bits long), ⑤ shifts the loaded entry—which corresponds to the base of BT—by 3 to remove the metadata, and ⑥ sums the base and the offset and ⑦ finally loads the BT entry from the resulting address. Note that a BT entry has an additional "pointer" field—if the actual pointer value and the value in this field mismatch, Intel MPX will mark the bounds as always-true (`INIT`), required for interoperability with legacy code.

This address translation mechanism is expensive—it requires approximately 3 register-to-register moves, 3 shifts, and 2 memory loads. On top of it, since these 2 loads are non-contiguous, the protected application has worse cache locality.

Analysis. As the first step in our evaluation, we measured latency and throughput of MPX instructions (Table 1), as well as their distribution among execution ports (Figure 3). The major bottleneck is storing/loading the bounds with `bndstx` and `bndldx` since they undergo a complex algorithm of accessing bounds tables.

In our measurement study (§4), we observed that Intel MPX protection does not increase the IPC (instructions/cycle) of programs, which is usually the case for memory-safety techniques (see Figure 11). This was surprising: we expected that Intel MPX would take advantage of underutilized CPU resources for programs with low original IPC. To understand what causes this bottleneck, we measured the throughput of typical MPX check sequences.

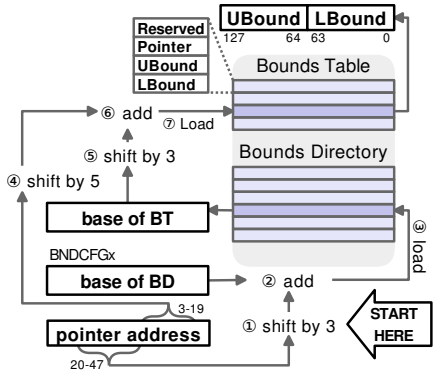

Fig. 4. Loading of pointer bounds in Intel MPX.

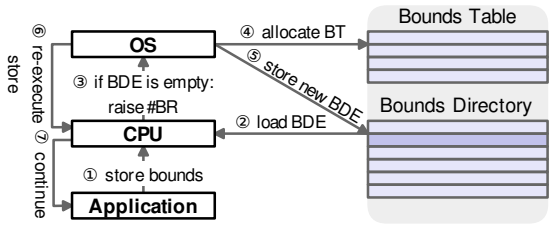

Fig. 5. The procedure of Bounds Table creation.

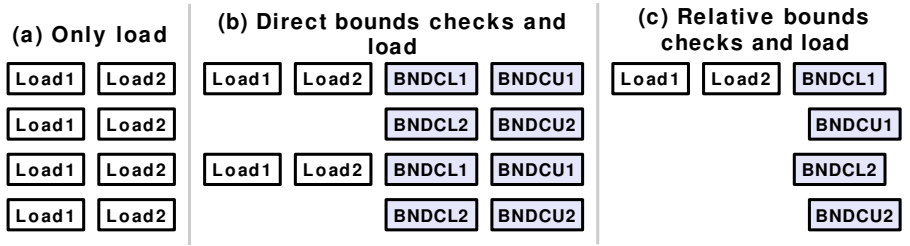

Fig. 6. Bottleneck of bounds checking illustrated.

The measurements pointed to a bottleneck of `bndcl/u b,m` instructions due to contention on a single execution port. Without checks (Figure 6a), our original benchmark could execute two loads in parallel, achieving a throughput of 2 IPC (note that the loaded data is always in a Memory Ordering Buffer). After adding `bndcl/u b,r` checks (Figure 6b), IPC increased to three instructions per cycle (3 IPC): one load, one lower-, and one upper-bound check per cycle. For `bndcl/u b,m` checks (Figure 6 c), however, IPC became *less* than original: two loads and four checks were scheduled in four cycles, thus IPC of 1.5. In summary, the final IPC was ~1.5–3 (compare to original IPC of 2), proving that the MPX-protected program typically has *approximately the same IPC as the original*. (This causes major performance degradation, as Figures 9 and 10 show.)

3.2 Operating System

The OS has two main responsibilities in the context of Intel MPX: it handles bounds violations and manages BTs, i.e., creates and deletes them. Both these actions are hooked to a new exception class, #BR, introduced solely for Intel MPX.

Bounds exception handling. If an MPX-enabled CPU detects a bounds violation, #BR is raised and the processor traps into the kernel. The kernel gets the violating address and bounds and delivers them to the application using the SIGSEGV signal. At this point the application developer has a choice: she can either provide an ad-hoc signal handler to recover or choose one of the default policies: crash, print an error, or ignore it.

Bounds tables management. Two levels of bounds address translation are managed differently: BD is allocated only once by a runtime library (at program startup) and BTs have to be created

		Increase in # of instructions (%)	
Type	**Slowdown**	**User space**	**Kernel space**
allocation	2.33×	7.5	160
+ de-allocation	2.25×	10	139

Table 2. Worst-case OS impact on performance of MPX.

dynamically on-demand. The later is a task of OS. The procedure is presented in Figure 5. Each time an application tries to store pointer bounds ①, the CPU loads the corresponding entry from the BD and checks if it is a valid entry ②. If the check fails, the CPU raises #BR and traps into the kernel ③. The kernel allocates a new BT ④, stores its address in the BD entry ⑤ and returns in the user space ⑥. Then, the CPU stores bounds in the newly created BT and continues executing the application in the normal mode of operation ⑦. Since the application is oblivious to BT allocation, the OS also frees these tables when they become unused.

Analysis. To illustrate the additional overhead of allocating and de-allocating BTs, we created two microbenchmarks.The first one indirectly creates a large amount of BTs by storing a set of pointers in sparse memory locations. The second one, in addition, frees all the memory right after it has been assigned, thus triggering BT de-allocation.

Our measurement results are shown in Table 2. The overheads observed are 2.3× and are caused purely by the BT management in the kernel (note the increase in number of instructions executed in kernel space). From this we conclude that the OS may cause up to 2.3× slowdown, although we did not encounter this scenario in real programs. We believe, the main reason why the overhead does not manifest itself in real applications is because pointers are usually not sparse enough to cause frequent allocations of BTs.

3.3 Compiler and Runtime Library

Hardware MPX support in the form of new instructions and registers significantly lowers performance overhead of each *separate* bounds-checking operation. However, the main burden of efficient and complete bounds checking of programs lies on the compiler and its associated runtime.

Compiler support. As of the date of this writing, only GCC 5.0+ and ICC 15.0+ have support for Intel MPX [17, 23] (we used GCC 6.1.0 and ICC 17.0.0). Both GCC and ICC introduce the new compiler pass called Pointer(s) Checker.

In short, Pointer Checker instruments the original program as follows. (1) It allocates static bounds for global variables and inserts `bndmk` instructions for stack-allocated ones. (2) It inserts `bndcl` and `bndcu` bounds-check instructions before each load or store from a pointer. (3) It moves bounds from one `bnd` register to another using `bndmov` whenever a new pointer is created from an old one. (4) It spills least used bounds to stack via `bndmov` if running out of available `bnd` registers. (5) It loads/stores the associated bounds via `bndldx`/`bndstx` respectively whenever a pointer is loaded/stored.

One of the advantages of Intel MPX is that it supports *narrowing of struct bounds* by design. Consider struct `obj` from Figure 2. It contains two fields: a 100B buffer `buf` and an integer `len` right after it. It is easy to see that an off-by-one overflow in `obj.buf` will spillover and corrupt the adjacent `obj.len`. Approaches like AddressSanitizer and SAFECode by design cannot detect such intra-object overflows. In contrast, Intel MPX can be instructed to narrow bounds when code accesses a specific field of a struct, e.g., on Line 9 in Figure 2b. Here, instead of checking against the bounds of the full object, the compiler would shrink `objptr_b` to only four bytes and compare

Compiler & runtime issues	GCC	ICC
– Poor MPX pass optimizations *	22/38	3/38
– Bugs in MPX compiler pass:		
– incorrect bounds during function calls	–	2/38
– conflicts with auto-vectorization passes	–	3/38
– corrupted stack due to C99 VLA arrays	–	3/38
– unknown internal compiler error	1/38	–
– Bugs and issues in runtime libraries:		
– Missing wrappers for libc functions	all	all
– Nullified bounds in `memcpy` wrapper	all	–
– Performance bug in `memcpy` wrapper	–	all

*One compiler has > 10% worse results than the other

Table 3. Issues in the compiler pass and runtime libraries of Intel MPX. Columns 2 and 3 show number of affected programs (out of total 38 tested in §4).[1]

against these narrowed bounds on Lines 10–11. Narrowing of bounds may require (sometimes intrusive) changes in the source code, and is enabled by default.

By default, the MPX pass instruments both memory writes and reads. The user can instruct the MPX pass to instrument only writes to reduce performance overhead (from 2.5× to 1.3× for GCC). This will provide lower but still sufficiently high security guarantees since the most dangerous bugs are those that overwrite memory (classic overflows to gain privileged access to the remote machine).

For performance, both GCC and ICC employ common optimizations: (1) Removing bounds-checks when the compiler can statically prove safety of memory access; (2) Moving (hoisting) bounds-checks out of simple loops.

Runtime library. As a final step of the MPX-enabled build process, the application must be linked against two MPX-specific libraries: `libmpx` and `libmpxwrappers` (`libchkp` for ICC).

The `libmpx` library is responsible for MPX initialization at program startup: it enables hardware and OS support, configures MPX runtime options (passed through environment variables), and registers a `#BR` exception handler.

The `libmpxwrappers` library (and its analogue `libchkp` in ICC) contains wrappers for functions from C standard library (libc). MPX implementations do not instrument libc and instead wrap all its functions with bounds-checking counterparts.

Issues. For both GCC and ICC, the compiler and runtime support have a number of issues summarized in Table 3.

Concerning performance, current implementations of GCC and ICC take different stances when it comes to optimizing MPX code. GCC is conservative and prefers stability over performance gains. On many occasions, we noticed that the GCC-MPX pass *disables* other optimizations, e.g., loop unrolling and autovectorization. It also hoists bounds-checks out of loops less often than ICC does.

[1] All bugs were acknowledged by developers. Bug reports:
https://software.intel.com/en-us/forums/intel-c-compiler/topic/700550;
https://software.intel.com/en-us/forums/intel-c-compiler/topic/700675;
https://software.intel.com/en-us/forums/intel-c-compiler/topic/701764;
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=78631

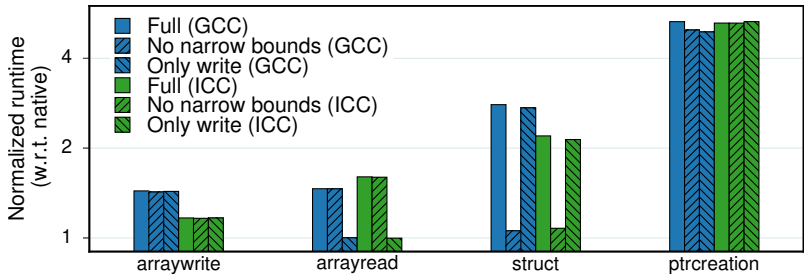

Fig. 7. Intel MPX overheads in 3 possible scenarios: application is dominated by bounds-checking (*arraywrite* and *arrayread*), by bounds creation and narrowing (*struct*), and by bounds propagation (*ptrcreation*).

ICC, on the other hand, is more aggressive in its MPX-related optimizations and does *not* prevent other aggressive optimizations from being applied. Unfortunately, this intrusive behavior renders ICC's pass less stable: we detected three kinds of compiler bugs due to incorrect optimizations.

We also observed issues with the runtime wrapper libraries. First, only a handful of most widely-used libc functions are covered, e.g., `malloc`, `memcpy`, `strlen`, etc. This leads to undetected bugs when other functions are called, e.g., the bug with `recv` in Nginx. For use in production, these libraries must be expanded to cover *all* of libc. Second, while most wrappers follow a simple pattern of "check bounds and call real function", there exist more complicated cases. For example, `memcpy` must be implemented so that it copies not only the contents of one memory area to another, but also all associated pointer bounds in BTs. GCC library uses a fast algorithm to achieve this, but ICC's `libchkp` has a performance bottleneck (see also §4).

Analysis. To understand the impact of different compiler flags and optimizations, we wrote four microbenchmarks, each highlighting a separate MPX feature. Two benchmarks—*arraywrite* and *arrayread*—perform writes to/reads from memory and stress `bndcl` and `bndcu` accordingly. The *struct* benchmark writes in an inner array inside a struct and stresses the bounds-narrowing feature via `bndmk` and `bndmov`. Finally, the *ptrcreation* benchmark constantly assigns new values to pointers and stresses bounds propagation via `bndstx`. Figure 7 shows their performance overheads.

We can notice three interesting details. First, *arraywrite* and *arrayread* represent bare overhead of bounds-checking instructions (all in registers), 50% in this case. *struct* has a higher overhead of 2.1–2.8× due to the more expensive making and moving of bounds to and from the stack. The 5× overhead of *ptrcreation* is due to storing of bounds—the most expensive MPX operation (see §3.1). Such high overhead is alarming because pointer-intensive applications require many loads and stores of bounds.

Second, there is a 25% difference between GCC and ICC in *arraywrite*. This is the effect of optimizations: GCC's MPX pass blocks loop unrolling while ICC's implementation takes advantage of it. (Interestingly, the same happened in case of *arrayread* but the native ICC version was optimized even better, which led to a relatively poor performance of ICC's MPX.)

Third, the overhead of *arrayread* becomes negligible with the only-writes MPX version: the only memory accesses in this benchmark are reads which are left uninstrumented. Finally, the same logic applies to *struct*—disabling narrowing of bounds effectively removes expensive `bndmk` and `bndmov` instructions and lowers overhead to a bare minimum.

3.4 Applications

Next, we discuss three main issues of MPX at the application level.

Application-level issues	GCC	ICC
– Flexible or variable-sized array (`arr[1]` / `arr[]`)	7/38	7/38
– of them fixable:	7/7	7/7
– Accessing struct through struct field*	1/38	3/38
– of them fixable:	0/1	0/3
– Custom memory management	2/38	2/38
– of them fixable:	0/2	0/2

* GCC affects less programs due to milder rules w.r.t. first field of struct

Table 4. Applications may violate memory-model assumptions of Intel MPX. Columns 2 and 3 show number of misbehaving programs (out of total 38).

Not supported C idioms. MPX does not work correctly with several common C idioms (see Table 4), especially when narrowing of bounds is applied and applications deviate from the standard memory model [8, 31]. First, *flexible array fields* with array size of one (e.g., `arr[1]`) as well as *variable-sized arrays* (e.g., `arr[]`) get incorrect bounds under Intel MPX, which leads to false positives. (Note that the C99-standard `arr[0]` is acceptable and does not break programs.) Second, *intra-structure accesses*—using a struct field (usually the first field of struct) to access other fields of the struct—breaks the assumptions of Intel MPX and leads to runtime `#BR` exceptions.[2] Third, some programs introduce *custom memory management* for performance, ignoring restrictions of the C memory model completely, which leads to false positives. More detailed discussion of this issues can be found on the website.

Ultimately, all such non-compliant cases must be fixed (indeed, we patched flexible/variable-length array issues to work under Intel MPX). However, sometimes the user may have strong incentives against modifying the original code. In this case, she can opt for slightly worse security guarantees and disable narrowing of bounds. Another non-intrusive alternative is to mark objects that must *not* be narrowed (e.g., flexible arrays) with a special attribute.

Multithreading issues. Current Intel MPX implementations may introduce false positives and negatives in multithreaded programs [8, 45]. The problem arises when a pointer and its bounds are loaded or stored. Ideally, these two operations must be performed *atomically*, but neither the current hardware implementation nor GCC/ICC compilers enforce this atomicity.

Consider the example in Figure 8. A "pointer bounds" data race happens on the `arr` array of pointers. The background thread fills this array with all pointers to the first or to the second object alternately. Meanwhile, the main thread accesses whatever object is currently pointed-to by the array items. Note that if `offset` is zero, then the main thread always accesses the correct object, otherwise it accesses an incorrect, adjacent object. The second case, introduces a concurrency vulnerability which could be exploited by an adversary [59].

With Intel MPX, additional `bndstx` instructions are inserted in Lines 2–3 to store the object bounds. Also, a `bndldx` instruction is inserted after Line 6 to retrieve the bound for an object referenced by `ai`. Bound checks `bndcl` and `bndcu` are also added after Line 6, before the actual access to the object. Now, the following race can occur. The main thread loads the pointer-to-first-object from the array and—right before loading the corresponding bound—is preempted by the background thread. The background thread overwrites all array items such that they point to the second object, and also overwrites the corresponding bounds. Finally, the main thread is scheduled back and loads the bound, however, the bound now corresponds to the second object. The main

[2] GCC makes an exception for the case of the access to other struct fields through the *first field* since it is such a popular practice, but ICC is strict and does not have this special rule.

```
  char* arr[1000]                                   ;; Array with MPX data race
  char obj1                                         ;; Two adjacent objects ⎤
  char obj2                                                                 ⎦
1 while (true):                                     ;; Background thread
2   for (i=0; i<1000; i++)   arr[i] = &obj1
3   for (i=0; i<1000; i++)   arr[i] = &obj2
4 while (true):                                     ;; Main thread
5   for (i=0; i<1000; i++):
6     ai = arr[i]
7     result += *(ai + offset)
```

Fig. 8. A multithreaded program that breaks Intel MPX. If `offset=0` then false alarms, else undetected bugs. Note that the code has undefined behavior under C11 memory model.

thread is left with the pointer to the first object but with the bounds of the second one—breaking the original program. If implemented in C, this test causes false positives (`offset=0`) and false negatives (`offset=1`) in both GCC and ICC versions.

We must note that the multithreading code from Figure 8 does not conform to the memory model introduced in C11 and C++11 standards. Under C11, this code by itself has undefined behavior, and the compilers are free to produce a potentially misbehaving program. Thus, the discussion above applies only to legacy C/C++ code where the data race in Figure 8 is technically allowed. Under the new C11 thread model, `arr` in Figure 8 must be declared as an array of *atomic* pointers. Ideally, the compiler would recognize loads/stores of atomic pointers and enclose them and their corresponding bounds loads/stores in a critical section. Enclosing the atomic pointer-and-bounds update in one synchronization block (e.g., via locks or TSX) could be a trivial addition to the MPX compiler pass, however, at the cost of substantial performance overhead. Unfortunately, our investigation proved that current compilers do not generate correct code neither with GCC-specific `__atomic_store()` nor with C11-defined `_Atomic` types.

Interaction with other ISA extensions. Intel MPX can cause issues when used together with other ISA extensions, such as Intel TSX and Intel SGX: When used inside an Intel TSX hardware transaction, it may cause additional spurious aborts [24, 27], further reducing applications' performance. In the case of SGX, the architectural features of SGX enclaves, especially the restricted EPC memory region, conflict with the memory requirements of Intel MPX. In our experience [28], some real-world applications using MPX inside the enclave (even with tiny input sizes) crash because of the high memory overheads of Intel MPX.

4 EVALUATION

In this section, we compare Intel MPX with the software-based approaches (§2) across three dimensions: performance (§4.1), security (§4.2), and usability (§4.3).

Applications. We based our evaluation on prominent benchmark suites: PARSEC 3.0 [5], Phoenix 2.0 [43], and SPEC CPU2006 [21] (38 benchmarks in total), and on attacks from RIPE [56]. In addition, we experimented with three real-world case studies: Apache, Nginx, and Memcached. We also evaluated security guarantees of all four approaches with real-world exploits such as Heartbleed, denial-of-service, and a ROP attack. Due to space constraints, we cover the results for case studies on the corresponding website page.

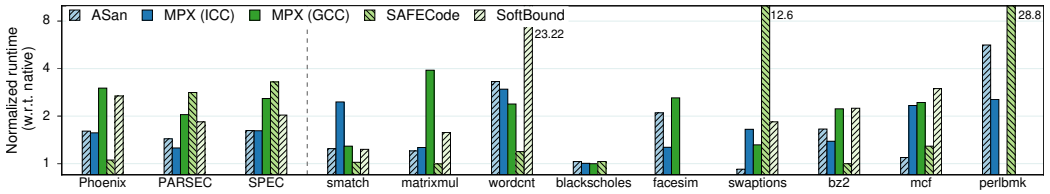

Fig. 9. Performance (runtime) overhead with respect to native version. (Lower is better.)

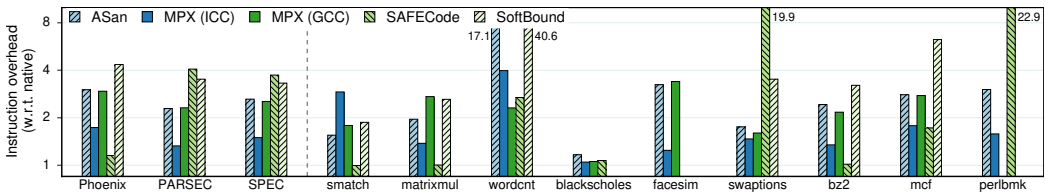

Fig. 10. Increase in number of instructions with respect to native version. (Lower is better.)

Experimental setup. The machines we used are equipped with an Intel Skylake[3] 3.40GHz CPU with 4 physical cores (8 hyper-threads), 32KB L1, 256KB L2, and 8MB shared L3 caches, 64 GB of RAM, and run a Docker container on top of Ubuntu 16.04 (Linux 4.4.0). The compilers we used are GCC 6.1.0, ICC 17.0.0, and Clang/LLVM 3.8.0. The complete experimental setup is described on the corresponding website.

4.1 Performance

To evaluate overheads incurred by Intel MPX, we tested the three benchmark suites on both the ICC and GCC implementations of MPX, as well as AddressSanitizer, SAFECode, and SoftBound. Because of the page limit, the paper presents only geomean averages (the first three bar groups on figures) together with a few peculiar outliers (three for each benchmark suite). The complete results can be found on the website.

Runtime overhead. We start with the single most important parameter: runtime overhead (see Figure 9).

First, we note that ICC-MPX performs significantly better than GCC-MPX. At the same time, ICC is less usable: only 30 programs out of total 38 (79%) build and run correctly, whereas 33 programs out of 38 (87%) work under GCC (see also §4.3).

AddressSanitizer, despite being a software-only approach, performs on par with ICC-MPX and better than GCC-MPX. This unexpected result testifies that the hardware-assisted performance improvements of MPX are offset by complicated design and suboptimal instructions.

SAFECode and SoftBound show good results on Phoenix programs, but perform much worse—both in terms of performance and usability—on PARSEC and SPEC. First, consider SAFECode on Phoenix: due to the almost-pointerless design and simplicity of Phoenix programs, SAFECode achieves a low overhead of 5%. However, it could run only 18 programs out of 31 (58%) on PARSEC and SPEC and exhibited the highest overall overheads. SoftBound executed only 7 programs on PARSEC and SPEC (23%). Moreover, both SAFECode and SoftBound showed unstable behavior: some programs had overheads of more than 20 ×.

[3]We have not tested Kaby Lake and Coffee Lake architectures as they do not introduce changes to Intel MPX.

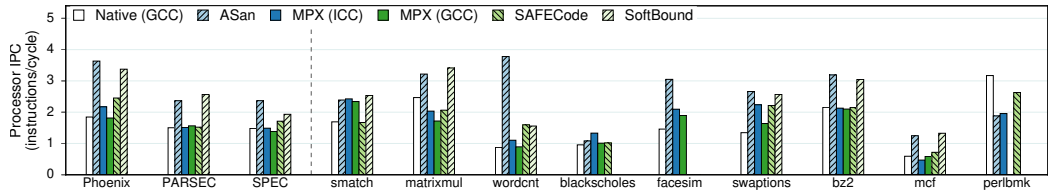

Fig. 11. IPC (instructions/cycle) numbers for native and protected versions. (Higher is better.)

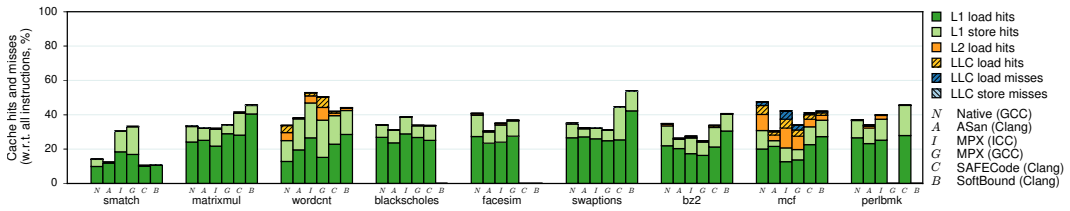

Fig. 12. CPU cache behavior of native and protected versions.

Instruction overhead. In most cases, performance overheads are dominated by a single factor: the increase in number of instructions executed in a protected application. This can be seen if we compare Figures 9 and 10; there is a strong correlation between them.

As expected, the optimized MPX (i.e., ICC version) has low instruction overhead due to its HW assistance (~70% lower than AddressSanitizer). Thus, one could expect sufficiently low performance overheads of Intel MPX once the throughput and latencies of Intel MPX instructions improve (see §6).

Instruction overhead of Intel MPX may also come from the management of BTs (see §3.2). However, we did not observe a noticeable impact in real-world applications. Even those applications that create hundreds of BTs exhibit a minor slowdown in comparison to other factors.

IPC. Many programs do not utilize the CPU execution-unit resources fully. For example, the theoretical IPC (instructions/cycle) of our machine is ~5, but many programs achieve only 1–2 IPC in native executions (see Figure 11). Thus, memory-safety techniques benefit from underutilized CPU and partially mask their performance overhead.

The most important observation here is that Intel MPX does not increase IPC. Our microbenchmarks (§3.1) indicate that this is caused by contention of MPX bounds-checking instructions on one execution port. If this functionality would be available on more ports, Intel MPX would be able to use instruction parallelism to a higher extent and the overheads would be lower—similarly to the software-based approaches. At the same time, software-only approaches—especially AddressSanitizer and SoftBound—significantly increase IPC, partially hiding their performance overheads.

Cache utilization. Some programs are memory-intensive and stress the CPU cache system. If a native program has many L1 or LLC cache misses, then the memory subsystem becomes the bottleneck. In these cases, memory-safety techniques can partially hide their performance overhead.

It can be illustrated with the *wordcnt* example compiled with ICC-MPX (Figure 12). It has a huge instruction overhead of 4×, IPC close to native, and (as we will see next) many expensive `bndldx` and `bndstx` operations. And still its performance overhead is only 3×. Why? It appears the native version of *wordcnt* has a significant number of cache misses. They have high performance cost and therefore can partially mask the overhead of ICC-MPX.

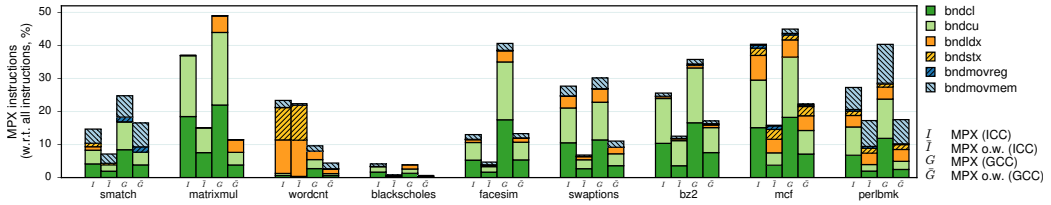

Fig. 13. Shares of Intel MPX instructions with respect to all executed instructions. (Lower is better.)

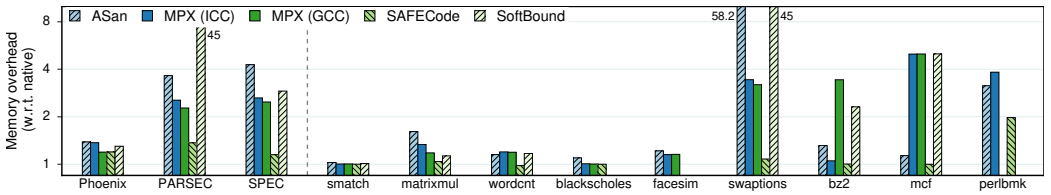

Fig. 14. Memory overhead with respect to native version. (Lower is better.)

Intel MPX instructions. For Intel MPX, one of the important performance factors is the type of instructions that are used in instrumentation. In particular, storing (`bndstx`) and loading (`bndldx`) bounds require two-level address translation — a very expensive operation that can break cache locality (see §3.1). To prove it, we measured the shares of MPX instructions in the total number of instructions of each program (Figure 13).

There is a strong correlation between the share of `bndstx` and `bndldx` instructions and performance overheads. For example, *matrixmul* under ICC-MPX contains almost exclusively bounds checks: accordingly, there is a direct mapping between instruction and performance overheads. However, the GCC-MPX version is less optimized and inserts many `bndldx`s, which leads to a significantly higher overhead.

The ICC-MPX version of *wordcnt* has a ridiculous share of `bndldx` and `bndstx` instructions. This is due to a performance bug in libchkp library of ICC that uses a naive bounds-copying algorithm for the `memcpy` wrapper (see §3.3).

Memory consumption. In some scenarios, memory overheads (more specifically, resident set size overheads) can be a limiting factor, e.g., for servers in data centers which co-locate programs and perform frequent migrations. Thus, Figure 14 shows memory overhead measurements.

On average, Intel MPX has a 2.1× memory overhead under ICC version and 1.9× under GCC. It is a significant improvement over AddressSanitizer (2.8×). There are three main reasons for that. First, AddressSanitizer changes the memory layout of allocated objects by adding "redzones" around each object. Second, it maintains a "shadow zone" that is directly mapped to main memory and grows linearly with the program's working set size. Third, AddressSanitizer has a "quarantine" feature that restricts the reuse of freed memory. On the contrary, Intel MPX allocates space only for pointer-bounds metadata and has an intermediary Bounds Directory that trades lower memory consumption for longer access time. Interestingly, SAFECode exhibits even lower memory overheads because of its pool-allocation technique. Unfortunately, low memory consumption does not imply good performance.

Influence of additional Intel MPX features. Figure 15 shows performance impact of two main security features of Intel MPX (§3.3). *Bounds narrowing* increases security level but may harm performance. However, the impact appears to be minor because the number of checks is not

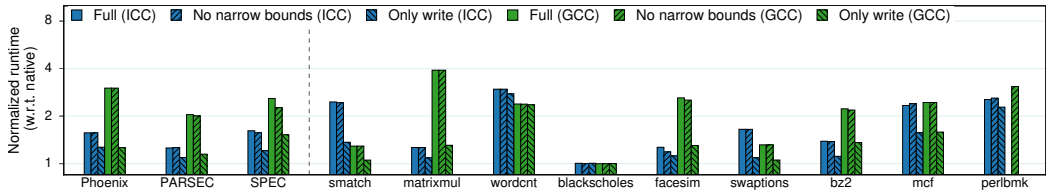

Fig. 15. Impact of MPX features—narrowing and only-write protection—on performance. (Lower is better.)

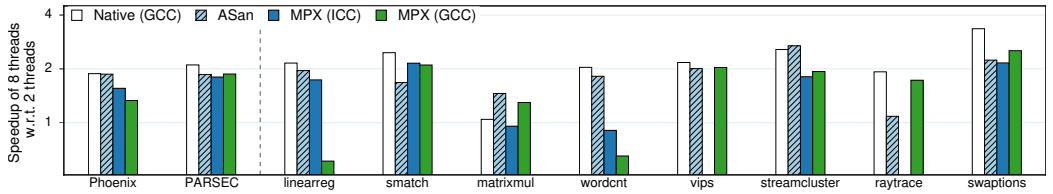

Fig. 16. Relative speedup (scalability) with 8 threads compared to 2 threads. (Higher is better.)

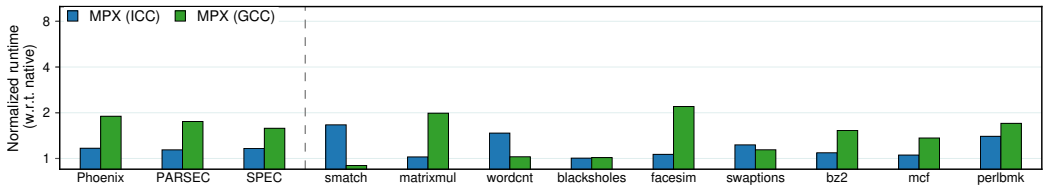

Fig. 17. Performance (runtime) overhead with respect to native version when MPX instructions are executed as NOPs. (Lower is better.)

changed. *Only-write protection*, on the other side, improves performance by removing checks on memory reads, having to instrument less code.

Multithreading. To evaluate multithreading, we measured execution times of all benchmarks on 2 and 8 threads (Figure 16). Note that only Phoenix and PARSEC are multithreaded (SPEC is not). Also, both SoftBound and SAFECode are not thread-safe and therefore were excluded from measurements.

Figure 16 shows that the difference in scalability is minimal. For Intel MPX, it is caused by the absence of multithreading support, which means that no additional code is executed in multithreaded versions. For AddressSanitizer, there is no need for explicit synchronization—the approach is thread-safe by design.

Peculiarly, GCC-MPX experiences not speedups but slowdowns on *linearreg* and *wordcnt*. Upon examining these cases, we found out that this anomaly is due to detrimental cache line sharing of BT entries.

For *swaptions*, AddressSanitizer and Intel MPX scale significantly worse than native. It turns out that these techniques do not have enough spare IPC resources to fully utilize 8 threads in comparison to the native version (the problem of hyperthreading). Similarly, for *streamcluster*, Intel MPX performs worse than AddressSanitizer and native versions. This is again an issue with hyperthreading: Intel MPX instructions saturate IPC resources on 8 threads and thus cannot scale as well as native.

Approach	Working attacks
MPX (GCC) default *	**41/64** (all memcpy and intra-object of.)
MPX (GCC)	**0/64** (none)
MPX (GCC) no narrow	**14/64** (all intra-object overflows)
MPX (ICC)	**0/34** (none)
MPX (ICC) no narrow	**14/34** (all intra-object overflows)
AddressSanitizer (GCC)	**12/64** (all intra-object overflows)
SoftBound (Clang)	**14/38** (all intra-object overflows)
SAFECode (Clang)	**14/38** (all intra-object overflows)

*`BNDPRESERVE=0` and w/o `-fchkp-first-field-has-own-bounds`

Table 5. Results of RIPE security benchmark. In Col. 2, "41/64" means that 64 attacks were successful in native GCC version, and 41 attacks remained in MPX version.

Performance overhead on Haswell CPU that does not support Intel MPX. MPX-protected applications can be executed even on older Intel CPUs that do not support Intel MPX (this feature allows to distribute the same binaries on old and new architectures). In this case, MPX instructions are executed as NOPs and, clearly, no memory-safety protection is provided. Yet, NOPs are not free—each NOP takes 1 cycle to execute and occupies space in caches, in the instruction pipeline, etc. It means that when running on old CPUs, the MPX-instrumented application is still slowed down. To evaluate this effect, we run the same set of benchmarks on a Haswell machine. As Figure 17 shows, ICC-MPX and GCC-MPX introduce the overheads of 15% and 60 – 90% respectively.

4.2 Security

RIPE testbed. We evaluated all approaches against the RIPE security testbed [56]. RIPE is a synthesized C program that tries to attack itself in a number of ways, by overflowing a buffer allocated on the stack, heap, or in data or BSS segments. RIPE can imitate up to 850 attacks, including shellcode, return-into-libc, and return-oriented programming. In our evaluation, even under relaxed security flags—we disabled Linux ASLR, stack canaries, and fortify-source and enabled executable stack—modern compilers were susceptible only to a small number of attacks. 64 attacks worked under native GCC, 34 under ICC, and 38 under Clang.[4]

The results for all approaches are presented in Table 5. Surprisingly, a default GCC-MPX version showed very poor results, with 41 attacks (or 64% of all possible attacks) succeeding. As it turned out, the default GCC-MPX flags are sub-optimal. First, we found a bug in the `memcpy` wrapper which forced bounds registers to be nullified, so the bounds checks on `memcpy` were rendered useless (see Table 3). This bug disappears if the `BNDPRESERVE` environment variable is manually set to one. Second, the MPX pass in GCC does *not* narrow bounds for the first field of a struct by default, in contrast to ICC which is stricter. To catch intra-object overflows happening in the first field of structs one needs to pass the `-fchkp-first-field-has-own-bounds` flag to GCC. When we enabled these, all attacks were prevented; all next rows in the table were tested with these flags.

Other RIPE results were expected. Intel MPX versions without narrowing of bounds overlook 14 intra-object overflow attacks, where a vulnerable buffer and a victim object live in the same struct. The same attacks are overlooked by AddressSanitizer, SoftBound, and SAFECode. We performed the same experiment with only-writes versions of these approaches, and the results were exactly the same. This is explained by the fact that RIPE constructs only control-flow hijacking attacks and not information leaks (which could escape only-writes protection).

[4]RIPE is specifically tailored to GCC, thus more attacks are possible under this compiler than under others.

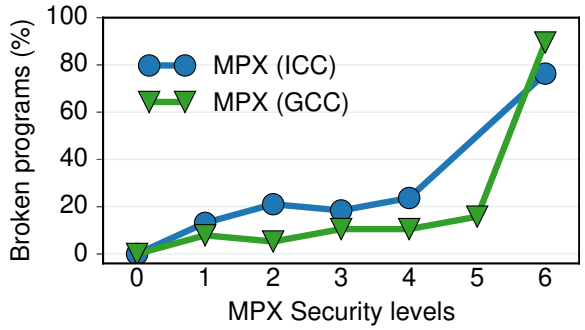

Fig. 18. Number of MPX-broken programs rises with stricter Intel MPX protection rules (higher security levels).

Other detected bugs. During our experiments, we found 6 real out-of-bounds bugs. Five of these bugs were already known, and one was detected by GCC-MPX and was not previously reported (refer to the website for the list of bugs).

All bugs were detected by GCC-MPX with narrowing of bounds. Predictably, three intra-object bugs and one read-only bug could not be detected by the no-narrowing and only-writes versions of Intel MPX respectively. ICC-MPX detected only 3 bugs in total: in other cases programs failed due to MPX-related issues (see §3.3 and §3.4). AddressSanitizer found only three of these bugs—it checks bounds at the level of whole objects and cannot detect intra-object overflows. SAFECode found two bugs and SoftBound—none of them (due to compiler bugs).

4.3 Usability

As we show in §3.4, some programs break under Intel MPX because they use unsupported C idioms or outright violate the C standard. Moreover, as shown in §3.3, other programs even fail to compile or run due to internal bugs in the compiler MPX passes (one case for GCC and 8 for ICC).

Figure 18 highlights the *usability* of Intel MPX, i.e., the number of MPX-protected programs that fail to compile correctly and/or need significant code modifications. Note that many programs can be easily fixed (Table 4, row 1); we do not count them as broken. Intel MPX *security levels* are based on our classification and correspond to the stricter protection rules, where level 0 means unprotected native version and 6 the most secure MPX configuration (see Table 7).

Encountered issues. Figure 19 presents an overview of the issues we encountered during our experiments.

We have not encountered any usability issues with AddressSanitizer: By design, it makes no assumptions on the C standard with respect to the memory model. Also, it is a supported product, fixed and updated with each new version of GCC and Clang.

On the contrary, SoftBound and SAFECode are research prototypes. They work perfectly with very simple programs from Phoenix, but are not able to compile/run correctly the more complicated benchmarks from PARSEC and SPEC. Moreover, SoftBound does not support multithreading, and any multithreaded program immediately fails under it.

Both GCC-MPX and ICC-MPX break most programs on Level 6 (with `BNDPRESERVE=1`). This is because `BNDPRESERVE` does not clear bounds on pointers transferred from/to unprotected legacy libraries. This means that any pointer returned from or modified by any legacy library (including C standard library) will almost certainly contain wrong bounds. Because of this, 89% of GCC-MPX and 76% of ICC-MPX programs break. These cases are represented as gray boxes.

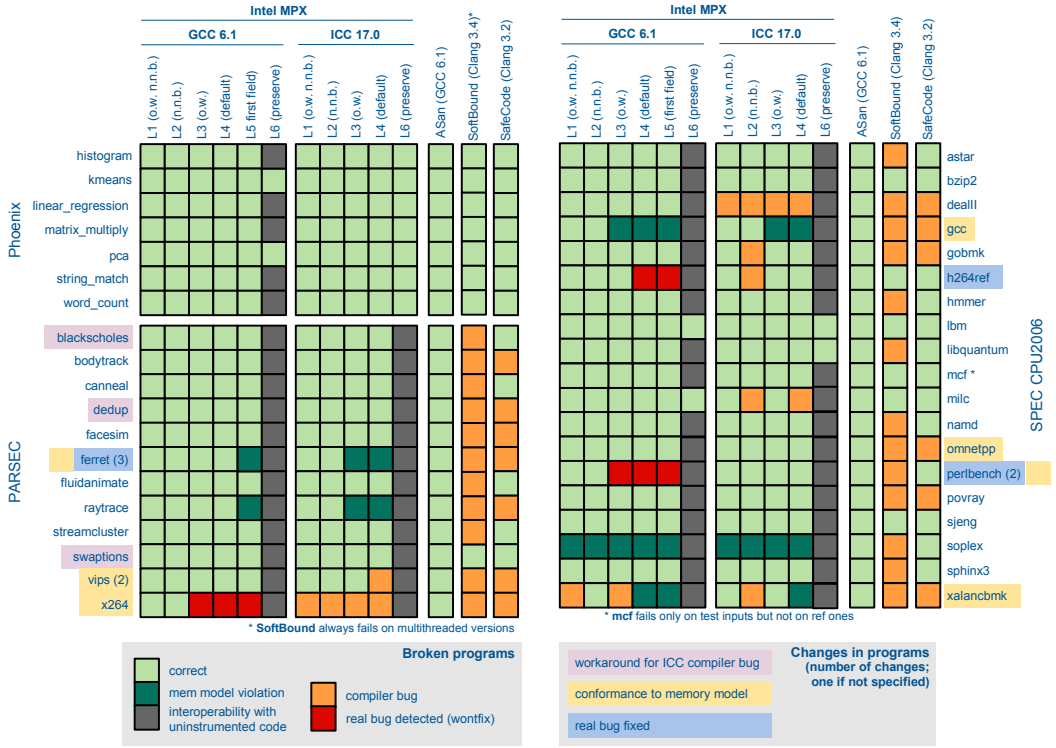

Fig. 19. All changes made to the programs under test as well as reasons why some programs break at compile- or run-time.

Note that for Phoenix, GCC-MPX fails in most cases while ICC-MPX works correctly. This is because of a slight difference in libc wrappers: all the failing programs use mmap64 function which is correctly wrapped by ICC-MPX but ignored by GCC-MPX. Thus, in the GCC case, the newly allocated pointer contains no bounds which (under `BNDPRESERVE=1`) is treated as an out-of-bounds violation.

One can wonder why some programs still work even if interoperability with C standard library is broken. The reason is that programs like kmeans, pca, and lbm require literally no external functions except malloc, memset, free, etc.—which are provided by the wrapper MPX libraries.

Some programs break due to *memory model violation*:

- ferret and raytrace both have structs with the first field used to access other fields in the struct (a common practice that is actually disallowed by the C standard). ICC-MPX disallows this behavior when bounds narrowing is enabled. GCC-MPX allows such behavior by default and has a special switch to tighten it (`-fno-chkp-first-field-has-own-bounds`) which we classify as Level 5.
- gcc has its own complex memory model with bit-twiddling, type-casting, and other practices deprecated by the C standard.
- soplex manually modifies pointers-to-object from one address to another using pointer arithmetic, without any respect towards pointer bounds. By design, Intel MPX cannot circumvent this violation of the C standard. (The same happens in mcf but only in one corner-case on test input.)

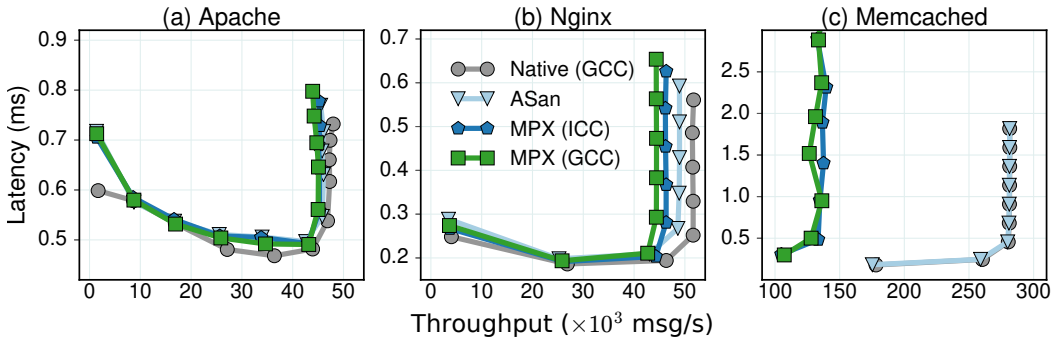

Fig. 20. Throughput-latency for (a) Apache web server, (b) Nginx web server, and (c) Memcached caching system.

- xalancbmk performs a container-style subtraction from the base of a struct. This leads to GCC-MPX and ICC-MPX breaking when bounds narrowing is enabled.
- We also manually fixed some memory-model violations, e.g., flexible arrays with size 1 (`arr[1]`). These fixes are represented as yellow background.

In some cases, real bugs were detected (see also §4.2):

- Three bugs in ferret, h264ref, and perlbench were detected and fixed by us. These fixes are represented as blue background.
- Three bugs in x264, h264ref, and perlbench were detected only by GCC-MPX versions. These bugs are represented as red boxes. Note that ICC-MPX missed bugs in h264ref and perlbench. Upon debugging, we noticed that ICC-MPX narrowed bounds less strictly than GCC-MPX and thus missed the bugs. We were not able to hunt out the root cause, but presume it is due to different memory layouts generated by GCC and ICC compilers.

In rare cases, we hit compiler bugs in GCC and ICC:

- GCC-MPX had only one bug, an obscure "fatal internal GCC compiler error" on only-write versions of xalancbmk.
- ICC-MPX has an autovectorization bug triggered on some versions of vips, gobmk, h264ref, and milc.
- ICC-MPX has a "wrong-bounds through indirect call" bug triggered on some versions of x264 and xalancbmk.
- ICC-MPX has a bug we could not identify triggered on dealII.
- We also manually fixed all manifestations of the C99 VLA bug in ICC-MPX. These bugs are represented as pink background.

5 CASE STUDIES

To understand how Intel MPX affects complex real-world applications, we experimented with three case studies: Apache and Nginx web servers and Memcached memory caching system. We evaluated these programs along three dimensions: performance and memory overheads, security guarantees, and usability.

We compare default Intel MPX implementations of both GCC and ICC against the native version, as well as AddressSanitizer. We were not able to compile any of the case studies under SoftBound and SAFECode: in most cases, the Configure scripts complained about an "unsupported compiler", and in one case (Apache under SoftBound) the compilation crashed due to an internal compiler

	Apache	Nginx	Memcached
Native	9.4	4.3	73
MPX	**120**	18	**352**
ASan	33	**380**	95

Table 6. Memory usage (MB) for peak throughput. (GCC-MPX and ICC-MPX showed identical results.)

error. The native version we chose to show is GCC: native ICC and Clang versions have almost identical results, with an exception of Nginx explained later. For the same reasons, we show only the GCC implementation of AddressSanitizer.

All experiments were performed on the same machines as for other experiments (see §4). One machine served as a server and a second one as clients, connected with a 1GB Ethernet cable and an actual bandwidth of 938 Mbits/sec. We configured all case studies to utilize all 8 cores of the server (details below). For other configuration parameters, we kept their default values.

All three programs were linked against their dependent libraries statically. We opted for static linking to investigate the complete overhead of all components constituting each program.

5.1 Apache Web Server

For evaluation, we used Apache version 2.4.18 linked against OpenSSL 1.0.1f [48]. This OpenSSL version is vulnerable to the infamous Heartbleed bug which allows the attacker to leak confidential information such as secret keys and user passwords in plain-text [49]. Since both AddressSanitizer and Intel MPX do not support inline assembly, we disabled it for all builds of Apache. To fully utilize the server, we used the default configuration of Apache's MPM event model.

The classic ab benchmark was run on a client machine to generate workload, constantly fetching a static 2.3K web-page via HTTP, with a KeepAlive feature enabled. To adapt the load, we increased the number of simultaneous requests at a time.

Unfortunately, while testing against Heartbleed, we discovered that ICC-MPX suffers from a run-time Intel compiler bug [5] in the `x509_cb` OpenSSL function, leading to a crash of Apache. This bug triggered only on HTTPS connections, thus allowing us to still run performance experiments on ICC-MPX.

Performance. As Figure 20a shows, GCC-MPX, ICC-MPX, and AddressSanitizer all show minimal overheads, achieving 95.3%, 95.7%, and 97.5% of native throughput. Overhead in latency did not exceed 5%. Such good performance is explained by the fact that our experiment was limited by the network and not CPU or memory. (We observed around 480 – 520% CPU utilization in all cases.)

In terms of memory usage (Table 6), AddressSanitizer exhibits an expected 3.5× overhead. In contrast, Intel MPX variants have dramatic 12.8× increase in memory consumption. This is explained by the fact that Apache allocates an additional 1MB of pointer-heavy data per each client, which in turn leads to the allocation of many Bounds Tables.

Security. For security evaluation, we exploited the infamous Heartbleed bug [49, 52]. In a nutshell, Heartbleed is triggered when a maliciously crafted TLS heartbeat message is received by the server. The server does not sanity-check the length-of-payload parameter in the message header, thus allowing `memcpy` to copy the process memory's contents in the reply message. In this way, the attacker can read confidential memory contents.

[5] https://software.intel.com/en-us/forums/intel-c-compiler/topic/700550

AddressSanitizer and GCC-MPX detect Heartbleed[6].

5.2 Nginx Web Server

We tested Nginx version 1.4.0—the last version with a stack buffer overflow vulnerability [2]. Nginx was configured with the "autodetected" number of worker processes to load all cores and was benchmarked against the same ab benchmark as Apache.

To successfully run Nginx under GCC-MPX with narrowing of bounds, we had to manually fix a variable-length array `name[1]` in the `ngx_hash_elt_t` struct to `name[0]`. However, ICC-MPX with narrowing of bounds still refused to run correctly, crashing with a false positive in `ngx_http_merge_locations` function. In a nutshell, the reason for this bug was a cast from a smaller type, which rendered the bounds too narrow for the new, larger type. Note that GCC-MPX did *not* experience the same problem because it enforces the first struct's field to inherit the bounds of the whole object by default—in contrast to ICC-MPX which takes a more rigorous stance. For the following evaluation, we used the version of ICC-MPX with narrowing of bounds disabled.

Performance. With regards to performance (Figure 20b), Nginx has a similar behavior to Apache. AddressSanitizer reaches 95% of native throughput, while GCC-MPX and ICC-MPX lag behind with 86% and 89.5% respectively. Similar to Apache, this experiment was network-bound, with CPU usage of 225% for native, 265% for Intel MPX, and 300% for AddressSanitizer. (CPU usage numbers prove that HW-assisted approaches impose less CPU overheads.)

As a side note, Nginx has predictable behavior only under GCC. Native ICC version reaches only 85% of the GCC's throughput, and native Clang only 90%. Even more surprising, the ICC-MPX version performed 5% *better* than native ICC; similarly, the AddressSanitizer-Clang version was 10% *better* than native Clang. We are still investigating the reasons for this unexpected behavior.

As for memory consumption (Table 6), the situation is opposite as with Apache: Intel MPX variants have a reasonable 4.2× memory overhead, but AddressSanitizer eats up 88× more memory (it also has 625× more page faults and 13% more LLC cache misses). Why then Intel MPX is slower than AddressSanitizer if their memory characteristics indicate otherwise? The reason for the horrifying AddressSanitizer numbers is its "quarantine" feature—AddressSanitizer employs a special memory management system which avoids re-allocating the same memory region for new objects, thus decreasing the probability of temporal bugs (use-after-free). Instead, AddressSanitizer marks the used memory as "poisoned" and requests new memory chunks from the OS (this explains huge number of page faults). Since native Nginx recycles the same memory over and over again for the incoming requests, AddressSanitizer experiences huge memory blow-up. When we disabled the quarantine feature, AddressSanitizer used only 24MB of memory.

Note that this quarantine problem does not affect performance. Firstly, Nginx is network-bound and has enough spare resources to hide this issue. Secondly, the rather large overhead of allocating new memory hides the overhead of periodically requesting new chunks from the OS.

Security. To evaluate security, the bug under test was a stack buffer overflow CVE-2013-2028 that can be used to launch a ROP attack [54]. Here, a maliciously crafted HTTP request forces Nginx to erroneously recognize a signed integer as unsigned. Later, a `recv` function is called with the overflown size argument and the bug is triggered.

Perhaps surprisingly, AddressSanitizer detects this bug, but both versions of Intel MPX *do not*. The root cause is the run-time wrapper library: AddressSanitizer wraps *all* C library functions

[6]The actual situation with Heartbleed is more contrived. OpenSSL uses its own memory manager which partially bypasses the wrappers around `malloc` and `mmap`. Thus, in reality memory-safety approaches find Heartbleed only if the length parameter is greater than 32KB (the granularity at which OpenSSL allocates chunks of memory for its internal allocator) [51].

including `recv`, and the wrapper—not the Nginx instrumented code—detects the overflow. In case of both GCC-MPX and ICC-MPX, only the most widely used functions, such as `memcpy` and `strlen`, are wrapped and bounds-checked. That is why when `recv` is called, the overflow happens in the unprotected C library function and goes undetected by Intel MPX.

This highlights the importance of full protection—not only protecting the program's own code, but also writing wrappers around all unprotected libraries used by the program. Another interesting aspect is that this overflow bug is read-only and cannot be caught by write-only protection. No matter how tempting it may sound to protect only writes, one must remember that buffer-overread vulnerabilities will slip away from such low-overhead checking.

5.3 Memcached Caching System

Lastly, we experimented with Memcached version 1.4.15 [18]. This is the last version susceptible to a simple DDoS attack [9]. In all experiments, Memcached was run with 8 threads to fully utilize the server. For the client we used a memaslap benchmark from libmemcached with a default configuration (90% reads of average size 1700B, 10% writes of average size 400B). We increased the load by adapting the concurrency number.

After some vexing debugging experiences with Nginx and Apache, we were pleased to experience no issues instrumenting Memcached with GCC-MPX and ICC-MPX.

Performance. Performance-wise, Memcached turned out to be the worst case for Intel MPX (see Figure 20c). While AddressSanitizer performs on par with the native version, both GCC-MPX and ICC-MPX achieved only 48 – 50% of maximum native throughput.

In case of native and AddressSanitizer, performance of Memcached was limited by network. But it was not the case for Intel MPX: Memcached exercised only 70% of the network bandwidth. The memory usage numbers in Table 6 help understand the bottleneck of Intel MPX. While AddressSanitizer imposed only 30% memory overhead, both Intel MPX variants used 350MB of memory (4.8× more than native). This huge memory overhead broke cache locality and resulted in 5.4× more page faults and 10 – 15% LLC misses, making Intel MPX versions essentially memory-bound. (Indeed, the CPU utilization never exceeded 320%.)

Security. For security evaluation, we used a CVE-2011-4971 vulnerability [9]. In this denial-of-service attack, a specially crafted packet is received by the server and passed to the handler (`conn_nread`) which tries to copy all packet's contents into another buffer via the `memmove` function. However, due to the integer signedness error in the size argument, `memmove` tries to copy gigabytes of data and quickly segfaults. All approaches—AddressSanitizer, GCC-MPX, and ICC-MPX—detected buffer overflow in the affected function's arguments and stopped the execution.

6 FUTURE RESEARCH DIRECTIONS

Given the fact that Intel MPX is impractical for fine-grained memory safety, the hardware is often employed for unintended purposes, such as sandboxing and memory isolation [7, 26, 33, 42], or even hardware fault tolerance [39]. These techniques usually work with only a few bounds that isolate coarse-grained memory regions. They remove the overhead of loading/storing bounds and managing the bounds tables, making the usage efficient.

Nevertheless—given the importance of memory safety violations — there is an urgent need for a lightweight, practical hardware-assisted memory safety mechanism. In this section, we reexamine the design of Intel MPX and suggest a set of changes that can help to achieve this goal. Note, however, that the scope of the paper is Intel MPX evaluation and we consider the suggested changes only as directions for future work. We neither implement nor evaluate them.

Issue #1: Intel MPX instructions incur high overheads. High performance overheads primarily come from two sources. First, current processors execute bounds checking mostly sequentially. Second, loading/storing bounds registers involves costly two-level address translation. Together, these issues lead to substantial runtime overheads of ~50% even with all optimizations applied (in the ICC case). Moreover, even on older architectures, with all MPX instructions treated as NOPs, it still incurs surprisingly high 15% overhead in the best (ICC) case. (We measured this number on a Haswell machine.)

Proposal: Parallel bounds checks. We can fix the first issue relatively easy. If the bounds checking functionality is present on at least one more execution unit, CPU will execute the checks in parallel with the corresponding memory accesses and the overhead will be hidden. We can observe this effect in AddressSanitizer—high utilization of ILP significantly amortizes the cost of protection (see §4.1). Considering that GCC-MPX has similar instruction overhead to AddressSanitizer, we estimate that it will also have similar performance if the checks are parallelized (~50%). Accordingly, ICC version would be even better, and the slowdowns may drop lower than 20%.

Proposal: Bounds cache. Object bounds tend to have high temporal and spatial locality, which makes them perfect candidates for caching. However, in the current implementation, they share L1 cache with the application data, which causes more frequent cache misses. Introducing a third type of cache—bounds cache—in addition to the existing data and instruction caches would solve this issue.

Proposal: Compactly storing bounds in memory. We can further improve bounds management by introducing a more suitable data structure for storing bounds. The scheme used in the current version is based on a lookup trie, similarly to virtual address translation. Although tries are relatively fast, they incur high memory overheads if the stored data is sparse. In the worst case, if pointers are far enough from each other, each Bounds Table will contain only a single entry. Accordingly, each pointer will waste a whole page of physical memory.

A hash-table-based data structure would have much lower memory overhead, although handling hash collisions might lead to higher performance overheads. Therefore, we consider finding an optimal data structure as future work.

Issue #2: Intel MPX does not provide temporal safety. Currently, Intel MPX protects only against spatial (out-of-bounds accesses) but not temporal (dangling pointers) errors.

Proposal: Lock-and-key temporal protection. We can enforce temporal safety via the "lock-and-key" strategy (similar to CETS [36]). It works as follows. Each memory object has an identifier—*key*—which is unique and never reused. MPX associates each pointer with a pair of values: a key of the referent object and a pointer to the *lock* memory region. When the memory allocator creates an object, it writes its key to the address of the corresponding lock. Later, when it frees the object, it also clears the lock. Thus, the lock and key values of the pointers will match only if the referent object is still valid.

We can implement the lock-and-key protection without changes to the current Intel MPX instruction set. If Intel MPX becomes thread-safe (see Issue 3), it will be sufficient to make the comparison when the bounds are loaded/stored, i.e., it can be embedded in the `bndldx/stx`. Memory layout can also be left as-is if we restrict the key value space to 2^{16} values. In this case, we can store both locks and keys in the "Reserved" field of Bounds Tables—the lock address in the lower 48 bits and the key value in the higher 16 bits.

Issue #3: Intel MPX does not support multithreading transparently. An MPX-protected multithreaded program can have both false positives (false alarms) and false negatives (missed bugs and undetected attacks) if the application does not conform to C11 memory model or if the compiler

does not update bounds in atomic primitives. Until this issue is fixed—either at the software or the hardware level—Intel MPX cannot be considered safe in multithreaded environments.

Proposal: Atomic bound loads/stores. One option would be to embed the bound stores/loads into the memory accesses by introducing "secure" store and load instructions. However, it would significantly limit compiler optimizations even when the optimizations are safe. Sometimes compiler can statically prove that other threads cannot change the object bounds, e.g., when the object is thread-local. In this case, it is legitimate to reuse the loaded bounds and to make such optimizations as loop hoisting.

To address this issue, we propose an alternative approach. Our suggestion is to merge pairs `bndldx/bndstx-mov` and assure their atomic execution. The instruction decoder could detect a `bndldx` immediately followed by the corresponding `mov` in the instruction queue, and instruct the rest of execution to handle these instructions atomically. Accordingly, when the compiler can statically prove that the bounds were not changed since the previous load, it can treat the load as a separate instruction and may apply the corresponding optimizations. If it is not provable, however, the compiler must protect the memory access using this pattern—`bndldx/bndstx-mov`.

Alternative designs. The solutions we propose could solve some of the issues, but the primary source of overhead in Intel MPX—numerous additional instructions—will stay. An efficient solution would require a complete revision of the technology.

One interesting option is to use a capability-based system such as CHERI [57], which associates each memory region with a "capability"—a pair of bounds and a set of permission rights for the region. The approach is more transparent to the application and does not produce as much pressure on instruction caches and CPU pipelines, although it comes at the cost of higher hardware budget. Another avenue is the research on PUMP [3, 10]. Its tagged architecture provides a general approach to enforce low-level security policies (including memory safety) through extensions to the hardware and the entire system stack. Even though this architecture offers strong verifiable security guarantees, the unconventional redesign of the whole stack seems too intrusive to be adopted by commercial systems. In a similar vein, the Oracle Silicon Secured Memory (SSM) in SPARC M7 processors [40] provides another custom hardware design to achieve probabilistic memory safety. The SSM architecture relies on the comparison of hardware-enforced version numbers to the pointers and respective memory regions to detect memory safety violations. Finally, the AddressSanitizer group is developing a hardware-based version of their approach called Hardware-Assisted AddressSanitizer (HWASAN) [13]. Although it has much lower memory requirements than the software version, it suffers from the same security flaws; direct accesses to wrong objects are not detected.

It is imperative that the future research on hardware-assisted memory safety should look for a sweet spot to overcome the aforementioned limitations.

7 CONCLUSION

In this paper, we presented a detailed root cause analysis of problems in the Intel MPX architecture through a cross-layer dissection of the entire system stack. To put our findings into perspective, we also present an in-depth comparison of Intel MPX with three prominent types of software-based memory safety approaches. Lastly, based on our findings, we propose potential changes to the Intel MPX architecture to overcome these limitations. Table 7 summarizes the results of our study. For convenience, we introduce six *MPX security levels* to highlight the trade-offs between security, usability, and performance.

Lvl	Description	Detects	RIPE attacks GCC	RIPE attacks ICC	Unfound bugs GCC	Unfound bugs ICC	Broken GCC	Broken ICC	Perf (×) GCC	Perf (×) ICC
0	native program (no protection)	—	64	34	6	3	0	0	1.00	1.00
1	MPX only-writes and no narrowing of bounds	inter-object overwrites	14	14	3	0	3	5	1.29	1.18
2	MPX no narrowing of bounds	+ overreads	14	14	3	0	2	8	2.39	1.46
3	MPX only-writes and narrowing of bounds	all overwrites*	14	0	2	0	4	7	1.30	1.19
4	MPX narrowing of bounds (default)	+ all overreads*	14	0	0	0	4	9	2.52	1.47
5	+ `fchkp-first-field-has-own-bounds`*	+ all overreads	0	–	0	–	6	–	2.52	–
6	+ `BNDPRESERVE=1` (protect all code)	all overflows	0	0	0	0	34	29	–	–
	AddressSanitizer [46]	inter-object overflows	12	–	3	–	0	–	1.55	–

* except intra-object overflows through the first field of struct, level 5 removes this limitation (only relevant for GCC version)

Table 7. The summary table with our classification of Intel MPX security levels—from lowest L1 to highest L6—highlights the trade-off between security (number of unprevented *RIPE attacks* and other *Unfound bugs* in benchmarks), usability (number of MPX-*Broken* programs), and performance overhead (average *Perf* overhead w.r.t. native executions). AddressSanitizer is shown for comparison in the last row

We hope that our work will help researchers develop the future security extensions, and practitioners—to better understand the benefits and caveats of Intel MPX. We believe a hardware extension that balances the trade-offs between the MPX and Capability-based ISAs (e.g., CHERI [57]) will have a lasting impact on improving the security and reliability of systems. So far, our study has been acknowledged by computer architects, compiler teams, and software systems groups. Many bugs we found in the MPX system stack have already been fixed by both GCC and ICC compiler teams.

Details of the experiments. A detailed description of the experiments used in this study is published on our website: https://Intel-MPX.github.io. Additionally, source code of the entire experimental infrastructure is publicly available under: https://github.com/TUDInfSE/Intel_MPX_Explained.

Acknowledgments. We thank our anonymous reviewers and our shepherd Thomas Wenisch for their helpful comments. We would like to thank the developer of the GCC-MPX pass Ilya Enkovich, the authors of AddressSanitizer (Konstantin Serebryany and Alexander Potapenko), SoftBound (Santosh Nagarakatte and Milo Martin), and SAFECode (John Criswell) for the provided help with their tools and for answering our questions. We also thank Bohdan Trach, Sergei Arnautov, and Franz Gregor for their insightful reviews and comments.

This work was partly funded by the Federal Ministry of Education and Research of the Federal Republic of Germany (03ZZ0517A, FastCloud) and by European Union's Horizon 2020 research and innovation programme under grant agreement 690111 (SecureCloud).

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. 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)*.

[2] Andrew Alexeev. 2016. nginx: The Architecture of Open Source Applications. http://www.aosabook.org/en/nginx.html. Online; accessed August, 2017.

[3] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*.

[4] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th Conference on Programming Language Design and Implementation (PLDI)*.

[5] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*.

[6] The Tor Blog. 2017. Tor Browser 5.5a4-hardened is released. https://blog.torproject.org/blog/tor-browser-55a4-hardened-released. Online; accessed August, 2017.

[7] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*.

[8] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. 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)*.

[9] CVE details. 2011. Memcached bug: CVE-2011-4971. http://www.cvedetails.com/cve/cve-2011-4971. Online; accessed August, 2017.

[10] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. 2015. Architectural support for software-defined metadata processing. *ACM SIGARCH Computer Architecture News* (2015).

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

[12] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th Conference on Programming Language Design and Implementation (PLDI)*.

[13] Clang 7 documentation. 2018. Hardware-assisted AddressSanitizer Design Documentation. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html. Online; accessed May, 2018.

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

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

[16] Frank Eigler. 2016. Mudflap: pointer use checking for C/C++. https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging. Online; accessed August, 2017.

[17] Ilya Enkovich. 2016. Intel(R) Memory Protection Extensions (Intel MPX) support in the GCC compiler. https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler. Online; accessed August, 2017.

[18] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. In *Linux Journal*.

[19] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight Bounds Checking. In *Proceedings of the 2012 International Symposium on Code Generation and Optimization (CGO)*.

[20] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*.

[21] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* (2006).

[22] Intel Corporation. 2013. Introduction to Intel(R) Memory Protection Extensions. https://software.intel.com/en-us/Articles/introduction-to-intel-memory-protection-extensions. Online; accessed August, 2017.

[23] Intel Corporation. 2016. Intel(R) Memory Protection Extensions Enabling Guide. https://software.intel.com/en-us/Articles/intel-memory-protection-extensions-enabling-guide. Online; accessed August, 2017.

[24] Intel Corporation. 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[25] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *Proceedings of the 2002 Annual Technical Conference (ATC)*.

[26] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*.

[27] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-assisted Fault Tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*.

[28] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBounds: Memory Safety for Shielded Execution. In *Proceedings of the 2017 ACM European Conference on Computer Systems (EuroSys)*.

[29] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[30] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. 2013. 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 Conference on Computer and Communications Security (CCS)*.

[31] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. 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).
[32] Microsoft Research. 2016. Checked C. https://www.microsoft.com/en-us/research/project/checked-c/. Online; accessed August, 2017.
[33] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS).*
[34] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL).*
[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th Conference on Programming Language Design and Implementation (PLDI).*
[36] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM).*
[37] George C. Necula, Scott McPeak, Westley Weimer, George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured. In *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL).*
[38] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 Conference on Programming language design and implementation (PLDI).*
[39] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. Efficient Fault Tolerance using Intel MPX and TSX. In *Proceedings of 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*
[40] Oracle. 2017. Introduction to SPARC M7 and Silicon Secured Memory (SSM). https://swisdev.oracle.com/_files/What-Is-SSM.html. Online; accessed August, 2017.
[41] GCC Patches. 2018. Remove MPX support. https://gcc.gnu.org/ml/gcc-patches/2018-04/msg01225.html. Online; accessed May, 2018.
[42] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR xor X: Comprehensive Kernel Protection Against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys).*
[43] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA).*
[44] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceeding of the Network and Distributed System Security Symposium (NDSS).*
[45] Konstantin Serebryany. 2016. Discussion of Intel Memory Protection Extensions (MPX) and comparison with AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizerIntelMemoryProtectionExtensions. Online; accessed August, 2017.
[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 Annual Technical Conference (ATC).*
[47] Matthew S. Simpson and Rajeev K. Barua. 2013. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Software — Practice and Experience* (2013).
[48] The Apache software foundation. 2016. Apache HTTP Server Project. http://httpd.apache.org/. Online; accessed August, 2017.
[49] Synopsys. 2016. The Heartbleed Bug. http://heartbleed.com/. Online; accessed August, 2017.
[50] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the Symposium on Security and Privacy (SP).*
[51] Ted Unangst. 2014. Heartbleed vs malloc.conf. http://www.tedunangst.com/flak/post/heartbleed-vs-mallocconf. Online; accessed August, 2017.
[52] The Register. 2014. Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug. http://www.theregister.co.uk/2014/04/09/heartbleed_explained/. Online; accessed August, 2017.
[53] Victor van der Veen, Nitish Dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*
[54] VN Security. 2013. 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. Online; accessed August, 2017.
[55] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High System-Code Security with Low Overhead. In *Proceedings of the 2015 Symposium on Security and Privacy (SP).*
[56] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC).*

[57] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*.

[58] Yichen Xie, Andy Chou, and Dawson Engler. 2003. ARCHER : Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. *ACM SIGSOFT Software Engineering Notes* (2003).

[59] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *Proceedings of the 4th Conference on Hot Topics in Parallelism (HotPar)*.

Received February 2018; revised May 2018; accepted May 2018