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

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

Oleksii Oleksenko
TU Dresden
Dresden, Germany
Oleksii.Oleksenko@tu-dresden.de

Dmitrii Kuvaikii
TU Dresden
Dresden, Germany
Dmitrii.Kuvaikii@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

ABSTRACT
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.

A complete version of this work appears in the 2018 proceedings of the ACM on Measurement and Analysis of Computing Systems.

KEYWORDS
Memory safety; ISA extensions; Intel MPX

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 [14].

These memory-safety violations are the root cause of most reliability and security vulnerabilities in legacy software systems [13]. 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, 3, 7, 9, 12].

In our work, we concentrate on deterministic dynamic bounds-checking since it is widely regarded as the only way of defending against all memory safety attacks [13]. 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 see 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 [4, 5]. 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.

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 [13]. 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.

Our work presents the first detailed cross-layer dissection of the Intel MPX system stack, comprising the hardware, operating system, compilers, and applications [11]. It 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 [12], object-based — SAFeCode [3], and pointer-based — SoftBound [9]. Surprisingly, even though Intel MPX is a specially designed hardware-assisted approach, it is not faster than the software-based approaches.

2 LESSONS LEARNED

Our investigation reveals that although Intel MPX strives to solve an important problem, it is not yet practical because of the following fundamental issues.

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 approximately 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.

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.

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.

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 [2, 6, 8]. Meanwhile, we know of no successful attempts to use MPX for the original purpose of complete memory safety.

3 CONCLUSION

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 [15]) will have a lasting impact on improving the security and reliability of systems.

In the extended version of this abstract [11], we discuss and evaluate the aforementioned issues and 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. We provide even more details in an accompanying technical report [10]. Finally, we release the entire experimental infrastructure to reproduce the study on our website: https://Intel-MPX.github.io.

Acknowledgments. This work was partly funded by the Federal Ministry of Education and Research of the Federal Republic of Germany (Förderkennzeichen 03ZZ0517A, FastCloud).

REFERENCES