SIMPLEX: Repurposing Intel Memory Protection Extensions for Secure Storage

Matthew Cole(✉) and Aravind Prakash

Binghamton University, Binghamton, USA
{mcole8,aprakash}@binghamton.edu

Abstract. The last few decades have seen several hardware-level features to enhance security, but due to security, performance, and/or usability issues these features have attracted steady criticism. One such feature is the Intel Memory Protection Extensions (MPX), an instruction set architecture extension promising spatial memory safety at a lower performance cost due to hardware-accelerated bounds checking. However, recent investigations into MPX have found that is neither as performant, accurate, nor precise as software-based spatial memory safety. Given its ubiquity, we argue that it provides an under-utilized hardware resource that can be salvaged for security purposes. We propose SIMPLEX, an open-sourced library that re-purposes MPX registers as general purpose registers. Using SIMPLEX, we demonstrate securely storing sensitive information directly on the hardware (e.g. encryption keys). We evaluate for performance, and find that deployment is feasible in all but the most performance-intensive code, with amortized performance overhead as low as about 1%.

Keywords: Information hiding · Hardware security · Intel MPX

1 Introduction

Intel Memory Protection Extensions (MPX) is an instruction set architecture (ISA) extension for modern Intel processors providing spatial memory safety using compile-time intentions. MPX is comprised of three key components working in harmony: architectural support through a set of two configuration, one status, and four bounds registers; compile-time instrumentation; and run-time support integrated with the operating system. This run-time manages enabling and disabling CPU interpretation of MPX instructions through the configuration registers, sets up a pointer bounds lookup table for spilling more objects' bounds than four registers can hold, interprets error codes indicated in the status register, and coordinates with the operating system to handle memory management and error handling.

In practice, MPX is unusable in its intended form. It was intended to be performant, inter-operable with un-instrumented legacy code, and configurable for both debug and release environments without rewriting the source. However, Oleksenko et al. and Serebryany independently showed that MPX does not perform as well as software- and language-based memory safety, demonstrating a

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
H. P. Reiser and M. Kyas (Eds.): NordSec 2022, LNCS 13700, pp. 215–233, 2022.
https://doi.org/10.1007/978-3-031-22295-5_12

50% amortized performance overhead with good compiler optimizations, and a 400% worst-case performance overhead [24,32]. The GNU C Compiler (GCC) recently removed its `libmpx` library and eliminated the instrumentation code, while Linux recently removed its support for kernel compilation due to a lack of community interest in maintaining the code. In short, MPX never achieved widespread adoption as a memory safety tool as envisioned by its designers, even as its architectural resources remain on widely-deployed processors. Yet even a conservative estimate puts the number of MPX-supported deployments at 100s of millions worldwide, thus MPX is a ubiquitous – yet unused – resource.

In this paper, we leverage MPX as a general storage primitive–specifically for storage of security-sensitive data such as cryptographic keys. Our contribution is named Simplex, which is comprised of a library enabling introspection and manipulation of the MPX context, a minimalist runtime that avoids the overhead associated with the compiler-provided MPX runtime, a test suite verifying correctness, and evaluations demonstrating the practicality of Simplex. Furthermore, our contribution allows manipulation of the MPX context *even in the complete absence of support* for compiler instrumentation or an operating system's runtime.

The Simplex library provides all necessary runtime components and functions for instrumentation, and the MPX context is part of the broader XSAVE context, thus it is still saved and restored on context switches even though Linux formally removed all MPX support as of kernel version 5.6. Only a microcode update from Intel would break Simplex by removing the CPU's ability to interpret the MPX opcodes, however we do not believe that this is likely to occur because there are no extant attacks against a victim which do not also link to an operating system's runtime (i.e. the attack proposed by Dekel and Kasif [7]).

Because the ability to prevent disclosure is a valuable resource in security, we emphasize applications of Simplex for moving information out of main memory. For example, Hargreaves and Chivers, and Kazim et. al showed two different techniques for extracting encryption keys from main memory [13,16]. On the one hand, hiding data in the kernel is often impractical as it incurs performance overhead due to the expensive transitions between user and kernel modes. On the other hand, reserving registers (e.g. [18,20,21,34])) is undesirable for two reasons: (1) it removes a register from the allocation pool, which could impact performance due to sub-optimal register allocation [2], and (2) it affects interoperability when handwritten assembly or binaries not compiled using the modified compiler may accidentally access or modify the reserved register. Because Simplex uses the MPX bounds registers, and because the bounds registers are not used unless the application was also explicitly compiled with MPX support, we can ensure that no other code will access or modify the hidden data or pointer stored inside the bounds register.

Our evaluation shows that Simplex is practical, and confirms initial observations by Otterstad [27] and Oleksenko et al. [24] that the majority of MPX's performance cost comes from handling exceptions and interacting with the bounds lookup table within the runtime. We avoid this overhead because Simplex avoids

using the bounds lookup table by writing to the bounds registers directly using the `bndmk` instruction and reading from the bounds registers using the `bndmov` instruction to spill the contents into memory. We evaluated for performance in two different ways. First, we created three custom benchmark fixtures: (1) a microbenchmark testing the rate at which load and store operations can be completed to both the `%r15` general purpose register and the `%bnd0` MPX bounds register, (2) a macro-benchmark simulating information unhiding by traversing and combining two hidden half-buffers, and (3) implementations of a subset of the `string.h` header. Second, we compiled sandboxed versions of two SPEC CPU2017 benchmarks: 519.lbm, a particle-fluid simulation written in C, and 531.deepsjeng, a chess engine written in C++ to demonstrate practicality of moving key data into the MPX bounds registers from global memory. Finally, we evaluated for usability and correctness by modifying the OpenSSL Blowfish cipher, then running the included integration and unit test suites.

The remainder of the paper is structured as follows: We discuss the history of MPX and the reasons prohibiting its widespread adoption as a memory safety tool in Sect. 2.1. Next, we examine the problems in information hiding continuing to plague security researchers in Sect. 2.2. An overview of our threat model and necessary modifications to a compiler to support SIMPLEX appears in Sect. 3. We describe the implementation of the SIMPLEX library, and answer questions about MPX context behavior during common program behavior including multithreading and system process lifecycles in Sect. 4. We present our evaluation in Sect. 5, showing SIMPLEX is both sound and practical. Finally, we survey related work in Sect. 6 and conclude in Sect. 7.

2 Background

2.1 Intel MPX

In 2012, Intel introduced POINTERCHECKER, which provides bounds checking in the software layer through the Intel Composer XE development environment for C and C++ [9]. Recognizing the potential for greatly improved performance through hardware support, Intel moved much of the POINTER CHECKER functionality into MPX, announced in 2013 [14] and subsequently debuted in the Skylake architecture in 2015.

MPX is a combination of an instruction set extension, compiler and operating system support, and runtime library. It provides four new 128-bit bounds registers (`%BND0` through `%BND3`), each of which are split into an upper half and lower half which have the purpose of holding an upper bound and lower bound address. MPX also employs the `%BNDCFGx` register pair to hold user-space and kernel-space configuration, and a `%BNDSTATUS` register to hold status information in case of a bounds check failure. These additional registers are encompassed in the larger Intel64 context, shown in Fig. 1. Intel designed MPX with the overarching goal of compatibility with un-instrumented code and unextended architectures. Where an MPX-supported CPU encounters un-instrumented code, such as a vendor-provided library, program execution continues with the cost that the

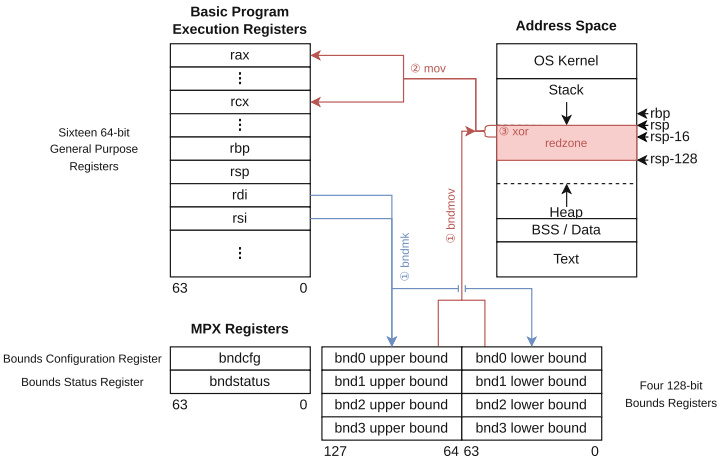

Fig. 1. The MPX context as part of the larger Intel64 context. The blue pathway shows how information is written to the bounds registers. The red pathway shows how information is read from the bounds registers (including sanitizing the stack afterwards). (Color figure online)

CPU can no longer provide memory safety because the bounds checks are not performed unless a `bndcl`, `bndcu`, or `bndcn` instruction is executed. Where an MPX-unsupported CPU encounters instrumented code, or when MPX has not been initialized by setting `%bndcfg[0]`, the instructions are interpreted as `nop` instructions instead of triggering unrecognized instruction exceptions.

Although MPX achieves a four- to five-fold speedup compared to Pointer Checker, it suffers prohibitive penalties of worst-case 200% performance overhead, 480% memory overhead, 5.4x more page faults [24], significant cache pressure, and a 50% slowdown even when bounds checking instructions are idempotent [32]. Furthermore, MPX cannot catch temporal memory safety issues [24], suffers false positives from otherwise legal C idioms due to restrictions on structure memory layouts [24,32] and false negatives in response to undefined behaviors which cause inappropriate bounds loads [27], conflicts with other Intel ISA extensions such as SGX and TSX [24], and it has no explicit support for multithreading [24]. As a result, support for MPX bounds checking has virtually ceased. Currently MPX's only remaining compiler support is Intel's own ICC since version 15.0 and Microsoft's Visual Studio 2015 Update 1.

2.2 Information Hiding

Information hiding techniques relying on probabilistic mechanisms can be defeated. Göktaş et al. demonstrated *thread spraying* [10] as a means of disclosing the safe regions with a known structure (e.g. the safe stack region used in [18]). By repeatedly creating objects that have both safe and regular allocations, then probing the space to find one of these hidden safe allocations,

they can effectively de-randomize the entire address space. They also discovered that information in the thread local storage (TLS) and the thread control block (TCB) provide clues to locating these stacks. Furthermore, Oikonomopoulos et al. introduced *allocation oracles* which eliminate the need for probing [23]. The idea is that an allocation oracle takes the size of an area to allocate as input, and if successful returns the location allocated. From this information and applying a binary search technique, an attacker can locate "holes" in the allocatable memory. If the attacker has knowledge of how a defense's sensitive data is laid out, then these holes reveal where the sensitive data *is not* hidden. With enough queries to the oracle, eventually the sensitive data can be located, and the process avoids crashes or distinguishable behavior usable by a runtime detector. Likewise, Evans et al. used timing side channels to read the contents of hidden metadata with or without crashes (the former is faster, the latter is difficult to detect) [8]. Using this technique, they can de-randomize the location of libraries such as libc, then use this to calculate the start of the safe region. Once complete, modifying the contents of the safe region permits an attacker to violate at least one implementation of Code Pointer Integrity (CPI).

State of the art defenses use registers to simulate segmentation as available in the IA-32 architecture in order to provide deterministic rather than probabilistic information hiding. One common point of these implementations is that they would benefit from dedicated registers. For example, two of the implementations of CPI require a dedicated register for information hiding [18]. In the reference implementation, `%fs` was reserved, however this may affect other legitimate usages of the register. For example, operating systems sometimes use this register to access TLS. Providing register storage via SIMPLEX helps return reserved general purpose registers to the compiler's allocation list and restores special purpose registers to their expected usage.

The dangers of storing secrets such as cryptographic keys in memory are also well known. For example, CERT Secure Coding Standard MEM06-C warns against writing sensitive data to disk, and Cold Boot Attacks [12,26] are a well explored vector when the key is located in DRAM. As a result, these secrets are often moved to un-swappable memory, such as registers or enclaves in order to maintain secrecy.

3 SIMPLEX

3.1 Threat Model

We assume a threat model similar to that offered by other work on information hiding, namely Koning [17] and Yun [33]. Our system under threat has an effective defense against code reuse, which in turn prevents an attacker from arbitrarily calling the SIMPLEX library functions, even though he or she may have an arbitrary read or write primitive. Although SIMPLEX might be used to store a pointer to a *hidden* memory region, it does not itself provide *isolation*. We presume that the programmer has a *Trusted Code Base* comprised of at least a privileged, trusted operating system and a trusted build toolchain used

to build the SIMPLEX library. We concede that an attacker may be able to load a Loadable Kernel Module (LKM) that enables or disables MPX at a privileged operating system level (and in fact, we provide one such implementation within the SIMPLEX code base). However, this would imply a compromised kernel, which is outside our scope. That said, we show in Sect. 3.4 that it is not sufficient for an attacker to emplace values into the bounds registers or leak values from the bounds registers in a way that is beneficial to the attacker. Finally, we assume that SIMPLEX is correctly implemented and is trusted by the programmer. We release our code as open source, and offer a full test suite within that code base as an assurance to that assumption.

3.2 Design Decisions

Previous works seeking to hide information from attackers have chosen one of three options. 1) Storing information in the kernel or in pages that can only be accessed in a privileged hardware mode (e.g. [11,15]) is secure as long as the operating system is not compromised. However these schemes come with the obligation of additional context switches for each query or update, hampering performance. 2) A more performant choice is storing information in a hidden region within the program's address space (e.g. [6,18,21]). Yet it relies on either probabilistic hiding measures which can be defeated if the attacker has knowledge of the type of information being hidden, or if the attacker is able to tolerate crashes and restarts while searching. 3) Alternatively, it is possible to reserve registers from the compiler's allocation pool and use these registers exclusively for storing sensitive data. Once the registers are selected, the defender can formally verify that no other code accesses these registers, guaranteeing security. Nonetheless, there is still the concern that available registers are limited and may conflict with other defenses or dynamically linked code that use the reserved register.

3.3 Simplex-Enabled Compilation

In our evaluations, we manually replaced global pointer objects and their reference/dereference statements with the necessary code to enable bounds register usage. However, we do not feel this is scalable. Consider the modifications made to the SPEC CPU2017 benchmarks: `519.lbm` has just 1 KLOC and required 22 modifications, `531.deepsjeng` has only 10 KLOC and required 173 modifications – these are very small code bases compared to `502.gcc` (1.3 MLOC) and `526.blender` (1.6 MLOC), the largest C/C++ benchmarks in CPU2017. Making these modifications are expensive in terms of developer effort and time, requiring both discovering and understanding the global variables' utilization. For example, modifying the two SPEC CPU2017 benchmarks took about two days of development time each. If the number and complexity of changes necessary were to scale, implementing the larger benchmarks by hand becomes infeasible. Therefore, we have designed but not yet implemented a system using Clang's annotation system to mark variables as candidates for placement in a bounds register. This reduces the developer's workload to simply recognizing

which variables should go into a bounds register, applying annotations to the declarations, then compiling the source code with the options necessary to enable SIMPLEX.

First, the developer applies the necessary annotation at the variable's declaration. The compiler recognizes the annotation, and maps that variable to one of the bounds registers, depending on its size or throws a compilation error if no more register space is available. Next, the compiler pass replaces references to these variables with appropriate SIMPLEX function calls. If the variable is an *lvalue*, it is replaced with a call to one of the mutator functions; if it is a *rvalue*, it is replaced with a call to one of the accessor functions.

Developer Annotation vs Automated Discovery: On the one hand, developer annotation has the benefit of precisely capturing what is of security relevance and importance as per software design, but on the other hand, developers are prone to make mistakes. Therefore, we recommend three modes of operation that make a trade off between security and performance.

Whitelisting: In this mode, we allow a developer to whitelist security-sensitive data that is stored in the MPX bounds registers by the compiler. This is the most conservative and performance-friendly, yet error-prone option.

Automatic Inference: In this mode, the compiler employs a heuristic approach to automatically profile and identify security sensitive information and accordingly provisions MPX bounds registers to manage such sensitive data. One option is to identify security-sensitive documented API functions and perform backward slicing to identify data of interest. This is the most aggressive option that favors security over performance.

Blacklisting: Finally, as an intermediate option, blacklisting allows a developer to define data items that *should not* be stored in the MPX registers. While blacklisting is just as prone to human error as is whitelisting, it is likely to have less adverse effects on security as compared to mistakes in whitelisting.

3.4 Context Behavior

Motivated by the desire to provide confidentiality between processes and/or threads – even when there is a relationship between the processes or threads – we explored the behavior of the MPX context. At process creation, the child inherits an identical MPX context to that of the parent because the MPX context is itself part of the larger CPU context (see Fig. 1).

Because SIMPLEX provides methods to initialize and finalize its minimal MPX context, the reader may question what would happen if a programmer or attacker called these methods repeatedly (whether by accident or malice). We found that each time the MPX context is initialized, the bounds registers' lower bounds are set to the system maximum unsigned value, and the upper bounds are set to 0. In MPX's design use case, this results in a guaranteed passed bounds check until the bounds register is set to some allocated object's bounds. In the SIMPLEX use case, repeated initialization destroys the values inside the bounds registers

by resetting them to the conservative bounds values. Although this may allow an attack against availability, it does not allow an attack seeking disclosure. Furthermore, it is no more dangerous for code-reuse attacks than the numerous `xor %reg %reg` instructions which are used by the compiler to place a zero value in a register.

4 Implementation

4.1 Components of Simplex

Unfortunately, there is no means of directly accessing the MPX bounds registers via a `mov` instruction. ICC does offer intrinsics, although these are only available if a MPX runtime is available and providing bounds checking [29]. This means it is not possible to use these intrinsics for accessing the bounds registers without also suffering the continual risk of a bounds check clobbering the bounds registers. Therefore, within Simplex we provide a system readiness check, a minimal runtime to enable and disable MPX execution, accessor and mutator functions, and a test suite to verify proper operation of the library.

System Readiness Check. Although it is possible for a user to test whether their system can support MPX from the command line using commands such as `lscpu` and `sysctl`, a program must be able to verify readiness itself and abort further execution if it cannot prove its readiness. This is because CPUs that do not support MPX will silently interpret these MPX instructions as `NOP`. We verify that `%CPUID[14]` is set (indicating that the CPU supports the MPX extension), and that `%XCR0[3:4]` are set (indicating that the CPU should include the MPX registers as part of a context save and restore) during initialization.

Enabling and Disabling Functions. We also provide a way of enabling and disabling MPX operations within both kernel mode and user mode applications. This can be done by setting flags on the `%BNDCFGS` and `%BNDCFGU` registers respectively. `%BNDCFGx[0]` enables interpretation of the MPX instruction extension, and `%BNDCFGx[1]` enables bounds register preservation when legacy instructions are encountered. Unlike the GCC run-time, we do *not* set `%BNDCFGx[63:12]` with the base address of the bounds table. This minimizes startup overhead, and also provides a small measure of security since accidentally attempting to access the bounds table will result in a segmentation fault rather than disclosing the contents of a bounds register.

Accessor and Mutator Functions. For each of the four bounds registers, a common accessor and mutator wrapper function provides a handle to the bounds register. There are four varieties of each wrapper function: lower-half 64 bits only, upper-half 64 bits only, all 128 bits, and a "quick" lower-half only which does not attempt to save the upper-half nor clean the stack of any spilled values. The applicable bounds register is selected through an enumerator with four values. When writing to the bounds registers, the value to be written is marshaled

from the function arguments into a `bndmk` instruction using sib-addressing. When accessing, the bounds register is atomically spilled onto the stack above the stack pointer (i.e. at a lower address than the top of the stack) then moved into a register because there is no bounds register-to-general purpose register instruction. This is accomplished using a `bndmov` instruction. As previously mentioned, all accessor functions except the quick variants will sanitize this value on the stack in case the value stored within the bounds registers is sensitive. We have verified that our extended assembly statements to perform the sanitation are not optimized by either GCC or Clang through manual inspection of disassembled code. See Fig. 1 for more information on data flows to and from the MPX context.

4.2 Security Impact of the SIMPLEX Implementation

Canella et al. recently reported a variety of Meltdown transient execution attacks, one of which is the Meltdown-BR (Bounds Check Bypass) attack [4,19]. Dekel also describes a post-exploitation technique called BoundHook, which allows an attacker to cause a bounds check exception in a user-mode context, and then catch the exception to gain control over the thread execution [7]. With both of these vulnerabilities, SIMPLEX does not increase a program's attack surface because both require a `#BR` exception to be raised in order to initiate exploitation. Since SIMPLEX does not use the `bndcl`, `bndcu`, or `bndcn` instructions, no such exception will be raised by our code. Additionally, because BoundHook requires that the attacker has also already compromised machine administrator rights, any attacker who can successfully execute a BoundHook intrusion can simply observe and modify the MPX context without the need to further compromise SIMPLEX.

Considerations for Multi-threaded Programs. Because SIMPLEX can be used in multi-threaded applications, we must address the dangers that an attacker-controlled thread could victimize a thread using SIMPLEX to interact with the MPX bounds registers during a brief period after spilling to the stack. We provide one mitigation in that SIMPLEX will zero out the memory used by the `bndmov` spill instruction immediately after copying to the destination register in all accessor functions except for `qgetbndl` which is performance- rather than security-optimized. We speculated an attacker-controlled victim thread or process with a pointer to the bottom of the stack could read this memory in a race condition assisted by a scheduler interrupt sometime between the spill from the bounds register to the time the stack memory is sanitized. Therefore we instrumented our library using a PAPI API [1] software defined event to measure the frequency of context switches within the SIMPLEX accessor functions and did not detect that such a sequence of events occurred. We hypothesize that this is because the accessor functions do not require any system calls and are very short-lived, and thus unlikely to trigger the scheduler's watchdog timer. Furthermore, we note that threads cannot directly access other threads' stacks, therefore the risk is limited to an attacker causing a process or thread to disclose its own bounds register values into shared memory during the window. We also note

that SIMPLEX spills outside the red zone, and therefore the compiler should not generate instructions that otherwise access this region without attacker input, and therefore such gadgets in the intended instruction stream are extremely rare.

5 Evaluation

We conducted our evaluation on an 8-core Intel Core i7-7700K CPU at 4.20 GHz with 62.8 GiB RAM running Ubuntu 20.04 LTS and the Linux 5.4 kernel. The system under evaluation conforms to POSIX.1-2017, and uses GNU libc and POSIX thread implementation version 2.27.

5.1 Benchmarks

We authored two benchmark fixtures to evaluate whether SIMPLEX attains performance that is comparable to using general purpose registers.

Load-Store Benchmark. First, we authored a micro-benchmark that tests load and store performance when SIMPLEX employs the `%bnd0` MPX bounds register compared to handwritten assembly using general purpose registers using `%r15`, segmentation registers using `%gs:0`, and the MMX and SSE instruction set extension registers using `%mm0` and `%xmm1` respectively, see Fig. 2. We find that the mean of writing to the MPX bounds registers is comparable to writing to the general purpose registers ($1.00x$), segmentation registers ($1.01x$), and MMX registers ($0.98x$). This is because all four of these operations have a fast, dedicated assembly instruction for writing to the register - either `mov` or `bndmk`. The fastest assembly instruction option for writing to the SSE registers is `movaps`, which moves four aligned, packed, single-precision floating point values to the register. However, it incurs significant overhead compared to the `mov` instruction because of microarchitectural limitations and thus the rate of MPX bounds register writes is $13.90x$ faster than these writes.

Loading from the MPX bounds registers is a different story. Additional overhead results because the MPX extension does not contain an instruction to move from a bounds register directly to another register, whether a bounds register or otherwise; `bndmov` only provides a bounds register to memory spill operation. Therefore load operations from a bounds register require that the data is first spilled to the quadword above the stack pointer through a `bndmov` instruction, then recovered through two additional `mov` instructions. General purpose register, segmentation register and MMX register loads can all be accomplished by a single `mov` instruction and thus MPX bounds register loads are only $0.74x$, $0.32x$, and $0.73x$ as fast, respectively. Segmentation register loads are particularly fast when repeatedly executed because of cache effects. Conversely, MPX bounds register loads are $1.69x$ faster than SSE register loads because these loads also must spill to stack, and because of the aforementioned micro-architectural limitations of the `apsmov` instruction.

Our findings also confirm the micro-architectural analysis of Oleksenko et al. [24] which found that it was not necessarily the MPX bounds operations that were particularly expensive, but rather the management of the bounds table through a two-level table lookup – particularly the `bndstx` and `bndldx` instructions. SIMPLEX uses neither of these instructions and thus avoids their associated performance overhead.

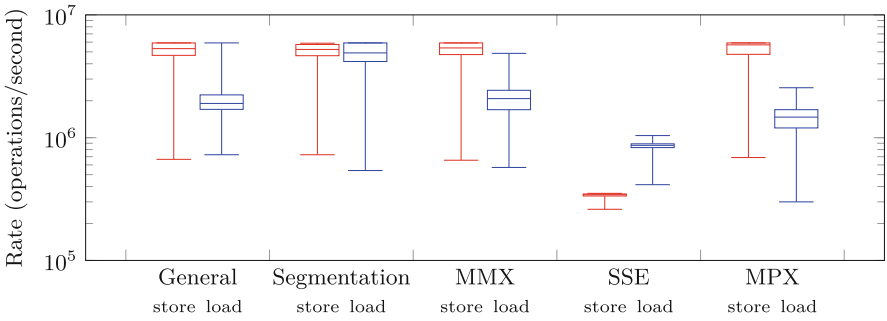

Fig. 2. Rate of load and store operations. Box and whisker plot shows median, minimum/maximum, and first/third quartile operation rates. We use `%r15` for *General*, `%gs:0` for *Segmentation*, `%mm0` for *MMX*, `%xmm1` for *SSE* and `%bnd0` for *MPX*. The test consisted of 10^4 runs, with 10^6 iterations per run. We report the steady-state rate of operations accomplished per second.

String Operations. We also evaluated the block memory operations from the `string.h` header using reference implementations of libgcc. We excluded the string-specific functions so that we could randomly fill buffers from test run-to-run without the concern of whether the buffer formed a single valid C string, and because our choice of functions does not include trivial functions that do not operate on buffers (e.g. `strerror`). We then refactored these functions for SIMPLEX to replace any passed argument that contains the address of a buffer with calls to instead load it from the corresponding bounds register. These benchmarks show that the performance cost of SIMPLEX is easily amortized, as we found that the maximum overhead was only 5.86%, and a 0.69% overall geometric mean. In the specific case of these function implementations, benchmarks that do not short-circuit (i.e. memcpy, memmove and memset) are able to amortize the cost fully compared to functions that do short-circuit (i.e. memcmp, memchr). We do not claim that there is a performance benefit to SIMPLEX, simply that if there is a performance cost, it is small enough to be unnoticeable to the user and that it is offset by the utility of the additional registers provided by SIMPLEX. We report specific data for each benchmark in Fig. 3.

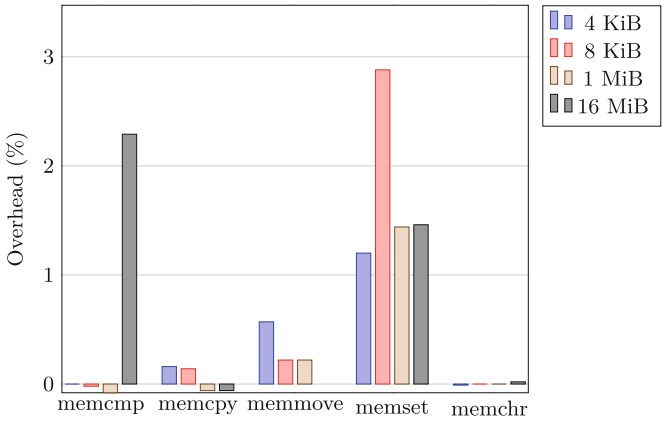

Fig. 3. String.h benchmarks' performance overhead when modified to pass pointer arguments in bounds registers. A negative percentage indicates the SIMPLEX-modified code ran in less time than the reference implementation.

5.2 Modifications to Existing Codebases

SPEC CPU2017. We hand-modified two SPEC CPU2017 benchmarks, `519.lbm` which simulates fluid flow through lattices, and `531.deepsjeng` which plays chess. In both cases, we selected the two global pointers to data structures that had the highest number of uses in order to fully stress the SIMPLEX library, showing an example of these modifications in Fig. 4. Although we selected global objects, it should be emphasized that SIMPLEX is not limited to just globals; heap or local objects could also be placed in the bounds registers. Using the SPEC benchmarks proves both correctness – the output is verified against a known correct output – and demonstrates performance cost of using SIMPLEX. We measured the performance rate ratio between runs with an unmodified benchmark and one where frequently used pointers to global variables were placed into a bounds register. This performance ratio was between 1.000 and 1.006 for `519.lbm`, and 0.975 and 0.985 for `531.deepsjeng` (see Table 1). Higher performance rate ratios indicate faster execution, but differ from performance overhead measurements since performance rate takes into account the number of threaded copies running simultaneously.

OpenSSL. We then modified the OpenSSL Blowfish symmetric key cipher to demonstrate how SIMPLEX might be used in a security application. In our modified Blowfish cipher, the address of the cipher's global key schedule structure is stored in a bounds register. Therefore wherever an encryption or decryption function would ordinarily receive a pointer to the key schedule as a function parameter, we instead pass a null value as the parameter and thus de-reference the bounds register at each usage of the parameter. Although the OpenSSL test suite provides test run time in its output, the Blowfish correctness test is very

```
 #include <sys/stat.h>
+#include "simplex.h"

-static LBM_GridPtr srcGrid, dstGrid;

void MAIN_initialize(const MAIN_Param* param) {
+       process_specific_init();

-       LBM_allocateGrid((double**) &srcGrid);
-       LBM_allocateGrid((double**) &dstGrid);
+       double* ptr;
+       LBM_allocateGrid(&ptr);
+       qsetbndl(BND0,(uint64_t) ptr);
+       ptr = 0;
+       LBM_allocateGrid(&ptr);
+       qsetbndl(BND1,(uint64_t) ptr);
+       ptr = 0;

-       LBM_initializeGrid(*srcGrid);
-       LBM_initializeGrid(*dstGrid);
+       LBM_initializeGrid(*((LBM_GridPtr)qgetbndl(BND0)));
+       LBM_initializeGrid(*((LBM_GridPtr)qgetbndl(BND1)));
}
void MAIN_finalize(const MAIN_Param* param) {
-       LBM_freeGrid((double**) &srcGrid);
-       LBM_freeGrid((double**) &dstGrid);
+       double* p0 = (double*) qgetbndl(BND0);
+       double* p1 = (double*) qgetbndl(BND1);
+       LBM_freeGrid(&p0);
+       p0 = 0;
+       LBM_freeGrid(&p1);
+       p1 = 0;

+       process_specific_finish();
}
```

Fig. 4. An example of modifications needed to store global pointers in bounds registers from the `lbm` benchmark. In this example, the global pointers `srcGrid` and `dstGrid` are placed in `BND0` and `BND1` respectively.

short in duration. As a result, our observed runtime overheads are smaller than the reported measurement resolution and not particularly useful as a metric of performance (see Table 2). We conclude that register repurposing presents minimal performance cost for cryptographic applications. We also emphasize that although we placed a pointer to a key schedule structure in the bounds registers for this evaluation, this structure is stored on the heap in the unmodified Blowfish cipher and therefore we have not introduced attack surface in our modified cipher. Additionally, some other OpenSSL ciphers' keys are less than 512 bits in size and would fit entirely within the bounds registers. The MPX bounds registers can hold any value, not just pointer values.

6 Related Work

Existing Evaluations. Explorations of Intel MPX generally find MPX to be flawed as a memory safety tool, and thus inspired our investigation as to whether MPX could be repurposed. Serebryany unfavorably evaluated the performance

Table 1. SIMPLEX SPEC CPU2017 evaluation data. *Run time* refers to how long the benchmark took to complete. *Base Rate* refers to the raw performance of this benchmark relative to the SPEC CPU2017 reference machine and thus provides insight into the underlying system under test. *Ratio* refers to the ratio of the modified benchmark's multi-threaded performance to the unmodified benchmark's multi-threaded performance. *Ratio* < 1 implies the modified benchmark ran slower than the unmodified benchmark.

Variables in bounds register	Copies	Run time	Base rate	Ratio
519.lbm_r				
None	1	202	5.21	
None	4	605	6.96	
srcGrid →bnd0	1	201	5.24	1.006
srcGrid →bnd0	4	605	6.96	1.000
srcGrid →bnd0, dstGrid →bnd1	1	202	5.23	1.004
srcGrid →bnd0, dstGrid →bnd1	4	606	6.96	1.000
531.deepsjeng_r				
None	1	283	4.04	
None	4	290	15.8	
state →bnd0, gamestate →bnd1	1	288	3.98	0.985
state →bnd0, gamestate →bnd1	4	297	15.4	0.975

Table 2. Simplex OpenSSL evaluation data. Measurements were obtained using `time(1)`, and presented columns reflect its output.

Variables in bounds register	usr	sys	cusr	csys	cpu
05-test_bf.t					
None	0.02	0.00	0.03	0.00	0.05
BF_KEY *schedule →bnd0	0.01	0.00	0.03	0.00	0.04
Overhead	−50.0%	0.0%	0.0%	0.0%	−20.0%

of Intel MPX versus the Address Sanitizer memory safety tool [31,32]. Notably, he discovered not only up to a $2.5x$ performance slowdown and $4.0x$ memory overhead on some benchmarks, but that the MPX instructions still exhibit a 50% slowdown even when they should be ignored on a system which does not have MPX support or has disabled it. He also identifies three categories of false positives that Address Sanitizer does not have: atomic pointers, un-instrumented bounds changes, and those caused by compiler optimizations after instrumentation. Otterstad examined the effectiveness of early implementations of MPX, identifying eight new categories of false positives and false negatives beyond those explored by Serebryany [27]. Furthermore, he demonstrates at least one toy program which can be victimized by ROP attacks because of these false positives and false negatives. Oleksenko et al. performed a study of the performance,

security guarantees, and usability issues of MPX after it became available in production hardware [24]. Furthermore, their empirical study was backed by an exhaustive investigation of how MPX is actually implemented at the hardware, operating system and software levels, supporting their experimental findings.

Other Uses of Intel MPX. We are not the only members of the community to propose repurposing MPX. Code Pointer Integrity (CPI) maintains a safe region to protect function pointers, return addresses and other pointers to code called a "safe stack" [18]. The authors propose one implementation of CPI using MPX to store the safe region's metadata, gaining performance benefits by moving some of the implementation into MPX's hardware accelerated checks. Burow further investigates using MPX to isolate CPI's shadow stacks and provide a highly-efficient implementation [3]. We note that SIMPLEX performs much of the management functionality they described, and could be used in conjunction with their defenses. Opaque Control-Flow Integrity (O-CFI) combines fine-grained code layout randomization with coarse-grained CFI in order to defeat sophisticated attacks seeking in-memory layout information to launch code-reuse attacks [21]. O-CFI uses MPX instructions to perform branch instrumentation, where legal branch targets are "chunked" together into a minimal address range, similar to a buffer. Oleksenko proposes a system combining MPX for hardware fault detection with Intel Transactional Synchronization Extensions (TSX) for fault rollback [25]. The underlying principle is that if a pointer's value is corrupted by a fault, then it will likely point to a dramatically different address outside the bounds of the referent object. MemSentry is a deterministic memory isolation framework addressing the threats of allocation oracles, thread spraying, crash-resistant memory disclosure primitives, and various side channels [17]. The authors use MPX and Intel Memory Protection Keys (MPK) to describe a more efficient method of intra-process isolation, similar to that provided by the kernel through `mprotect` and Software Fault Isolation (SFI). CFIXX is a C++ defense for virtual table pointers providing Object Type Integrity (OTI) [3]. CFIXX protects against corruption attacks against OTI by protecting the memory region containing the OTI metadata with selective MPX instrumentation. By reimagining the layout of the address space, they are able to halve the number of bound checks compared to a full memory safety solution provided by MPX. BOGO extends the MPX bounds tables to not only provide spatial memory safety, but also temporal memory safety [35]. Since MPX already initializes bounds table entries at allocation, BOGO additionally invalidates these entries upon deallocation and thus gains temporal memory safety. Since doing this operation at every deallocation can be expensive, the authors also introduce more efficient techniques for managing the deallocation metadata updates and for scanning the bounds table. DataShield provides three methods for coarse-grained bounds checks for non-sensitive pointer dereferences, one of which utilizes MPX to avoid the need to information hide the non-sensitive data regions [5]. Up to four of these regions' addresses are initialized in the MPX bounds register at program startup, with each pointer dereference in order to assure that the pointer does not escape the non-sensitive region. The Linux kernel can be protected against

Just-in-Time code reuse attacks by kR^X, which hardens benign read operations that an attacker might reuse to disclose code to find useful JIT gadgets [28]. Intel MPX is used in one implementation of kR^X to accelerate the execute-only range checks to reduce the performance overhead. The Spons & Shields Framework (SSF) for Intel SGX trusted execution environments uses the MPX bounds check instructions to verify memory accesses, however it does so outside of the traditional MPX runtime [30].

Repurposing Hardware Registers. The idea of repurposing hardware registers as with SIMPLEX is not unique. TRESOR is a patch that implements the AES encryption algorithm for the Linux kernel, and also provides additional security by utilizing the Intel AES-NI instruction set extension plus keeping encryption keys in the x86 debug registers instead of in RAM [22]. Ginseng keeps secrets in an encrypted secure stack until they are needed, then moves the secret into dedicated registers [33]. This has the effect of reducing the amount of sensitive data kept in the ARM TrustZone Trusted Execution Environment (TEE) and thus reduces the TEE's attack surface and does not require placing the operating system within the trusted computing base.

7 Conclusion

SIMPLEX is an open-source library repurposing the Intel MPX instruction set extension. We present evidence that suggests that MPX is ubiquitous, and show that MPX bounds registers can be repurposed as general purpose storage. In particular, they can be used to hide security sensitive data. We demonstrate that although the MPX ISA lacks a dedicated instruction to move data directly to and from the bounds registers, it is still possible to do so through the available spill and fill instructions, `bndmk` and `bndmov`. Furthermore, we show that such operations are not overly-burdensome, especially once the operations are amortized across the entire execution of a program. We do this through a collection of refactored programs and a partial implementation of the C standard library. Finally, we make SIMPLEX available to the community as open-source software at https://github.com/bingseclab/simplex.

References

1. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC 2000 (2000). https://doi.org/10.1109/SC.2000.10029
2. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2003, pp. 265–275. IEEE Computer Society (2003)

3. Burow, N., Mckee, D., Carr, S.A., Payer, M.: CFIXX: object type integrity for C++. In: Network and Distributed Systems Security Symposium 2018 (2018). https://doi.org/10.14722/ndss.2018.23279
4. Canella, C., et al.: A systematic evaluation of transient execution attacks and defenses. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 249–266 (2019). https://www.usenix.org/conference/usenixsecurity19/presentation/canella
5. Carr, S.A., Payer, M.: DataShield: configurable data confidentiality and integrity. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS 2017 (2017). https://doi.org/10.1145/3052973.3052983
6. Davi, L., Liebchen, C., Sadeghi, A.R., Snow, K.Z., Monrose, F.: Isomeron: code randomization resilient to (just-in-time) return-oriented programming (2015). https://doi.org/10.14722/ndss.2015.23262
7. Dekel, K.: BoundHook: exception based, kernel-controlled user-mode hooking (2017). https://www.cyberark.com/threat-research-blog/boundhook-exception-based-kernel-controlled-usermode-hooking/
8. Evans, I., et al.: Missing the point(er): on the effectiveness of code pointer integrity. In: 2015 IEEE Symposium on Security and Privacy, pp. 781–796 (2015). https://doi.org/10.1109/SP.2015.53
9. Ganesh, K.: Pointer checker: easily catch out-of-bounds memory accesses (2012). https://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf
10. Göktas, E., et al.: Undermining information hiding (and what to do about it). In: Proceedings of the 25th USENIX Conference on Security Symposium, pp. 105–119 (2016)
11. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: KASLR is dead: long live KASLR. In: Engineering Secure Software and Systems, pp. 161–176 (2017). https://doi.org/10.1007/978-3-319-62105-0_11
12. Halderman, J.A., et al.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009). https://doi.org/10.1145/1506409.1506429
13. Hargreaves, C., Chivers, H.: Recovery of encryption keys from memory using a linear scan. In: 2008 Third International Conference on Availability, Reliability and Security (2008). https://doi.org/10.1109/ARES.2008.109
14. Intel Corporation: Introduction to Intel Memory Protection Extensions (2013). https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions
15. Intel Corporation: Control-flow Enforcement Technology Specification, May 2019. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf
16. Kazim, A., Almaeeni, F., Ali, S.A., Iqbal, F., Al-Hussaeni, K.: Memory forensics: recovering chat messages and encryption master key. In: 2019 10th International Conference on Information and Communication Systems (ICICS), pp. 58–64 (2019). https://doi.org/10.1109/IACS.2019.8809179
17. Koning, K., Chen, X., Bos, H., Giuffrida, C., Athanasopoulos, E.: No need to hide: protecting safe regions on commodity hardware. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 437–452 (2017). https://doi.org/10.1145/3064176.3064217
18. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: Proceedings of the 11th USENIX Conference

on Operating Systems Design and Implementation, OSDI 2014, pp. 147–163. USENIX Association (2014). https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov
19. Lipp, M., et al.: Meltdown: reading kernel memory from user space. In: 27th USENIX Security Symposium, pp. 973–990 (2018). https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
20. Lu, K., Song, C., Lee, B., Chung, S.P., Kim, T., Lee, W.: ASLR-guard: stopping address space leakage for code reuse attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 280–291 (2015). https://doi.org/10.1145/2810103.2813694
21. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque control-flow integrity. In: Network and Distributed Systems Security Symposium 2015 (2015). https://doi.org/10.14722/ndss.2015.23271
22. Müller, T., Freiling, F.C., Dewald, A.: TRESOR runs encryption securely outside RAM. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011 (2011). https://doi.org/10.5555/2028067.2028084
23. Oikonomopoulos, A., Athanasopoulos, E., Bos, H., Giuffrida, C.: Poking holes in information hiding. In: 25th USENIX Security Symposium, Austin, TX, pp. 121–138 (2016)
24. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., Fetzer, C.: Intel MPX explained: an empirical study of Intel MPX and software-based bounds checking approaches (2017). https://doi.org/10.48550/ARXIV.1702.00719
25. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Fetzer, C., Felber, P.: Efficient fault tolerance using Intel MPX and TSX. In: Fast Abstract in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Toulouse, France (2016)
26. Ooi, J.G., Kam, K.H.: A proof of concept on defending cold boot attack. In: 2009 1st Asia Symposium on Quality Electronic Design (2009). https://doi.org/10.1109/ASQED.2009.5206245
27. Otterstad, C.W.: A brief evaluation of Intel MPX. In: 2015 Annual IEEE Systems Conference Proceedings, pp. 1–7. IEEE (2015). https://doi.org/10.1109/SYSCON.2015.7116720
28. Pomonis, M., Petsios, T., Keromytis, A.D., Polychronakis, M., Kemerlis, V.P.: kR^X: comprehensive kernel protection against just-in-time code reuse. In: Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017 (2017). https://doi.org/10.1145/3064176.3064216
29. Ramakesavan, S., Rodriguez, J.: Intel memory protection extensions enabling guide (2016). https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide
30. Sartakov, V.A., O'Keeffe, D., Eyers, D., Vilanova, L., Pietzuch, P.: Spons & shields: practical isolation for trusted execution. In: Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2021). https://doi.org/10.1145/3453933.3454024
31. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: 2012 USENIX Annual Technical Conference. pp. 309–318 (2012)
32. Serebryany, K.: Address sanitizer Intel memory protection extensions (2016). https://github.com/google/sanitizers/wiki/AddressSanitizerIntelMemoryProtectionExtensions

33. Yun, M.H., Zhong, L.: Ginseng: keeping secrets in registers when you distrust the operating system. In: Network and Distributed Systems Security Symposium 2019 (2019). https://doi.org/10.14722/ndss.2019.23327
34. Zhang, M., Sekar, R.: Control flow and code integrity for COTS binaries: an effective defense against real-world ROP attacks. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp. 91–100 (2015). https://doi.org/10.1145/2818000.2818016
35. Zhang, T., Lee, D., Jung, C.: BOGO: buy spatial memory safety, get temporal memory safety (almost) free. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, New York, NY, USA, pp. 631–644. Association for Computing Machinery (2019). https://doi.org/10.1145/3297858.3304017