CoMpk: Isolating Data in Private, and Secure Compartments

Arul T Sagayam
Virginia Tech

Jubayer Mahmod
Virginia Tech

Saim Ahmad
Virginia Tech

Abstract

Operating systems provide inter-process isolation by virtualizing the memory allocated to each process, but intra-process isolation still remains a concern for developers to this day. An attacker may utilize untrusted parts of the application to gain access to private data within the virtual memory of a process. Hence, techniques that isolate application data from untrusted parts of the application itself are essential to ensure data safety. Such a technique should guarantee data isolation amongst different functioning parts of an application and incur a minimal run-time overhead in contrast with the unmodified, vanilla application.

We propose CoMpk, an isolation technique that builds on existing techniques to provide a solution that is both, secure and efficient. We utilize static analysis techniques along with Intel x86 Memory Protection Keys (MPKs) to isolate data in the virtual memory of a process from certain parts of the application that are not trusted, and only allow access to user-defined sections of code termed as private functions. We aim to design a leak-proof process with minimal privilege switching.

1 Motivation

For application security, it is good practice to isolate data from parts of the application that one may consider untrusted—parts of an application that can be utilized by a malicious attacker to steal data or gain access to a system. A variety of user-space attacks exist that can allow access, local or remote, to a malicious user.

An attacker may gain access to a processes' memory via injected code or malicious libraries that a user includes as part of a program. The growth and abundance of libraries for different programming languages has made it easy for attackers to trick a user into believing that a library is safe to include with a program. This can cause theft of sensitive data such as public/private keys and passwords. Application operating over the network are subject to such attacks. Apart from this, malicious code included as part of a process can silently mine data on a host system such as credit card numbers, sensitive logs and passwords.

All the aforementioned attacks make use of the in-process memory to achieve their goals. This shows how important intra-process isolation can be when it comes to making systems more secure. Intra-process isolation and protection can greatly reduce the attack surface that malicious users can utilize to achieve their goals.

As part of our research, we study and survey multiple state-of-the-art techniques and systems that obtain inter-process memory isolation. Such techniques serve as a base for COMPK. Shreds [1] is a novel kernel abstraction that uses static analysis and memory-pools to secure application data from different parts of the application. Shreds are part of a program that can only access memory-pools assigned to them; hence, each shred has a very local view of the memory despite sharing the same virtual memory. Sensitive pieces of code that require data such as private keys or passwords are executed in different shreds. This prevents malicious code from running in the same process, not to see secrets stored as part of the in-process memory. Shreds also comes coupled with a compiler that performs data-flow analysis of secure segments of code at compiler time to ensure no secrets are leaked.

An idea to isolate data is to run different segments of a program in a different process. However, such an approach results in high overhead due to regular context-switching. Light-weight contexts [4] utilize this idea to make isolated snapshots of a process within a process itself, thereby eliminating the high overhead of context switching amongst different processes. *LwC*s provide APIs to create and switch between different snapshots of a process. Each *LwC* has its own virtual address space, thereby providing inter-process isolation of data.

ERIM [6] utilizes Intel memory protection hardware keys (MPKs) [2] to isolate data within the user-space process. However, unlike prior work such as [1] [4], ERIM [6] does not need to make switches to the kernel-space to isolate data, thereby eliminating switching overhead almost entirely. ERIM [1] utilizes protection keys and the PKRU register to assign pages to different groups, which can only be accessed if appropriate permissions are obtained. ERIM [6] provides safe and secure gateways to manipulate and change the PKRU register, which can be executed from the user-space (they provide this interface as a user-space library). Along with this, ERIM [1] provides an infrastructure for binary rewriting to ensure that attackers cannot leverage ROP gadgets within the binary to gain access to protected parts of the memory.

Lastly, we look into [5], a library that provides APIs to use memory protection keys [2] safely. There are multiple issues with the current implementation of the linux kernel's APIs to allocate, set, and free memory protection keys. One freeing a protection key, it is only removed from the key bitmap and not from the specific page table entry, meaning that on reallocating the freed key, it might have access to previous pages assigned to it. Furthermore, memory protection keys are restricted to only 16 groups of private data, out of which one is used my the OS, leaving the user-space applications with only 15 groups (*i.e.* domain). Hence, only 15 groups of addresses can be successfully isolated from each other. *libmpk* extends this to include as many groups as desired by the user program. In addition, synchronization is required among the different threads to ensure consistent permissions for different pages allocated to a key. We will utilize the *libmpk* [5] APIs for the development of our system as it provides a safe and secure interface for using memory protection keys [2].

2 Research Goals

The aim of this project is to develop a system that provides intra-process level data isolation. Our aim was to integrate ERIM [6], which provided data isolation using memory protection keys [2] with libmpk [5]. ERIM [6] is restricted by the number of private data groups it can form and hence using it with libmpk [5] would help us address the limited number of groups. However, ERIM [6] is entirely user-space based whereas [5] is both, user-space and kernel-space. Therefore, instead of integrating with ERIM [6]'s modules, we reuse some of the modules and develop an ERIM-like system with which we integrate libmpk [5]. Like ERIM [6], we develop a kernel module that serves as a hypervisor for applications that are running using our system.

ERIM [6] requires that the programmer be considerate of not leaking any sensitive data to the stack while executing in privileged mode. However, this requires that the programmer be vary every time he/she is modifying or writing code. We eradicate this need by introducing the concept of a private stack frame. We provide the programmer with an API to declare a private, reusable segment of code, termed as a private function. The programmer need not worry about leaking any data to the program's stack frame as the function's stack frame is allocated and de-allocated upon entry and exit from the function. Furthermore, the private function is backed by our compiler which is inspired from Shreds [1].

Our system uses an LLVM pass that performs intra-module analysis of the program to ensure and enforce that all private functions meet the requirements of our system. This includes performing basic control-flow and data-flow analysis on every private function declared by the programmer. We only require basic analysis of the function to determine if there is any data leakage. Furthermore, our control-flow analysis consists of checking whether the private function correctly sets mpks [2] upon entry and unsets them upon every return call from the function.

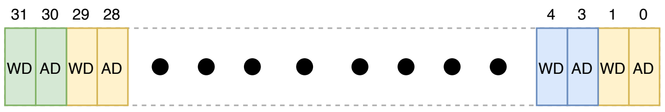


Figure 1: PKRU register

3 Background

CoMPK relies on architectural support, memory protection key, available in modern server processor such as Intel Xeon [3]. MPK allows tagging each virtual page with a hardware key (*pkeys*) and full user-mode permission change of a memory region. A key is bound to a group of pages, and privilege level is controlled by reading/writing from/to PKRU register (Figure 1). Processor architecture provides RDPKRU and WRPKRU instructions to read and write PKRU register, respectively. Note, WRPKRU is an unprivileged instruction that changes the permission of a key and the group of pages it is associated with. Conceptually, permission change is an indirect process. That is, a key is bound to a page or a group of pages by tagging the pages' PTEs[32:35]. Then permission operation is enforced on the key rather than memory itself. The reason for only 4 bits comes from the architectural limitation. As illustrated in Figure 1, the PKRU register is a 32-bit thread-local register that can hold permission for 16 keys. A pair of bits hold the permission of a key, write access(WD), and all access (AD) are two modes allowed for a key.

We elaborate on the usefulness of MPK using a simple example. Let us assume a thread wants to perform some cryptographic operation, such as key generation and storage in a memory region $0x7F342202C000$ to $0x7F342202b000$. The thread binds this region with a pkey = 1 (requested through a system call). The thread gets full control over this memory region, and it can allow any other untrusted part of the process to access this memory. The permission change is entirely user mode and can be executed using WRPKRU/RDPKRU instructions, which eventually eliminates the context-switch penalty incurred in the traditional method (*mprotect()*). The reason for this is, the is no need to for flushing the TLB and

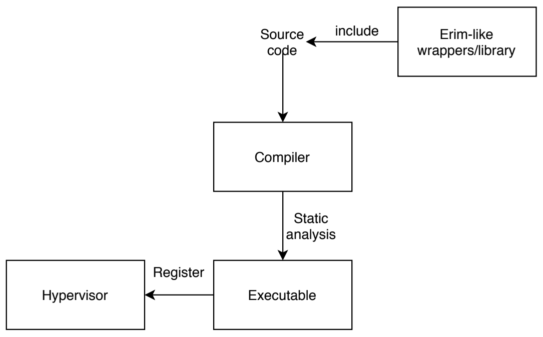


Figure 2: Overview of our system design and interaction between different components.

updating PTE. *What stops other parts of the process changing permission to the key?* PKRU register is thread-local, and it is available for each thread; therefore, permission change from another thread at run-time is not feasible.

The recent kernel comes with APIs that support MPK operations. $pkey\_alloc()$ allows the user program to get an available key from the CPU, and $pkey\_set()$ changes permission to that key. $pkey\_mprotect()$ is the MPK supported version of $mprotect()$ that binds the key to the memory specified memory area. User program releases the key using $pkey\_free()$. Note that $pkey\_free()$ does not update the PTE and so reusing the same key somewhere else in the program, trusted or untrusted, essentially produces inadvertent memory binding to that key.

Apart from architectural support, CoMPK depends on decomposition of code into private compartments we term as private functions and static analysis techniques employed in an LLVM compiler pass. For checking compile time errors, we write a compiler pass that scans the intermediate representation of a program that is generated through the Clang front-end for C-based applications. Our analysis consists of control-flow analysis for restricting access to MPKs outside of private functions and data-flow analysis to make sure no data is being leaked outside our private function at compile time. Furthermore, we deploy a beta-kernel module for monitoring system calls made by our application. The primary goal of this module is to restrict system calls made from outside of private functions. This adds another ring of security to overall architecture of the applications that are written using CoMPK. The kernel module is responsible for run-time security of applications along with our implementation of libmpk [5].

4 Design

This section discussed the progress that we have made so far on all the deliverables we listed in the initial project proposal. For reference, Figure 2 shows the interaction between the different design components.

4.1 ERIM-libmpk Integration Design

As discussed in the research goals, one of the major objectives is to provide intra-process level data isolation, and that could be achieved by using an ERIM like program layered over *libmpk* to provide more than the 16 physical keys available. The process begins by requesting the kernel component of libmpk for all available keys. This is essentially $pkey\_alloc()$. Once the key is returned, it is maintained in a cache like structure that maps *pkeys* and *vpkeys*. The limited physical key is mapped with virtual keys using eviction of a *vpkeys* from the cache. libmpk provides an API, $mpk\_mmap()$ to allocate secured memory. This function internally calls mmap() API, but before returning to the user space, it binds a pkeys. When returns, the $mpk\_mmap()$ returns a vpkey and a memory region protected by that vpkey. The cache is updated with current pkeys and vpkeys at an eviction rate in $mpk\_init(evictionrate)$. Once we have a memory allocated and grouped to a virtual key, we create a protected zone in any location of a process by calling $BRIDGE\_DOMAINRW(vpkeys)$ or $BRIDGE\_DOMAINR(vpkeys)$ and $EXI_DOMAIN(vpkeys)$. The first API changes the PKRU value, such it protection memory can be read or written. Once we exit the domain $EXIT\_DOMAIN(vpkeys)$, further access to the memory produces segment fault. $DESTROY\_DOMAIN(vpkeys)$ releases a virtual key and unmaps all the pages bound to that key; this essentially removes the page address, and all metadata from the hash data structure maintained in the kernel component in the libmpk.

4.2 Compiler Design

Our compiler is structured as a single LLVM pass that performs control-flow and data-flow analysis on specific functions that are part of the module. Our compiler works by first identifying the trusted and untrusted parts in an application. The trusted parts are declared as private functions so the programmer does not have to worry about leaking data that is protected by MPKs to the stack. After locating both, the trusted and untrusted parts of a program, we proceed to analyze these parts differently.

In an untrusted part, the programmer is not allowed to begin operations by setting a specific key. Our primary analysis of the untrusted part consists of making sure that no MPKs are being set in any untrusted function in the source code. This is achieved by traversing the control-flow graph of every untrusted function and asserting that no instruction is a call to $mpk\_begin()$. If we find a call to this function, we issue a warning to the programmer but still continue to look for other errors in the code.

The trusted part, which is implemented as private functions, are subject to basic control-flow and data-flow analysis. The control-flow analysis detects whether the format of the private function is correct. The private function must begin by setting the required key, allowing the private function to access pages associated with that key. If the compiler does not find this call, it issues an error to the programmer. Secondly, the control-flow analysis consists of making sure that the mpk is given up before the function is exited. For this purpose, the compiler locates calls to the "ret" instruction in the function control-flow and begins traversing the instructions in the basic block backwards. If it does not locate the $mpk\_end()$ instruction before reaching the beginning of the basic block the compiler issues an error to the programmer as this is a control-flow violation. The "ret" instruction and the $mpk\_end()$ instruction should be part of the same basic block so as to ensure that mpks are given up successfully before exiting the function.

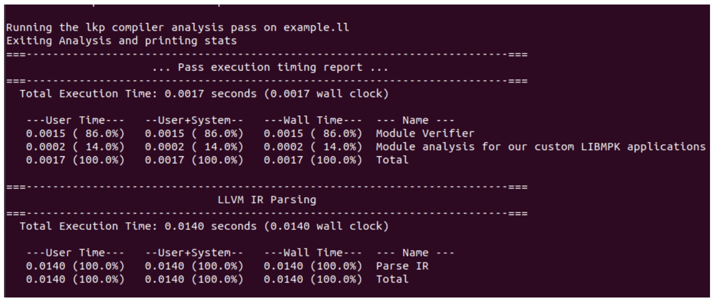
Lastly, every trusted part of our application is subject to data-flow analysis. To make sure that no data is copied from the pages that are guarded by MPKs, our compiler conducts basic data-flow analysis of every private function in the module. The data-flow analysis consists of inspecting memory locations that are part of the function and memory locations that are outside the functions. Memory locations are merely locations in the memory which stores variables. The compiler stores all variables and intermediate variables that are used in the private function. Next, we inspect the "store" and "memcpy" instructions as they directly interact with memory. The compiler identifies the use of both of these instructions in the private function and inspects the operands of these instructions. If the operands are not part of the stored variables of the specific private function, we tag this as a data leak to memory locations that are outside of the private function and issue a warning to the programmer.

4.3 Hypervisor Design

The hypervisor module is a kernel module that utilizes kprobes to capture calls to specific system calls that we want to monitor. The hypervisor obtains the address to the in-memory system call table. This is then used to add probes to any specific system call that we want to monitor. Initially, we added probes for only the read and write system calls to check if our implementation works. We then tested our module by adding calls to the first 200 system calls in the system call table.

We require that each process that wants to use the hypervisor first open the device driver file that links to our module in $/dev$ and then register with the kernel module by making an ioctl call. The ioctl call allows registering and unregistering with our hypervisor module. Once registered, the kernel module stores the PID of the process that registered in a hash table inside the module. The kernel only tracks the system calls made by the PIDs stored in the hash table.

This kernel module can allow us to achieve multiple features. Firstly, it can allow us to monitor the system calls being made by any process that is using our system for account-



```
Running the lkp compiler analysis pass on example.ll
Exiting Analysis and printing stats
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0017 seconds (0.0017 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0015 ( 86.0%)   0.0015 ( 86.0%)   0.0015 ( 86.0%)  Module Verifier
   0.0002 ( 14.0%)   0.0002 ( 14.0%)   0.0002 ( 14.0%)  Module analysis for our custom LIBMPK applications
   0.0017 (100.0%)   0.0017 (100.0%)   0.0017 (100.0%)  Total

===-------------------------------------------------------------------------===
                                LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0140 seconds (0.0140 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0140 (100.0%)   0.0140 (100.0%)   0.0140 (100.0%)  Parse IR
   0.0140 (100.0%)   0.0140 (100.0%)   0.0140 (100.0%)  Total
```

Figure 3: Run-time complexity of our compiler pass.

ing purposes. Secondly, it can enable us to restrict system calls from trusted zones only by checking the *PKRU* register value at time of the system call. If the call is coming from our trusted zone/private function, then the value of the *PKRU* should be non-zero. However, if the call is coming from an untrusted part of an application written using COMPK, we can immediately abort the process that made the call as it can be malicious.

5 Implementation

Libmpk removes the hardware barrier by using *vpkeys* (virtual *pkeys*). A kernel component manages a hash structure that holds the ERIM like call gates are constructed over libmpk with the help of C macros along with the usage of user-mode WRPKRU function for writing to the PKRU register for 1024 virtual keys.

The system call monitoring mechanism is implemented as a kernel module with the help of Kprobes library. Using the kprobes library, we place probes over desired syscalls to ensure that the module doesn't have rogue system calls through injection into the binary. Due to the usage of an compile time and run time analysis for rogue MPT calls, no binary analysis would be required.

The compiler is implemented as an LLVM pass using LLVM 10.0.0 and Clang 10.0.0. Clang is the front-end we use to generate LLVM bitcode which is not architecture dependent. We then use the LLVM bitcode to perform our analysis. We use LLVM's extended API to traverse control-flow graph of an application and perform policy checks.

6 Evaluation

The evaluation criteria for a system that seeks to improve intra-process isolation is threefold:

- Overall system correctness
- Resilience against user-space attacks that tend to leak sensitive data

- The run-time overhead for the execution of an application built using the system

We aim to showcase that our system when used with an application that requires isolating segments of code from each other correctly isolates different segments by not allowing a part of the program that does not have the relevant permissions to access certain data. To do this, we can either use already built applications and modify them to work with our system or come up with new small-scale programs. We aim to prove evaluate our system's correctness and run-time overhead using small-scale programs that require basic data isolation. We will also adapt our system to already built applications that are used by [1], [4] and [6] for evaluation. This will allow us to directly compare against the previous state-of-the-art and compare run-time overheads of our system all prior systems we use to motivate COMPK. Furthermore, it will also help us measure how well our system would integrate with real-world applications.

As an extra metric for evaluation, we plan on comparing the binary size overhead incurred by each system for a given application. Most of the prior work requires modifying the application source code to run and so does COMPK. Comparing how the binary size of the executable of an application changes when deployed using different systems will add an extra dimension to our evaluation.

6.1 Platform

Our evaluation platform is based on a virtual machine that is equipped with 8-core Intel Xeon Platinum 8180 CPU @ 2.5GHz, and Ubuntu 16.04. It runs a kernel (v4.14.2) with *libmpk* system calls. Note that the kernel is configured with a protection flag ON [1] to allow the MPK access from the OS. Since this is not a standard kernel, we need to update kernel header manually. In addition, the perf tools does not run on custom kernel due to dependency issues; we avoid using it by writing a performance measurement tool.

6.2 Microbenchmark setup

The primary overhead comes from the key(*pkey*) virtualization. *libmpk* maintains a cache-like table using a hash data structure in the kernel to keep track of the virtual key to physical key translation. We utilize *RTDSCP* instruction to capture cycles needed for a function. *RTDSCP* by default takes care of the out-of-order execution issues by taking the time stamp only after all the instructions before *RTDSCP* instruction. Accurate measurement requires disabling all interrupts and preemption to take ownership of the CPU core for the measurement duration. Therefore, measuring cycles in the kernel context produces more consistent results. Previously reported results are calculated using a kernel module.

[1] $CONFIG\_X86\_INTEL\_MEMORY\_PROTECTION\_KEYS = y$

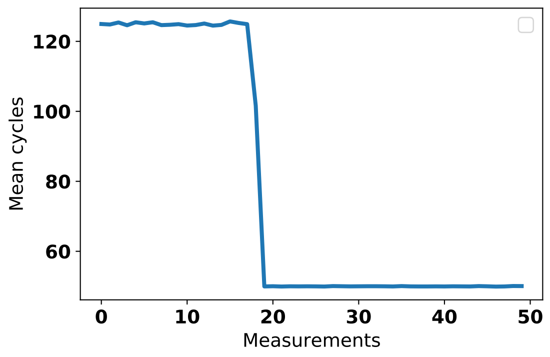


Figure 4: The plot illustrates measurement overhead (error) generated from the measurement tools.

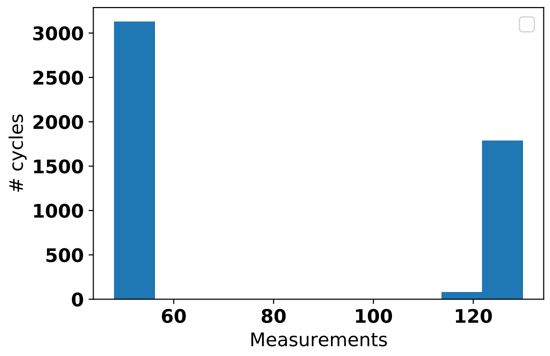


Figure 5: The graph illustrates the measurement error distribution *RDTSCP* instruction.

However, we adapted that full-user space cycle measurements because COMPK contains a library that is not available to the kernel. To ensure that the process is ergodic, we measure cycles in a nested structure following Intel's suggestion. We have 50 iterations in the outer loop and 100 iterations in the inner loop. Therefore, a total of 5000 measurements. All measurements are performed using a single thread process. Before running the test for COMPK, we measure the measurement system's errors by running the test setup blank (no software to test). The module reports only the offset/overhead incurred by the evaluation module itself. In Figure 4, we plot the mean overhead cycles. Although a large error is generated at the beginning, the error settles ≈ 50 cycles. We observe a larger error in the user mode module compared to the kernel version. We plot the distribution of error cycles in Figure 5. To mitigate such noise, a library compatible with kernel context may be a solution — this remains future work.

6.3 Overhead

To Evaluate the cycle counts in the COMPK, we create 15 isolated domains; each spans 4KB virtual address. Although the system works with a large number of keys, we consider

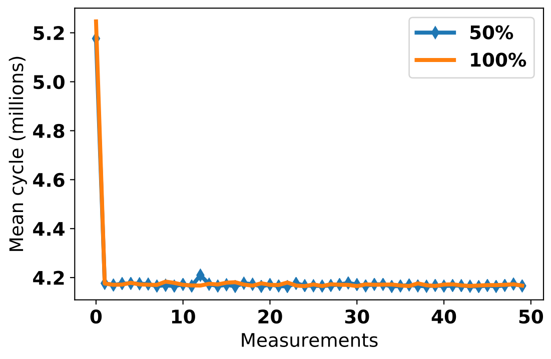


Figure 6: Illustrates the cycle counts at two different eviction rate.

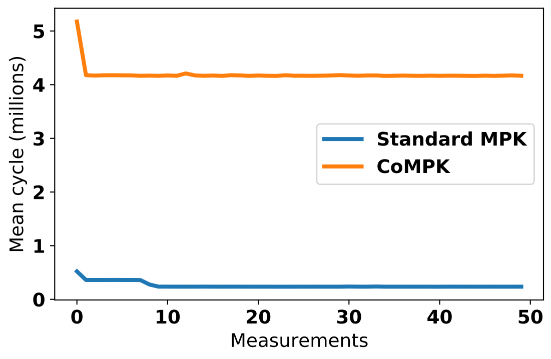


Figure 7: Overhead incurred by the COMPK.

only 15 because standards MPK allows 15 keys at a time. We change the permission of the domains 5000 times to compare the performance. In Figure 6, we plot the mean cycle counts for 50 measurements (each group 100) for two different eviction rates. 100% eviction rate produces a large cycle count initially, but it follows the cycle counts at 50% eviction rate closely afterward. Final result is similar for both the rates because the key allocation is remains out of the count measurements, multiple threads that uses same core should consume more cycles at 100% rate because of frequent cache miss.

We compare the performance with standard API in 7. The cycle count increases in COMPK produces larger cycle counts because of the *libmpk's* key virtualization and page handling mechanism. A potential future work can be to look for this overhead reduction technique.

6.4 Compile-time overhead

Figure 3 shows the compile time overhead for our compiler pass. The LLVM compiler infrastructure runs multiple passes on the generated bitcode to optimize the target independent intermediate representation. Our compiler runs as part of this pipeline. We run clang with optimization level O0 hence LLVM only runs the most basic passes required to generate the intermediate representation. Our pass takes only 14% of the total time and hence does not result in a lot of overhead in the compiler optimization phase.

7 Future Work

We leave room for future work in our current implementation:

Compiler: One addition to the system could be to perform more efficient compile time static analysis. We can incorporate alias analysis and dead-code elimination in our compiler pipeline. Furthermore, we can also use dynamic analysis to detect malicious activity on run-time and launch handlers to deal with the activity.

Kernel module: We do not fully integrate the hypervisor with our system and leave that to future work. We could implement the kernel module more efficiently to reduce overhead. Furthermore, right now the module only registers processes and captures system calls for these processes but does not kill COMPK processes if they access system calls from outside the trusted zone. This could be extended to kill the process by checking the value of the *PKRU* at the time the system call is made.

Binary rewriter: Lastly, ERIM [6] integrates with a binary rewriter to insert no-ops in the binary where a malicious user/process could execute the *WRPKRU* instructions to change into trusted mode. By integrating the binary rewriter with our system we can mitigate this threat. We leave this for future work

8 Challenges

Our project primarily relies on the architectural support of a processor. Memory protection key is somewhat a modern concept introduced in the Intel processors. The first challenge was to find a system that has such a feature. Since we needed lower level access that allows kernel modification, it became a significant concern. Here, we would like to thank Dr. Min for allowing access to such a state-of-the-art machine.

Finding guidance on setting up a system that allows MPK operation is a challenging task because of the lack of materials online. Learning how to set up a system with MPK support and getting used to it was one of the main challenges we faced.

Working with a research code in tandem with the paper is always a challenging task. Lack of consistency in the naming convention between paper and its codes thwarted the progress of the project. This is essentially a lesson for all of us that maintaining proper documentation and following a consistent coding style are important issues for such complex system building.

References

[1] Y. Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. *2016 IEEE S&P)*, 2016.

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

[3] Intel Intel. and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64):64, 64.

[4] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An {OS} abstraction for safety and performance. In *12th {USENIX} ({OSDI} 16)*, 2016.

[5] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys, 2018.

[6] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th ({USENIX} Security*, 2019.