Secure and Efficient In-process Monitor (and Library) Protection with Intel MPK

Xiaoguang Wang  
Virginia Tech  
xiaoguang@vt.edu

Pierre Olivier  
The University of Manchester  
pierre.olivier@manchester.ac.uk

SengMing Yeoh  
Virginia Tech  
sengming@vt.edu

Binoy Ravindran  
Virginia Tech  
binoy@vt.edu

ABSTRACT
The process reference monitor is a common technique to enforce security policies for application execution. Reference monitors can be used to enforce access control, check program integrity, detect attacks and even transform program states. Deciding where the monitor resides involves a trade-off between strong monitor isolation and low switching overheads. Running the monitor in the same address space as the protected/traced application (in-process monitors) allows for low overhead but raises isolation concerns. Thus, existing work place monitors in a separate address space, which leads to expensive monitor invocation cost.

We present MonGuard, a system in which a high-performance in-process monitor is efficiently isolated from the rest of the application. To that aim, we leverage the Intel Memory Protection Key (MPK) technology to enforce execute-only memory, combined with code randomization to protect and hide the monitor. MonGuard inserts instrumentation around sensitive instructions to further prevent possible code reuse attacks. We built a prototype of MonGuard as a loader extension and implemented a multi-variant execution (MVX) monitor. The evaluation shows MonGuard enhances the monitor protection with nearly zero performance overhead.

CCS CONCEPTS
• Security and privacy → Systems security; Software and application security.

KEYWORDS
Memory Protection, In-process Monitor, Software Security, Multi-Variant Execution

ACM Reference Format:

1 INTRODUCTION
The process reference monitor is commonly used to enforce security policies for application execution [1, 4, 6, 13, 15, 22, 26, 28, 29]. In the reference monitor model, the potentially malicious code is confined in a sandbox with a limited number of entrances to validate the reference requests. The reference requests are made in the form of remote procedure calls to the monitor. Based on a pre-defined policy, the monitor may enforce an access control model [6, 28], check the program integrity [1, 15] or behavior [2], transform the program state [29], etc.

One well-known problem of such reference monitor systems is the conflicting goals of strong monitor isolation and low communication cost. Some existing works use process-level isolation [21, 22, 26] or hypervisor-based isolation [4, 13, 27] to enforce a strict isolation between the monitor and the untrusted application code. However, the fact that the monitor resides in a separate address space from the monitored code leads to unavoidable performance overheads when the untrusted code frequently traps to the monitor. For example, a number of Multi-Variant eXecution systems (MVX, a code-hijacking detection technique) use the ptrace interface to isolate security monitors [6, 21, 26]. The main drawback of an out-of-process monitor design is the performance overhead that can be up to 9x when running commodity server applications such as Nginx [26]. On the other hand, in-process monitors often have cheaper communication costs since no context switch is needed. However, protecting the monitor code itself becomes a new problem. For example, Shuffler [29] uses an in-process monitor to continuously randomize the application code location to defeat code reuse attacks. Meanwhile, Shuffler also randomizes itself to prevent the monitor from being exploited. The extra effort to protect monitor code itself makes it hard to be applied to general cases.

Some other research uses protection domains within the target address space to isolate the monitor code. For example, segmentation in x86-32 CPU can be used to define logically isolated memory regions for sensitive data isolation [15, 30]. Unfortunately, the x86-64 architecture has mostly dropped support for segment limit in 64-bit mode. The monitor can also be placed in the Operating System (OS) kernel. Kernel-based monitors are logically separated from the target application code. However, with recent side channel protection (Kernel Page Table Isolation), the kernel and application address spaces are separated which adds to an already high user/kernel world switch latency [19]. Furthermore, they may have limited access to the target memory, since recent processor features such as SMAP and SMEP prevent kernel code from executing/accessing user space memory to prevent ret2usr like attacks [12]. Finally, developing application reference monitors inside the kernel is a
nontrivial work because it is very hard to avoid introducing unintended bugs to the system TCB [26]. Other technologies such as Trusted Execution Environments [10] have also been shown to have a high switching cost [14].

In this paper, we present MonGuard, a system to protect in-process reference monitors and shared libraries. MonGuard exploits the Intel Memory Protection Keys (MPK) hardware extension [11]. MPK provides intra-address space memory permission checks in addition to the existing hardware memory protection architecture. Specifically, it allows the application memory to be split into several domains with different access permissions (write disabled or access disabled). The application can update the access permission of memory domains instantly by writing the permission bit array to a register. As this does not involve page table update, the MPK permission switch can be very fast. For example, it only takes less than a few hundred CPU cycles for each memory permission switch [24]. MPK only checks data accesses but not instruction fetches. As such, most existing works leverage MPK to protect sensitive data from being accessed (Heartbleed [8] alike attacks) inside the process space [9, 20, 24].

MonGuard, on the other hand, leverages MPK to protect an in-process monitor from being exploited. Specifically, MonGuard combines execute-only memory and code randomization to hide the in-process monitor from application code. Furthermore, the sensitive instructions inside the monitor are instrumented so that even if powerful attackers find the hidden monitor the unintended monitor code execution will be detected. To demonstrate the usability of MonGuard, we build a multi-variant execution prototype with MonGuard. The evaluation shows that our prototype performs 4x faster than the out-of-process monitor approach when running Nginx workload and 10% faster than a state of the art in-process monitor code execution will be detected. To set/change permissions of a memory domain, an unprivileged instruction wrpkru can be used to update the PRKU register. The memory permissions can be update instantly. This is different from the memory domain mechanism in ARM and PowerPC, where the kernel maintains the memory domain privilege [31]. Note that the memory key protection only works for memory data accesses. Interestingly, if the code pages are associated with an access disabled protection key, the code will be no longer read but still can be executed. This implements the execute-only memory (XoM) [3]. MonGuard leverages XoM to prevent trampoline code from leaking out the monitor location.

Intra-Process Isolation: An application can be split into different protection domains. This is especially useful for sensitive data protection (e.g., SSL key). Researchers have proposed using OS primitives [17], x86 segmentation [16], and even virtualization techniques [18] to protect sensitive data inside the address space. For example, light-weight contexts (lwCs) modifies the OS kernel to provide independent units of isolation within a process [17].

Following this direction, recent work leverage Intel MPK to achieve sensitive data isolation with cheaper performance cost [9, 20, 24]. For example, libmpk is proposed as a library to virtualize the protection keys for the scalability problem [20]. ERIM further utilizes binary analysis/rewriting to prevent unintended sensitive MPK instructions from being maliciously used [24]. Although the MPK memory permission switch can be very efficient, MPK itself does not guarantees the code page safety.

2.2 Threat Model

We assume the attacker has access to the target binaries, such as application, its shared libraries, as well as the monitor. At runtime, the attacker can only access the target process remotely through the standard I/O interface, namely a socket connection. The attacker can send arbitrary data to the target process. We assume a trustworthy TCB including the OS kernel and the compiler toolchain.

3 DESIGN AND IMPLEMENTATION

3.1 Overview

Figure 1 shows a high-level overview of the application address space layout when running under MonGuard. Application code and data memory are separated from the monitor and shared libraries through a call gate which directly transfers the control from application code to the reference monitor. The call gate contains direct jump instructions to the monitor code. MonGuard further marks the call gate code pages with access disabled. As a result, attackers are unable to read the call gate code memory to locate the monitor in the address space, thereby preventing just-in-time payload generation attacks [23].

By doing so, MonGuard ensures the call gate is the only legal entrance to invoke the reference monitor. Since the monitor was loaded at a random offset from the application code, it is difficult to locate the monitor address without having any direct pointer information. The only pointer information (monitor/library addresses) in the application context is embedded in the trampoline code as immediates (within the jump instructions). While an attacker may try to brute-force the address space in order to modify the monitor credential data (e.g. change the result of a monitor request), those
Secure and Efficient In-process Monitor (and Library) Protection with Intel MPK

Figure 1: System overview of MonGuard with application context and the monitor (shared libraries) context. The call gate memory is executable but not readable.

3.2 Monitor Memory Isolation

MonGuard leverages MPK to enable the "one-way visibility" of a reference monitor. MonGuard logically splits the application address space into an application context and a monitor/library context. Only the monitor/library context is allowed to access the application context, but not vice versa. To achieve this, MonGuard prepares two memory protection keys for the monitor (including the shared libraries) code and data respectively.

The first protection key (PKEY 1) is assigned to all code pages, which includes the code of the monitor, shared libraries and the trampoline call gate. PKEY 1 disallows the read permission of those code pages, making the monitor code execute-only. This can prevent the issue of direct code address leakage, which might be used for dynamical vulnerability discovery and payload generation [23]. Application code can be also optionally assigned with PKEY 1, further preventing attackers from directly reading from the application’s code pages. The second protection key (PKEY 2) is associated with the monitor and shared library data pages. The attribute of PKEY 2 will be updated based on the execution context. When the code execution is in the application context, PKEY 2 is set with access disabled. However, when the application calls the monitor or library code through the trampoline, the monitor enables the data access by clearing the access disabled bit of PKEY 2 in the PRKU register. Before leaving the monitor context, the monitor sets the access disabled bit again, thereby ensuring the monitor’s data integrity.

Currently, MonGuard isolates both the reference monitor and the shared library data from the application code context, which makes the attack surface as small as possible. To achieve this, MonGuard intercepts all the library calls and updates PKEY 2 accordingly to allow legal library calls. MonGuard does not isolate the stack and heap memory during the application-monitor context switch. In MonGuard’s design, we assume the monitor does not call the application code. Therefore, the application code will never have a chance to manipulate the monitor’s stack. Heap is another potential target for attacker. In current monitor implementation, we did not use any dynamic memory. However, we do provide a hooked MonGuard mmap implementation to associate 4k pages with PKEYs. A more comprehensive solution might be to embed a simple malloc() free() implementation on top of the protected mmap’ed memory.

3.3 Monitor Instrumentation

Besides monitor data, monitor code is also a prime target for attackers. Although the monitor and the libraries are hidden from the application code by Address Space Layout Randomization (ASLR), there is a probability a powerful attacker may perform a brute-force search of the 64-bit address space. For example, an attacker could use an infinite loop to fork child processes to jump into random addresses within the address space. If the attacker is able to continuously cause jumps to memory locations and observe the effects of these jumps (e.g. an infinite loop instead of a trap), they might be able to figure out the location of the monitor code pages. This technique was similarly used in the Blind-ROP attack [5]. With MPK, the PKEYs can only remove the read permission of the code page but not the execution permissions. The possibility of unintended monitor code execution is considered harmful since the attacker may have the ability to construct ROP payload and bypass the monitor checks.

To solve that, MonGuard adds instrumentation to sensitive monitor instructions. First, MonGuard inserts an instruction that touches monitor memory before every indirect control transfer instruction, for example, the indirect jmp/call and the ret instructions. These indirect control instructions can be used to launch an attack and redirect control back to the application code. When an attacker attempts to chain gadgets with a jump to the instructions preceding a return instruction the memory touch instruction will cause an MPK fault as this access did not go through the call gate to switch the PKRU permissions. The wrprku instruction is another instruction an attacker should not be able to have control over. MonGuard adopts a similar instrumentation method as used by ERIM [24]. Specifically, it inserts an additional check to the %eax value right after the wrprku instruction which removes protection to prevent PKRU register from being maliciously used to remove protection from pages.

3.4 Implementation

In MonGuard, the monitor needs to be compiled as a position-independent shared library (i.e., libmonguard.so) so that it can be loaded at a random location inside the 64-bit process address space. We use LD_PRELOAD to load the monitor into the application address space. The Linux LD_PRELOAD environment variable instructs the dynamic linker to preload the shared library before processing the application’s dependency list. MonGuard also leverages LD_PRELOAD to override symbols in other dynamically linked libraries such as libc, enabling it to intercept calls to shared libraries from application context.

3.4.1 MonGuard Call Gates. The MonGuard call gate is a jmp instruction followed by the memory permission (PKRU) update. The jmp instruction is originally from the PLT code, which transfers control to the monitor. MonGuard leverages the structure of the shared library to hide the location of the monitor. Specifically, the compiler generates the PLT/GOT for each external library function (e.g., printf in libc). Each PLT entry contains an indirect jump with the destination address stored in GOT (the .got.plt section in ELF
The loader resolves the external function address and fills in the corresponding GOT entry in what is known as on-demand symbol resolution (lazy-binding). However, there is a subtle security issue: the GOT contains the monitor and libraries addresses which can then be leaked.

### Before PLT patching:

```
<monitor_call@plt> jmpq *GOT[n]
<monitor_call@plt> pushq n
<monitor_call@plt> jmpq RESOLVER_ADDR
```

### After PLT patching:

```
<monitor_call@plt> movabs MONITOR_ADDR, %rax
<monitor_call@plt> jmpq *%rax
<monitor_call@plt> nop
```

![Figure 2: Original PLT slot patched to new PLT slot](image)

The premise of MonGuard relies on the reference monitor being well hidden against potential attackers. Since we have preloaded the libc calls with our call gates, the .got.plt slots now contain pointers to the hidden reference monitor. To prevent the program from leaking information about the location of the reference monitor in the address space, we take advantage of musl-libc's lack of lazy binding support. By the time the constructor of MonGuard is called, the jmp instruction in the PLT has already been resolved to reference its respective slot in the .got.plt section containing the address of the call targets in the shared library. We patch the PLT slots with immediate addresses corresponding to their respective jump targets, and clear the .got.plt slots (As shown in Figure 2). Subsequently, we apply the execute-only memory protection to the PLT code pages. This patching is performed on program startup in the MonGuard constructor.

Once the application invokes a monitor call, the patched PLT redirects the control to the monitor code. The monitor clears the $rax value to prevent any potential monitor address leakage. Next, the monitor deactivates the PKEY protection for monitor data and will reactivate it when leaving the monitor. An extra benefit of using the trampoline jmp instruction to update MPK protection is that we can hide the sensitive wrpkru instructions from the application code. For those unintended wrpkru instructions occurred in the application code, we could adopt similar techniques used in ERIM [24] or Hodor [9]. For example, using binary rewrite to replace the unintended wrpkru instructions with semantically same instructions [24], or using the debug register to monitor the unintended wrpkru use [9]. In order to prevent internal libc functions from calling other libc functions through the PLT and going through the call gate again, we passed the -Bs symbolic flag to the linker when building musl-libc, thereby preventing symbol inter-positioning.

Listing 1 shows the pseudo code of a monitor call routine. The monitor defines a global dummy variable (line 1 in Listing 1). We instrument a dummy variable write before each indirect control transfer instruction, preventing a powerful attacker from performing a ROP attack into the monitor code and hijacking the control flow back to application code. The monitor DEACTIVATE is a macro of inline assembly removing the PKEY data access protection. After completing the real monitor code in line 9, the monitor re-enables the protection by using ACTIVATE macro. The ACTIVATE macro’s implementation is very similar to the call gate used by ERIM [24]. Specifically, it enables the MPK protection by updating the PKRU and checks the eax value after the wrpkru instruction to prevent the control flow hijack breaking the integrity of the gate code.

```c
int canary = 0; // Barrier variable
int monitor_call@Ref_Monitor()
{
    // Clear $rax used in .plt trampoline
    asm("xor %rax,%rax");
    DEACTIVATE(); // Disable data protection

    // Reference monitor implementation */
    real_monitor_code();

    // Touch monitor data
    asm("mov $0x0 , %0",=r(canary));
    ACTIVATE(); // Re-enable protection
    return retval;
}
```

### Listing 1: Pseudo code of a monitor call routine

#### 4 CASE STUDY AND EVALUATION

In this section, we first study a use case of the in-process monitor protected with MonGuard, next we report the evaluation results.

**Case Study: An MVX Monitor.** We implemented a Multi-Variant eXecution (MVX) monitor using MonGuard. MVX is a technique aiming to detect software attacks based on control flow hijacking [6, 13, 21, 26]. It does so by running in parallel multiple instances of the same program, named variants, that are functionally equivalent but which implementations differ. A MVX monitor feeds the variants with same inputs and monitor their behavior. The difference of implementation between the variant makes that an attack leads to a divergence of their behavior. When it happens, the MVX system raises a security flag. For example, when the variants’ address spaces are fully non-overlapping [25], a ROP chain jumping to an absolute address will lead to the crash of one variant.

Figure 3 shows the layout of our in-process MVX monitor under MonGuard protection. The monitor and libc context is protected with the techniques mentioned in Section 3. There are two variants currently running as two processes. The master variant (variant 1) handles the I/O directly, e.g., the socket connection. The follower variant (variant 2) duplicates the execution for the intrusion detection. For most cases, the duplicated execution will not cause
Figure 3: In-process MVX monitor protection with MonGuard.

Figure 4: Performance overhead of MonGuard and a ptrace monitor (Normalized to application running w/o monitors).

any problem because of the process space isolation. However, the duplication may double the system wide state, for example, writing a memory buffer to a local file twice. In this case, the MVX monitor simulates the file write by updating the side effects (e.g., the return value of the write call). In our MVX monitor, we intercept and simulate the state at the libc call level.

**Evaluation:** First, we evaluate the performance overhead of a MonGuard-based monitor and a ptrace-based monitor under CPU intensive (SPEC INT2006) and I/O intensive (Nginx) workloads. SPEC INT2006 consists of several CPU-intensive benchmarks, stressing the system’s processor and memory subsystems. Nginx is a widely used web server in production environment. In our evaluation, both monitors only intercept the external procedure calls (system calls, libc calls) without any further inspections. Figure 4 shows the evaluation results. For most CPU intensive workload, the ptrace monitor and the MonGuard monitor perform similarly. However, ptrace-based monitor brings 4.5x performance overhead for I/O intensive work (e.g., Nginx). This is likely because Nginx issues more frequent external procedure calls than CPU intensive workloads, which amplifies the overhead for the costly out-of-process monitor. To prove our assumption, we profiled the external procedure call rate (number per second). Nginx performs 31,847 system calls per second while 403.gcc performs 9,281 system calls per second. Other applications only issue less than 500 system calls per second on our test machine.

Next, we evaluate the performance of the MVX monitor mentioned at the beginning of this section. Our MVX prototype is compiled as a shared library protected by MonGuard. Currently, the MVX monitor simulates 35 libc calls for the follower variant execution. For example, the file descriptor related functions such as fopen, fdopen, close; the networking and event poll related functions such as epoll_create, sendfile, writev, etc. We measured the performance of running two Nginx variants under our MVX monitor. We also used ApacheBench as the workload generator and tested on the loopback (0.1ms network latency) as mentioned in a state-of-the-arts MVX system (ReMon [26]). With MonGuard, MVX monitor brings 2.7x overhead for Nginx workload, which outperforms the 3x overhead of ReMon [26].

5 DISCUSSION

**Generalization:** Intra-Process Monitors with Intel MPK: The capacity of Intel MPK to provide memory protection on a per-thread basis within a single address space, combined with the low latency of the protection domain switch operation, make for some interesting applications beyond its use in MVX systems. The principles presented in this paper could be generalized to other tracing/monitoring security applications in which a trusted monitor needs to be isolated from the untrusted monitored program. Using MPK, the monitor can be placed within the untrusted program address space and protected from the rest of the untrusted code. Locating the monitor within the monitored process should allow for low latency switching between monitored and monitoring code. As a result, the performance should be enhanced compared to traditional solutions placing the monitor in another process or in the kernel/hypervisor [13, 22, 26, 27]. There are several examples of systems beyond MVX that may benefit from protecting a trusted monitor with MPK, such as sandbox and fault isolation [30, 31], malware tracers/monitors [22], fine-grained system call tracing/filtering [26], etc. We believe it would be an interesting topic to convert those existing tools to use process monitors protected by MPK-based approach.

**Limitations of Intel MPK:** Although it is an interesting technology, MPK does not come without limitations. First of all, there is a limited set of protection keys (16) that may not suit all monitoring scenarios. This limitation can be removed by virtualizing the keys [20], however this comes at the cost of lower performance. A second limitation stems from the fact that the PKRU switch operation is, for performance reasons, an unprivileged instruction. Combined with the fact that MPK does not check memory accesses on instruction fetches, it raises concerns about PKRU manipulation by untrusted code, either directly or indirectly through techniques such as Return-Oriented Programming (ROP). Thus, isolation schemes must be complemented with static code analysis [24] to validate each update of the PKRU register. Sequences of bytes forming PKRU manipulating instructions may also appear...
due to the variable size of the x86-64 instruction set, in a similar manner as ROP gadgets. Solution have been proposed including binary rewriting techniques [24] as well as traps based on hardware watchpoints [9] to address that issue. Finally, simply updating the PKRU upon security domain switch does not prevent the leakage of registers content between domains. This can be exploited to mount an attack. Although the solution is to save, scrub and restore registers values, this may impact the latency of security domain switch operations.

6 CONCLUSION
We have presented the design and implementation of MonGuard, a system to protect in-process monitor and the libraries. MonGuard leverages Intel MPK to efficiently update memory access permissions. MonGuard implements execute-only memory and code randomization to hide the monitor code. We have built a prototype of MonGuard and use it to implement a protected in-process MVX monitor. The evaluation shows MonGuard can greatly improve the monitor performance with reasonable intra-component isolation.

ACKNOWLEDGMENTS
We would like to thank the anonymous reviewers for their insightful comments. This work is supported in part by US Office of Naval Research under grants N00014-16-1-2711 and N00014-18-1-2022.

REFERENCES