Efficient Sealable Protection Keys for RISC-V

Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi

Department of Electrical and Computer Engineering, Boston University
{delshad, scanakci, megele, joshi}@bu.edu

***Abstract*—With the continuous increase in the number of software-based attacks, there has been a growing effort towards isolating sensitive data and trusted software components from untrusted third-party components. A hardware-assisted intra-process isolation mechanism enables software developers to partition a process into isolated components and in turn secure sensitive data from untrusted components. However, most of the existing hardware-assisted intra-process isolation mechanisms in modern processors, such as ARM and IBM Power, rely on costly kernel operations for switching between trusted and untrusted domains. Recently, Intel introduced a new hardware feature for intra-process memory isolation, called Memory Protection Keys (MPK), which enables a user-space process to switch the domains in an efficient way. While the efficiency of Intel MPK enables developers to leverage it for common use cases such as Code-Pointer Integrity, the limited number of unique domains (16) prohibits its use in cases such as OpenSSL where a large number of domains are required. Moreover, Intel MPK suffers from the protection key use-after-free vulnerability. To address these shortcomings, in this paper, we propose an efficient intra-process isolation technique for the RISC-V open ISA, called SealPK, which supports up to 1024 unique domains. SealPK prevents the protection key use-after-free problem by leveraging a lazy de-allocation approach. To further strengthen SealPK, we devise three novel sealing features to protect the allocated domains, their associated pages, and their permissions from modifications or tampering by an attacker. To demonstrate the feasibility of our design, we implement SealPK on a RISC-V Rocket processor, provide the OS support for it, and prototype our design on an FPGA. We demonstrate the efficiency of SealPK by leveraging it to implement an isolated shadow stack on our FPGA prototype.**

***Index Terms*—Intra-Process Memory Isolation, Memory Protection Keys, RISC-V, Isolated Shadow Stack**

I. Introduction

With the ever-increasing complexity of software applications, today's software code consists of both trusted components designed in-house and untrusted components such as third-party libraries and application plugins. The coexistence of trusted components with potentially malicious or vulnerable untrusted components in the same address space could compromise the security of the system through information leakage, denial-of-service attack, etc [19]. While the user-space inter-process isolation protects processes from one another, the intra-process isolation of various software components has been a challenge. Although it is feasible to invoke the `mprotect` system call from the user space to update the permission bits of specific pages, the performance overhead of `mprotect` due to the context switches between the kernel and user-space can be prohibitive (1,094 cycles on avg. on a modern processor [13]).

To facilitate the intra-process memory protection, in recent years, some processors such as ARM [14] and IBM Power [15] have provided new features to create memory domains by assigning the same `key` to a group of memory pages. However, these features still rely on costly kernel operations for changing domains. More recently, Intel proposed a similar hardware feature, called Intel Memory Protection Keys (MPK) [16], to efficiently support intra-process memory isolation using a user-space instruction (`WRPKRU`) to update the associated permission of a domain. Intel MPK allows the user to create a protection domain by assigning a protection key (pkey) to a group of memory pages. The non-privileged `WRPKRU` instruction, which updates the pkey permissions, takes about 11-260 cycles [18], does not require a context switch, and does not lead to a TLB flush. However, Intel MPK suffers from two major drawbacks, i.e., security and scalability. In terms of security, Intel MPK suffers from pkey use-after-free vulnerability [13]. Once a pkey gets freed, the kernel does not update the pkey bits of its associated pages. The same freed pkey can later on be allocated to a new domain; as a result, the old pages and the new ones will unintentionally share the same pkey. Additionally, if an attacker tampers with a protection domain, its associated pages, or its corresponding permission, the protection keys serve no purpose. In particular, since Intel MPK allows a user-space code to modify the pkey permissions, a malicious component might contain `WRPKRU` instructions or inject those instructions at run-time to update the permission bits of a domain and attain access to a protected domain. In terms of scalability, Intel MPK provides only 16 pkeys. However, some real-world use cases such as Persistent Memory Object (PMO) [24] and OpenSSL [13] require more than 1000 pkeys. The above-mentioned drawbacks hinder the deployment of Intel MPK for enforcing granular intra-process memory isolation.

To enable pervasive deployment of Intel MPK, recent works have focused on addressing one or more of the above-mentioned drawbacks. To prevent the manipulation of a domain's permissions by an attacker, recent works have leveraged binary inspection and binary rewriting approaches [10], [18]. Although Control-Flow Integrity (CFI) [2] can also be utilized to prohibit an uncontrolled execution of `WRPKRU` instruction [13], CFI enforcement mechanisms incur considerable performance overhead. To address the limited number of pkeys, libmpk [13] and Xu et al. [24] have proposed a software-based and a hardware-based virtualization technique, respectively. The virtualization technique of libmpk suffers from large overheads due to expensive Page Table Entry (PTE) updates [24]. While the hardware-based virtualization technique by Xu et

al. [24] provides an efficient implementation, it is not generic and is tailored for a specific application (PMO protection). Note that the pkey virtualization techniques can eliminate the pkey use-after-free problem.

In this paper, we propose an efficient intra-process memory isolation capability, called SealPK, leveraging the Open RISC-V Instruction Set Architecture (ISA) [22]. Similar to Intel MPK, SealPK provides a per-page protection key; however, SealPK supports up to 1024 domains (64× more than Intel MPK) by leveraging the 10 unused bits available in the PTE of each virtual page (Sv-39). We eliminate the pkey use-after-free problem at Operating System (OS) level by keeping track of the number of pages belonging to the same domain and a lazy de-allocation approach.

While Intel MPK does not provide a solution to maintain the integrity of protection domains and their permissions, we propose three novel sealing features to prevent an attacker from modifying sealed domains, their corresponding sealed pages, and their permissions. In particular, our hardware-assisted permission sealing feature enables the software developer to restrict the access to `WRPKRU` within a contiguous range of memory addresses, e.g., a trusted component. Any attempt to execute a `WRPKRU` instruction outside of the specified range would lead to a hardware exception. Hence, this sealing feature efficiently prevents the manipulation of a domain's permission by an attacker. To summarize, our contributions are as follows:

- We present an efficient intra-process isolation capability, called SealPK, which supports up to 1024 unique isolated domains. We propose an OS-level solution to avoid the pkey use-after-free issue. We devise three novel sealing features to protect the domains, their associated pages, and their permissions from unauthorized modifications.
- We implement SealPK on a RISC-V Rocket processor [3] and extend the Linux kernel to support the protection keys for the RISC-V ISA. We evaluate a prototype of our hardware design on an FPGA with a full Linux software stack.
- We demonstrate the efficiency of our design by implementing an isolated shadow stack leveraging SealPK. Our isolated shadow stack prototype is, on average, ∼88× faster than an isolated implementation using mprotect across SPECint2000, SPECint2006, and MiBench benchmarks.

II. Background

A. Memory Protection Keys

In recent years, a growing number of modern processors have provided a per-page protection key capability, where a group of virtual memory pages form a domain and all pages in the domain are assigned the same protection key. Intel MPK utilizes 4 previously unused bits of the PTE to specify the pkey of each page and to divide the address space into up to 16 different protection domains. Intel MPK stores the permission bits of all the pkeys in a single 32-bit register (per logical core), called protection key rights register (`PKRU`). The access permission of each pkey is specified using a 2-bit value in the `PKRU`. Accordingly, each pkey specifies a domain as `readable/writable`, `read-only` or `non-accessible`.

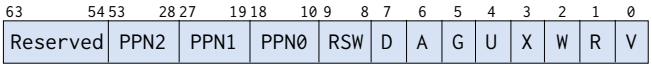

Fig. 1. Sv39 PTE according to the RISC-V ISA [21].

Intel MPK provides two new unprivileged instructions, i.e., `WRPKRU` and `RDPKRU`, to write/read into/from `PKRU`. A user can leverage the `WRPKRU` instruction to update the permission bits of all domains without the need for a context switch. Hence, updating the permission bits of a domain is fast (11-260 cycles [18]); however, `PKRU` is not protected from manipulation by control-flow hijacking attacks [10], [17], [18].

The Linux kernel provides support for Intel MPK (since v4.6) through three new system calls, i.e., `pkey_alloc`, `pkey_free`, and `pkey_mprotect`. The kernel maintains a 16-bit allocation bitmap to keep track of the allocated keys. A user-space thread has to allocate a new pkey using the `pkey_alloc` system call prior to assigning the pkey to a page (group) by invoking the `pkey_mprotect` system call. Using the `pkey_free` system call, the user frees an allocated pkey; however, the kernel only updates the allocation map to indicate that the corresponding pkey is free without erasing the pkey from the PTE of all the corresponding memory pages. The same pkey might be assigned to another domain on future `pkey_alloc` invocations; hence, unintentionally the previous domain would share the same pkey as the new domain, giving rise to the pkey use-after-free problem.

B. RISC-V

The RISC-V Instruction Set Architecture (ISA) [22] is an open ISA. As part of the privileged ISA, RISC-V specifies a page-based 39-bit virtual memory, i.e., Sv39, for 64-bit systems [21]. Figure 1 shows the PTE bits of Sv39, where bits 1-3 (R, W, and X bits) are the permission bits of the page, indicating whether the page is `readable`, `writable`, and `executable`, respectively. As shown in Figure 1, bits 54-63 are currently unused and reserved for future use; hence, as we will discuss in Section III, we leverage these 10 bits to store our per-page pkey.

III. SealPK: Design

In this section, we explain the baseline hardware design of SealPK and its OS support. We discuss the design and implementation of the three novel features of SealPK in Section IV.

A. Hardware Design

As mentioned before, scalability is one of the limitations of Intel MPK, as it cannot support more than 16 pkeys. In RISC-V, we leverage the 10 unused bits (which enables up to 1024 pkeys) of the Sv39 PTE to store the pkey.[1] Figure 2 demonstrates our hardware modifications to support SealPK.

[1]Note that the Sv48 PTE also has 10 unused bits while a 32-bit RISC-V processor uses Sv32, where there are no unused bits in PTE. In this case, we can store the pkey information in a separate OS-managed data structure and use a TLB to cache the information at hardware level.

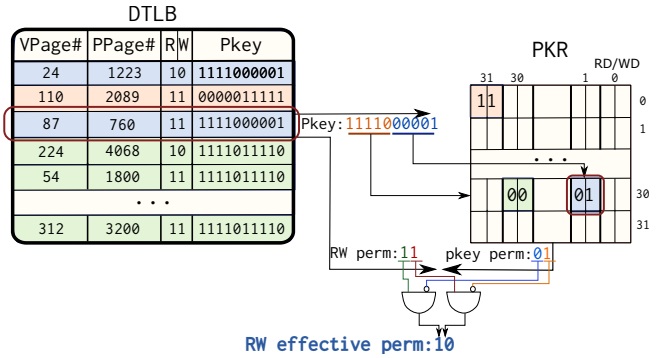

Fig. 2. Modified MMU of the RISC-V Rocket core. Here, we color-code the TLB entries of each domain, consisting of various pages sharing the same pkey. For each data memory access, the effective permission bits are determined by the intersection of the PTE permissions and pkey permissions stored in PKR.

We add a new entry to each line of the DTLB to store the corresponding 10-bit pkey of each virtual page.[2] Hence, our SealPK design supports up to 1024 domains, which is 64× more than the 16 domains supported by Intel MPK. We can use a virtualization-based mechanism, like libmpk [13], to support more than 1024 domains. Note that with a virtualization technique, we can create more than 1024 domains, but in reality we are still limited to 1024 concurrent physical pkeys. For an unlimited number of domains, we can store the pkey information in a separate OS-managed hierarchical structure. We store the permission bits of the pkeys separately. In our design, we use 2 bits, i.e., (Read Disable (RD), Write Disable (WD)), to specify the access permission of each protection key. Following the principle of the least privilege, unlike Intel MPK and previous works, our design enables a write-only page, which can in turn reduce the attack surface. Such a write-only page is specifically useful for log entries, where one thread is responsible for writing the log and another thread processes the written log. Note that the RISC-V ISA does not support write-only pages,[3] and our design provides this feature by leveraging pkeys regardless of the support in PTE permissions.

We support 1024 pkeys in our design; hence, unlike Intel MPK, we cannot simply use a single register to store all the pkey permission bits. To provide fast access to these bits, we use a 2Kb on-chip SRAM-based memory to store the permission bits. This memory, called PKR (shown in Figure 2), consists of 32 rows, where each row stores the permission bits of 32 pkeys (64 bits total). We utilize the custom instruction extension of the RISC-V ISA [22] to define two new instructions, RDPKR and WRPKR, to read from and write to PKR.[4] The RDPKR instruction uses two registers, i.e., rs1 and rd, for its operation. The input register (rs1) contains the pkey. At hardware level, the upper 5-bits of the pkey are used to index into PKR and read the corresponding 64-bit row of permissions. This 64-bit value is returned as the output and stored in rd. The WRPKR instruction uses two input registers, i.e., rs1 and rs2, for its operation. The first input register (rs1) contains the pkey, which is used to index into PKR. The second input register (rs2) contains the new value of 64-bit permissions of the corresponding row. The hardware uses this new 64-bit value to overwrite the permission bits of the row indexed by pkey.

In our hardware design, we provide a control logic to determine the effective permission bits of each data memory access. Consider the example shown in Figure 2, where there is an incoming write request to the virtual page #87. In addition to reading the page's read/write permission bits stored in DTLB (11), the control logic reads the corresponding 2-bit permission bits of the pkey (111100001) stored in PKR. The control logic uses the upper 5 bits of the pkey to index into a specific 64-bit row of PKR and the lower 5 bits to select the 2 permission bits (01). The effective permission is the intersection of the DTLB's and pkey's permission bits. In this example, the effective permission is 10; hence, the write access is not allowed. If a data access is not allowed according to the effective permission, it leads to a load/store page fault; the processor triggers an exception, and the OS handles the page fault.

B. Kernel Support

At the OS level, we add the support to store each page's pkey in the 10 unused bits of the PTE. Our RISC-V kernel support is built upon the existing Linux kernel support for MPK.

1) Lazy de-allocation: To keep track of the allocated pkeys, we implement a 1024-bit allocation bitmap. To efficiently address the pkey use-after-free problem of Intel MPK, we leverage a lazy de-allocation approach. We implement a 1024-bit dirty map to indicate whether each pkey has been lazily de-allocated. We also keep track of the number of pages currently associated with each pkey using a counter map. If a pkey's corresponding counter is not zero, pkey_free updates the permission bits of the pkey in PKR to (0,0); hence, the page-table permissions determine the effective permission of the corresponding pages. Rather than clearing the corresponding bit of the pkey in the allocation map, pkey_free sets the dirty bit and pkey_alloc would not allocate a dirty pkey. Whenever a memory page with a dirty pkey gets freed, we update the number of pages associated with the dirty pkey in the counter map, accordingly. Once the counter becomes zero, we erase the dirty bit of the corresponding pkey; hence, it can safely be allocated afterwards. If pkey_alloc cannot find a free non-dirty pkey, it returns an allocation error to indicate no free pkey is available.

2) Per thread OS support: We modify the task_struct in the Linux kernel to maintain the contents of PKR for each thread during the context switches.[5] Furthermore, we modify the RISC-V page fault handler in the Linux kernel to identify a page fault caused by a pkey permission violation. We augment

[2]Note that pkey checks are only applicable to data memory accesses and not an instruction fetch. Hence, we do not modify the ITLB.

[3]The PTE permissions for a write-only page is a feature reserved for the future use.

[4]To simplify the implementation of the custom instructions, we leverage the RoCC extension of the Rocket core, which adds the support for decoding and executing custom instructions to the Rocket core's pipeline.

[5]According to our evaluations, maintaining PKR information during context switches incurs less than 1% performance overhead.

the segmentation fault with the pkey information to accurately reflect the cause of the fault to the developer and assist with debugging.

IV. SealPK: Sealing Features

As mentioned before, Intel MPK does not protect the allocated domains, their associated pages, and their permission bits from tampering by an attacker. In this section, we describe three novel sealing features to protect against such tampering. To clarify the defensive capabilities of these features, consider the example shown in Figure 3. In this example, a software developer writes a program that handles sensitive financial records. The `Main` function (written in-house) initially allocates the memory pages for the financial record (`log`) as `readable-writable` and assigns a protection key to these pages. Following the principle of the least privilege, the initial value of the pkey restricts the permission to `read-only` pages. In this example, `Func-A` updates the contents of the `log`. We assume that this function is developed in-house and has access to the pkey. Prior to writing the sensitive financial information into the `log`, `Func-A` modifies the domain permission of the `log` to `write-only`. For performance reasons, the software developer leverages third-party untrusted libraries in the implementation of `Func-B`, `Func-C`, and `Func-D`. `Funct-B` reads the `log` and returns a sorted copy of the `log`. `Func-C` does not have access to the `log`, instead it receives a list of prices and converts them to a different currency. `Funct-D` reads the `log` and prints all the transactions of a specific account. Hence, `Func-B` and `Func-D`, can only access the `log` as **read-only** memory. For security reasons, the untrusted functions are not aware of the pkey value. In the rest of this section, we explain how each of our sealing features protects the `log` against potential attacks originating from the untrusted components.

Sealing the domain: In this scenario, `Func-B` is a malicious third-party component, which receives the `log` as a `read-only` input. `Funct-B` is supposed to read the `log` and return a sorted copy of it. However, as shown in Figure 3, this untrusted component allocates a new `readable-writable` pkey, invokes the `mprotect` system call and assigns the new pkey to the `log`. In this way, `Func-B` can falsify the financial records stored in the `log`. Unfortunately, the developer who uses this untrusted function does not have access to its source-code and is unaware of its maliciousness. In this scenario, Intel MPK is not capable of preventing this malicious modification to the `log` within the same thread. To prevent such unauthorized modifications, we provide a domain sealing option by adding a `sealed_domain` map to the kernel. We modify the `pkey_mprotect` system call to check the `sealed_domain` map prior to modifying a domain's pkey. Once a domain is sealed, `pkey_mprotect` prevents any further modifications to PTE permissions as well as the pkey value, efficiently throwing such attacks.

Sealing pages: We assume that after the initialization step in the `Main` function, no more pages will be added to the protection domain. Consider a scenario where `Func-C`, a malicious third-party component, aims to crash this financial

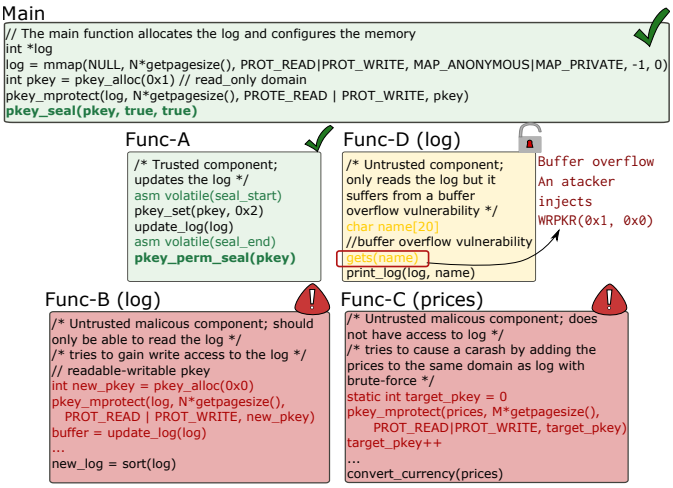

Fig. 3. Example scenario for our sealable features: The red texts in `Func-B` and `Func-C` show an effort to attack the pkeys, the yellow texts in `Func-D` show a vulnerability that can be leveraged by an attacker to compromise the pkey permissions. The green texts in the `Main` and `Func-A` functions show our sealing features to protect the domain, its associated pages, and its permissions from unauthorized modifications.

application. Crashing the application at run-time could lead to denial-of-service and financial losses. `Func-C` does not have access to the `log`; it only receives a list of prices and converts them from one currency to another one. This price list does not include any sensitive information; hence, `Func-A` does not assign a protection domain to it. In this example, in each call, the malicious `Func-C` adds the pages associated with the price list to a different domain, hoping that the new domain would restrict the `read` permission. As a result, after the price list is assigned with the same pkey as the `log`, once `Func-A` tries to read the price list the program crashes with a segmentation fault. Intel MPK cannot prevent this issue within the same thread; similarly, our domain sealing feature is not sufficient in this scenario. To ensure that no more pages can be added to a domain (either by mistake or by a malicious component), we provide a page sealing option by adding a `sealed_page` map to the kernel, indicating whether the pages associated with each pkey are sealed. We modify the `pkey_mprotect` system call to check the `sealed_page` map and only allow adding new pages to a pkey domain if the associated pages of that domain are not sealed. As shown in Figure 3, we add a new system call, `pkey_seal(int pkey, bool seal_domain, bool seal_page)`, which allows the programmer to seal a domain and/or its associated pages. Note that once a domain or its associated pages are sealed, the seal cannot be broken unless the corresponding pkey and all its associated pages are freed.

Sealing permissions: In this scenario, we assume `Func-D` is a third-party component, suffering from a buffer overflow vulnerability. As shown in Figure 3, an attacker can leverage this vulnerability to inject a `WRPKR` instruction at run-time and modify the permission bits of the `log` to `readable-writable`. Subsequently, the attacker can falsify the sensitive contents of the financial record. Intel MPK does

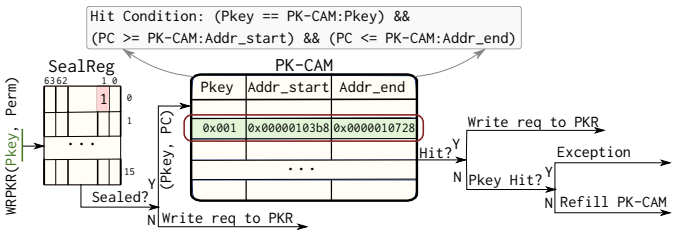

Fig. 4. High-level view of SealPK's hardware support to seal pkey permissions.

not protect pkey permissions against control-flow hijacking attacks that leverage the `WRPKRU` instruction. To prevent such a tampering, we provide a permission sealing feature, which allows the developer to restrict the execution of the `WRPKR` instruction to a specified range of memory addresses. In this example, we aim to restrict the occurrence of the `WRPKR` instruction to the address range of `Func-A`.

At hardware level, as shown in Figure 4, we keep track of sealed pkey permissions using a local memory, called `SealReg`. We modify the Rocket core's pipeline to consult `SealReg` prior to executing a `WRPKR` instruction. If the permission bits of the pkey are sealed, the `WRPKR` instruction is only allowed in the permissible range, specified by the developer. We leverage a Content-Addressable Memory (CAM) like structure, named `PK-CAM`, to cache the permissible range of each pkey. If the pkey information is available in `PK-CAM` but the current address of the `WRPKR` instruction is not in the permissible range, then SealPK prevents the execution of `WRPKR` and causes an exception. If `PK-CAM` does not include the pkey information, we will refill `PK-CAM`.[6]

We also provide the software support for sealing the permissions. We provide two new custom instructions, i.e., `seal_start` and `seal_end`, to specify the contiguous permissible range of each pkey. Although these instructions can be added to the source code (Figure 3), the more efficient way of using them is by a compiler pass or through run-time mechanisms such as ld-preload. After specifying the start and end addresses of a permissible range for `WRPKR`, the developer has to invoke a newly added system call (`pkey_perm_seal`) to seal the permissions. This system call leverages a custom instruction, which is only accessible to the supervisor mode, to seal the permission bits by updating the `SealReg` and `PK-CAM`. We modify the Linux kernel to maintain the `SealReg` information as well as permissible range of each pkey during context switches for each process. Note that `SealReg` and the permissible range of a pkey are implemented similar to a one-time fuse, i.e., they can only be written once for each process. Hence, after configuration, the permission sealing feature cannot be modified. The simplicity and efficiency of our permission sealing feature distinguishes our work from existing works focused on preventing the manipulation of a domain's permissions by an attacker, e.g., [10] and [18].

[6]Currently, we trigger an interrupt and insert the pkey and its permissible range to `PK-CAM` in the OS interrupt handler. As part of our future work, we plan to delegate this interrupt to user level and provide a secure software library to update `PK-CAM`.

By leveraging SealPK's sealing features, the software developer can implement a **tamper-proof** `log` of financial records in the face of buggy and malicious third-party components.

V. Evaluation

A. Experimental Setup

We use the Chisel HDL [4] to implement SealPK on a RISC-V Rocket core [3], configured with a 16KB L1 instruction and data caches. We add the OS support for SealPK to the Linux kernel v4.15. As a case study, we implement an isolated shadow stack using LLVM front-end and back-end passes. We use Clang v.7 and v.8 for our front-end and back-end passes, respectively. We prototype our hardware design with the full software stack on a Xilinx Zedboard FPGA.

For performance evaluation, we use RISC-V LLVM to cross-compile 6 applications (out of 12) from SPECint2000 [11], 4 applications (out of 12) from SPECint2006 [12], and 7 applications from MiBench [9] benchmark suites. Due to compilation issues and memory limitations of our FPGA, we were not able to successfully cross-compile and run all the applications from these benchmark suites. In particular, for SPECint2000, we got a segmentation fault for the baseline execution of `vortex` and `gcc`, and faced LLVM cross-compilation issues for the remaining 4 applications. For SPECint2006, with the baseline code, we got an out of memory error for `mcf` and a segmentation fault for `gcc`. We faced various LLVM cross-compilation issues for the remaining 6 applications. Note that RISC-V LLVM is still not as mature as GCC support for RISC-V. In our evaluations, we use the `large` inputs for MiBench and evaluate SPECint2000 and SPECint2006 applications using `test` inputs.[7]

B. Case Study: An Isolated Shadow Stack

To demonstrate the effectiveness of SealPK, as a case study, we use SealPK to protect an isolated shadow stack that prevents Return-Oriented Programming (ROP) attacks. An ROP attack is a contemporary code-reuse attack that allows an attacker to execute arbitrary code by overwriting the return addresses on the stack. A shadow stack protects the return addresses by storing them in a separate memory. It is imperative to guarantee the integrity of the shadow stack [5], i.e., the shadow stack area should be an **isolated** area within the process' address space to prevent attackers from modifying it. We isolate the shadow stack memory in a protection domain. Once the shadow stack memory is allocated and assigned to a domain, no more pages will be added and the protection domain stays the same during the process execution. We leverage the domain and page sealing features to protect the allocated domain and pages of the shadow stack from further modifications (similar to scenarios described in Section IV) after the initial configuration.

For the shadow stack implementation, we first implement a baseline front-end pass LLVM plugin [7]. This front-end pass allocates a memory area for the shadow stack and instruments

[7]We use the `test` inputs for SPEC evaluations due to the memory limitation of our FPGA board (256MB) as well as the long execution time of the benchmarks for the `mprotect` comparison point (multiple days).

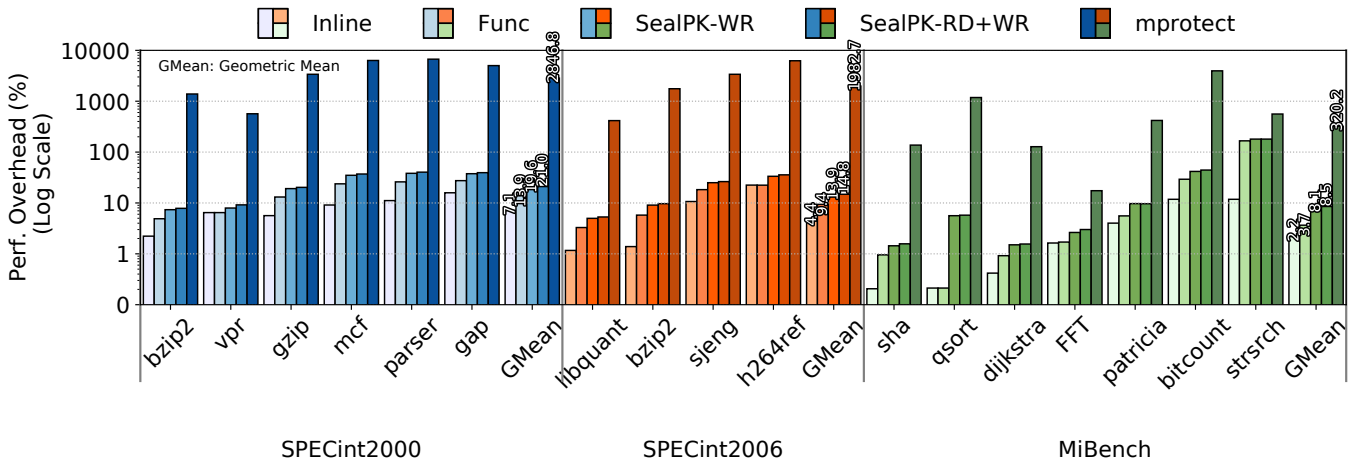

Fig. 5. Performance overhead of various LLVM-based shadow stack implementations for SPECint2000, SPECint2006, and MiBench benchmarks. The `Inline` implementation is a front-end LLVM pass, where the shadow stack capability is inserted as an inline code. The `Func` implementation uses a function call in the front-end pass rather than an inline code. `SealPK-WR` is implemented as a back-end pass built upon `Func`, where it writes the new value of pkey permission bits without maintaining the rest of the permission bits. `SealPK-RD+RW` adds the support to read the corresponding row of the pkey before updating it. `mprotect` is implemented as an inline front-end pass by invoking `mprotect` system call before and after writing the return address into the shadow stack.

the prologue and epilogue of each function to push the original return address into the shadow stack memory and pop the shadow return address from that memory, respectively. To isolate the shadow stack, we modify the front-end pass to allocate a pkey and to assign it to the shadow stack memory pages. To protect the shadow stack from modifications, we initialize the pkey as `read-only`. We implement a RISC-V back-end pass to temporarily update the pkey permission to `readable-writable` in the prologue, where we push the return address into the shadow stack. Right after pushing the return address, the back-end pass disables the pkey write permission. Our back-end pass inserts the required `RDPKR` and `WRPKR` instructions to update the pkey's permission bits. We can leverage our permission sealing feature to restrict the `WRPKR` occurrences to the memory range of the back-end pass.

In our evaluations, we measured the total execution time of an application as our performance metric. For the baseline, we compiled the benchmarks using Clang v.8 without applying any passes and ran the benchmarks on an unmodified core and Linux kernel. We ran each application three times and report the geometric mean of the execution times.

Figure 5 shows the performance overhead of various shadow stack implementations compared to the baseline. `Inline` and `Func` are front-end LLVM passes that cannot guarantee the integrity of the shadow stack; hence, the shadow stack memory remains unprotected. `SealPK-WR` and `SealPK-RD+RW` are isolated shadow stack implementations, leveraging SealPK in a back-end pass. `mprotect` is our comparison point, an isolated shadow stack implemented by leveraging the `mprotect` system call. As expected, using `mprotect` incurs considerable performance overhead, i.e., 2875.62%, 1982.70%, and 320.21%, on average, for SPEC2000, SPEC2006, and MiBench, respectively, which makes it an infeasible option. `mprotect` requires a context switch into the kernel, followed by a full page table walk to change the permissions of all the specified pages, and then a TLB flush. On the contrary, leveraging SealPK to implement an isolated shadow stack uses a user-space instruction to modify the pkey permission bits. `SealPK-RD+RW`, has an average of 21.00%, 14.81%, and 8.52% performance overhead for SPEC2000, SPEC2006, and MiBench applications, respectively.

TABLE I
THE FPGA UTILIZATION OF SEALPK COMPARED TO THE BASELINE ROCKET CORE.

	Baseline		Rocket Core + SealPK	
	Used	Utilization	Used	Utilization
Total Slice Luts	32030	60.21	35019	65.83
Luts as logic	30907	58.1	33852	63.63
Luts as Memory	1123	6.45	1167	6.71
Slice Registers as Flip Flop	16506	15.51	19392	18.23

C. Hardware Overhead of SealPK

Table I shows the FPGA utilization of adding SealPK to the Rocket core compared to the baseline unmodified Rocket core. In our FPGA prototype, enhancing Rocket core with SealPK increases the LUT and FF utilization by 5.62% and 2.72%, respectively.[8] The main source of area and power overhead for SealPK is `PKR`, a 2Kb local memory. Accordingly, we estimate that our power overhead is also less than 5%, even when considering a 100% access rate to `PKR`. In our FPGA evaluation, the Rocket core operated with a maximum frequency of 25 MHz (both in the baseline and the enhanced version with SealPK experiments).[9] According to our FPGA place and route results, SealPK's modifications to the Rocket core did not change the critical path.

[8]Note that the reported LUT and FF overhead includes the resource utilization of adding RoCC custom instruction support to the Rocket core.

[9]Note that an ASIC implementation of the Rocket core can perform with a target frequency of 1 GHz.

VI. Related Work

There is a considerable amount of prior work on intra-process memory isolation. A Software Fault Isolation (SFI) technique [20] instruments each memory access by address masking instructions to prevent unintended memory accesses; however, it suffers from large performance overhead. Prior to Intel MPK, CODOMs [19] and CHERI [23] proposed efficient capability-based systems, which require significant and invasive hardware modifications. IMIX [8] enables secure data encapsulation by minimally extending the x86 ISA with secure load and store instructions. To address Intel MPK's limitations, Hodor [10] and ERIM [18] combine Intel MPK with binary inspection to prevent reusing of `WRPKRU` instruction by an attacker. The sealing permission feature of SealPK provides a similar capability by restricting valid `WRPKR` instructions to a contiguous range of memory addresses for each pkey. Although our sealing feature is limited to one valid memory range for each pkey, its simplicity and efficiency distinguishes our work form Hodor and ERIM. To allow the occurrence of `WRPKR` instructions in more than one trusted component, we can rely on a CFI technique for the RISC-V Rocket core [6] to protect `PKR` from manipulation by an attacker. libmpk [13] and Xu et al. [24] provide a software-based and a hardware-based virtualization technique, respectively, to address the limited number of pkeys. We can leverage such virtualization techniques to support more than 1024 domains for SealPK.

Donky [17] provides a secure user-space software framework to protect the domain permissions against CFI attacks without relying on binary inspection or CFI. Donky proposes a pkey extension for RISC-V ISA, and implements it on the Ariane core [1]. Similar to SealPK, Donky uses the 10 unused bits of Sv39 PTEs to store the pkeys; however, Donky relies on a 64-bit CSR (managed by a software library) to store the permission bits of only 4 pkeys at a time. If the pkey of the accessed memory address is not loaded into that CSR, Donky requires extra cycles for the software library (which stores all the pkey information) to load the missing pkey and its permission into the register. In our design, we access `PKR` in the same cycle as page-table permission checks. The permission sealing feature of SealPK allows us to protect a domain against CFI attacks in cases where the valid `WRPKR` instructions occur in contiguous memory addresses. In addition to this feature, SealPK provides two other novel sealing features to prevent a domain and its associated pages from tampering.

VII. Conclusion

In this paper, we proposed an efficient intra-process memory isolation technique (SealPK) for a RISC-V processor, which supports up to 1024 domains. In our design, we provided three novel sealing features to protect a domain, its associated pages, and its permission bits from unauthorized modifications. To address the pkey use-after-free problem, we used an OS-level lazy de-allocation approach. We prototyped RISC-V Rocket + SealPK on an FPGA with full software stack, and demonstrated the efficiency of SealPK by securing a shadow stack.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1916393.

References

[1] Ariane RISC-V CPU. [online] https://github.com/lowRISC/ariane, 2018.
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.
[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The Rocket Chip generator. *EECS Department, UCB, Tech. Rep. UCB/EECS-2016-17*, 2016.
[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proc. DAC*, 2012.
[5] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *Proc. S&P*, 2019.
[6] Sadullah Canakci, Leila Delshadtehrani, Boyou Zhou, Ajay Joshi, and Manuel Egele. Efficient context-sensitive cfi enforcement through a hardware monitor. In *Proc. DIMVA*, 2020.
[7] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. PHMon: A programmable hardware monitor and its security use cases. In *Proc. USENIX Security*, 2020.
[8] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *Proc. USENIX Security*, 2018.
[9] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. WWC*, 2001.
[10] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proc. ATC*, 2019.
[11] John L Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7), 2000.
[12] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
[13] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proc. ATC*, 2019.
[14] ARM. Arm architecture reference manual ARMv7-A and ARMv7-R edition. 2018.
[15] IBM Corporation. Power isa version 3.0b. 2017.
[16] Intel Corporation. Intel 64 and ia-32 architectures software developers manual. 2019.
[17] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient in-process isolation for RISC-V and x86. In *Proc. USENIX Security*, 2020.
[18] 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 *Proc. USENIX Security*, 2019.
[19] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with code-centric memory domains. In *Proc. ISCA*, 2014.
[20] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proc SOSP*, 1993.
[21] Andrew Waterman, Krste Asanovic, and SiFive Inc. The RISC-V instruction set manual Volume II: Privileged architecture, version 1.12-draft. Technical report, UCB, 2020.
[22] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume i: Base user-level ISA. *UCB, Tech. Rep. UCB/EECS-2011-62*, 2011.
[23] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. S&P*, 2015.
[24] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *Proc. ISCA*, 2020.