# SealPK: 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. Recently, Intel introduced a new hardware feature for intra-process memory isolation, called Memory Protection Keys (MPK). The limited number of unique domains (16) provided by Intel MPK prohibits its use in cases 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. Additionally, we devise three novel sealing features to protect the allocated domains, their associated pages, and their permissions from modifications or tampering by an attacker. We demonstrate the efficiency of SealPK by leveraging it to implement an isolated secure shadow stack on an 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. While the user-space inter-process isolation protects processes from one another, the intra-process isolation of various software components has been a challenge.

Recently, Intel proposed a hardware feature, called Memory Protection Keys (MPK) [10], to efficiently support intra-process memory isolation. Intel MPK allows the user to create a protection domain by assigning a protection key (pkey) to a group of memory pages, and it provides a user-space instruction (`WRPKRU`) to update the associated permission of a domain. However, Intel MPK suffers from security and scalability issues. In terms of security, Intel MPK suffers from pkey use-after-free vulnerability [9]. 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 instruction 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 OpenSSL [9] require more than 1000 pkeys.

In this paper, we propose an efficient intra-process memory isolation capability, called SealPK, leveraging the Open RISC-V Instruction Set Architecture (ISA) [13]. SealPK provides a per-page protection key and supports up to 1024 domains (64× more than Intel MPK). 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. 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 specific contiguous range of memory addresses, e.g., a trusted component. Any attempt to execute a `WRPKRU` instruction from outside of the specified range would lead to a hardware exception. 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 [1] 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. *We open-source our design at https://github.com/bu-icsg/SealPK.*

In the rest of this paper, we discuss SealPK's design, sealing features, and evaluation in Section II, III, and IV, respectively. Section V discusses the related work and Section VI concludes the paper. We provide a more detailed description of the SealPK design, implementation, and evaluation in [5].

## II. SealPK: Design

### A. Hardware Design

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 of the Sv39 PTE to store the pkey. Figure 1 demonstrates our hardware modifications to support SealPK. We add a new entry to each line of the Data Translation Lookaside Buffer (DTLB) to store the corresponding 10-bit pkey of each virtual page. Hence, SealPK supports up to 1024 domains. 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

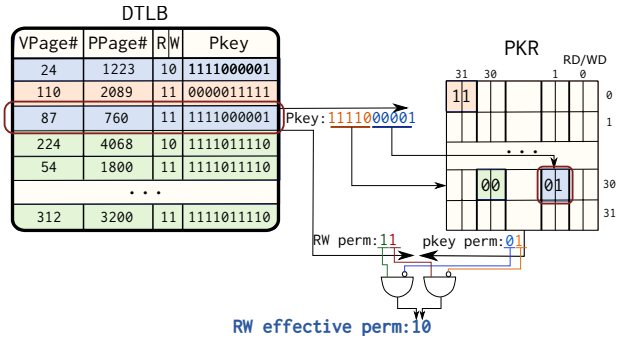


Fig. 1. *Modified MMU of the RISC-V Rocket core.*

privilege, unlike Intel MPK and previous works, our design enables a `write-only` page, which can in turn reduce the attack surface. 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 memory to store the permission bits. This memory, called `PKR` (Figure 1), consists of 32 rows, where each row stores the permission bits of 32 pkeys. We utilize the custom instruction extension of RISC-V ISA [13] to define two new instructions, `RDPKR` and `WRPKR`, to read from and write to `PKR`.

We provide a control logic to determine the effective permission bits of each data memory access. Consider the example shown in Figure 1, 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 (`1111000001`) 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. This 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 (with negligible performance overhead). Furthermore, we modify the RISC-V page fault handler in the Linux kernel to identify a page fault caused by a pkey permission violation.

## III. 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 2. 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. In the rest of this section, we explain how each sealing feature 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, 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`. 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

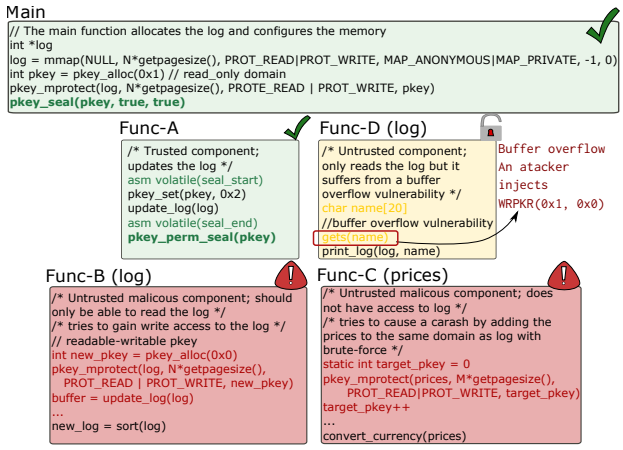


Fig. 2. *Example scenario for our sealable features.*

protection domain. Consider a scenario where `Func-C` aims to crash this financial application, which 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, 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. As shown in Figure 2, 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` suffers from a buffer overflow vulnerability. 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 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 `WRPKR` to a specified range of memory addresses (e.g., `Func-A`).

At hardware level, 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.

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 instruction can be added to the source code (Figure 2), 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. 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.

## IV. Evaluation

### A. Experimental Setup

We use the Chisel HDL [2] to implement SealPK on a RISC-V Rocket core [1]. 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 (Clang v.7) and back-end (Clang v.8) passes. 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 [7] and 4 applications (out of 12) from SPECint2006 [8] 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.

### B. Case Study: An Isolated Shadow Stack

As a case study, we use SealPK to protect an isolated shadow stack that prevents Return-Oriented Programming attacks. 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 [3], 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 III) after the initial configuration.

For the shadow stack implementation, we first implement a baseline front-end pass LLVM plugin. This front-end pass

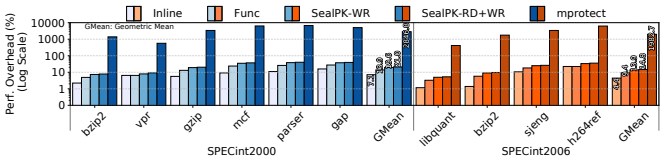


Fig. 3. *Performance overhead of LLVM-based shadow stack implementations (`test` inputs). `Func` implementation uses a function call in the front-end pass rather than an inline code (`Inline`). `SealPK-WR` is implemented as a back-end pass built upon `Func`, where it writes the new value of pkey permission 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.*

allocates a memory area for the shadow stack and instruments 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.

To evaluate the performance overhead of SealPK, in our experiments, we used the total execution time of an application as our performance metric. We ran each application three times and report the geometric mean of the execution times. Figure 3 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% and 1982.70%, on average, for SPEC2000 and SPEC2006, respectively, which makes it an infeasible option. `SealPK-RD+RW`, has an average of 21.00% and 14.81% performance overhead for SPEC2000 and SPEC2006 applications, respectively. In terms of area overhead, enhancing Rocket core with SealPK increases the LUT and FF utilization of our FPGA by 5.62% and 2.72%, respectively.

## V. Related Work

To address Intel MPK's limitations, Hodor [6] and ERIM [12] 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 [4] to protect `PKR` from manipulation by an attacker. libmpk [9] and Xu et al. [14] 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 [11] provides a secure user-space software framework to protect the domain permissions against manipulations without relying on binary inspection or CFI. Donky proposes a pkey extension for RISC-V ISA implemented on Ariane core. Similar to SealPK, Donky uses the 10 unused bits of Sv39 PTEs to store the pkeys but it 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 to load the missing pkey and its permission into the register. 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.

## VI. 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.

## Acknowledgments

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

## References

[1] Asanovic, K. et al. The Rocket Chip generator. *EECS Department, UCB, Tech. Rep. UCB/EECS-2016-17*, 2016.
[2] Bachrach, J. et al. Chisel: constructing hardware in a scala embedded language. *Proc. DAC*, 2012.
[3] Burow, N. et al. SoK: Shining light on shadow stacks. *Proc. S&P*, 2019.
[4] Canakci, S. et al. Efficient context-sensitive CFI enforcement through a hardware monitor. *Proc. DIMVA*, 2020.
[5] Delshadtehrani, L. et al. Efficient sealable protection keys for RISC-V. *arXiv preprint arXiv:2012.02715*, 2020.
[6] Hedayati, M. et al. Hodor: Intra-process isolation for high-throughput data plane libraries. *Proc. ATC*, 2019.
[7] Henning, J.L. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7), 2000.
[8] Henning, J.L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
[9] Park, S. et al. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). *Proc. ATC*, 2019.
[10] Intel Corporation. Intel 64 and ia-32 architectures software developers manual. 2019.
[11] Schrammel, D. et al. Donky: Domain keys–efficient in-process isolation for RISC-V and x86. *Proc. USENIX Security*, 2020.
[12] Vahldiek-Oberwagner, A. et al. ERIM: Secure, efficient in-process isolation with protection keys (MPK). *Proc. USENIX Security*, 2019.
[13] Waterman, A. et al. The RISC-V instruction set manual, volume i: Base user-level ISA. *UCB, Tech. Rep. UCB/EECS-2011-62*, 2011.
[14] Xu, Y. et al. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. *Proc. ISCA*, 2020.