FlexFilt: Towards Flexible Instruction Filtering for Security

Leila Delshadtehrani
delshad@bu.edu
Boston University
Boston, Massachusetts, USA

Sadullah Canakci
scanakci@bu.edu
Boston University
Boston, Massachusetts, USA

William Blair
wdblair@bu.edu
Boston University
Boston, Massachusetts, USA

Manuel Egele
megele@bu.edu
Boston University
Boston, Massachusetts, USA

Ajay Joshi
joshi@bu.edu
Boston University
Boston, Massachusetts, USA

ABSTRACT

As the complexity of software applications increases, there has been a growing demand for intra-process memory isolation. The commercially available intra-process memory isolation mechanisms in modern processors, e.g., Intel's memory protection keys, trade-off between efficiency and security guarantees. Recently, researchers have tended to leverage the features with low security guarantees for intra-process memory isolation. Subsequently, they have relied on binary scanning and runtime binary rewriting to prevent the execution of unsafe instructions, which improves the security guarantees. Such intra-process memory isolation mechanisms are not the only security solutions that have to prevent the execution of unsafe instructions in untrusted parts of the code. In fact, we identify a similar requirement in a variety of other security solutions. Although binary scanning and runtime binary rewriting approaches can be leveraged to address this requirement, it is challenging to efficiently implement these approaches.

In this paper, we propose an efficient and flexible hardware-assisted feature for runtime filtering of user-specified instructions. This flexible feature, called FlexFilt, assists with securing various isolation-based mechanisms. FlexFilt enables the software developer to create up to 16 instruction domains, where each instruction domain can be configured to filter the execution of user-specified instructions. In addition to filtering unprivileged instructions, FlexFilt is capable of filtering privileged instructions. To illustrate the effectiveness of FlexFilt compared to binary scanning approaches, we measure the overhead caused by scanning the JIT compiled code while browsing various webpages. We demonstrate the feasibility of FlexFilt by implementing our design on the RISC-V Rocket core, providing the Linux kernel support for it, and prototyping our full design on an FPGA.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**.

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488019

KEYWORDS

Hardware security, OS security, memory protection domains

ACM Reference Format:
Leila Delshadtehrani, Sadullah Canakci, William Blair, Manuel Egele, and Ajay Joshi. 2021. **FlexFilt: Towards Flexible Instruction Filtering for Security. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages.** https://doi.org/10.1145/3485832.3488019

1 INTRODUCTION

Today's software is a complex mixture of trusted code written in-house and untrusted code such as third-party libraries and application plugins. The coexistence of trusted code with potentially vulnerable or malicious untrusted code in the same process could comprise the confidentiality and integrity of the trusted code. To limit the effects of bugs and security vulnerabilities, a variety of security solutions partition sensitive data and code into isolated components. Researchers have leveraged various techniques including Operating System (OS)-based [10, 44] and virtualization-based techniques [8, 40, 46], hardware-based trusted execution environments [4, 23], and memory protection domains [32, 58, 61] to enforce isolation.

To guarantee the integrity of the isolation, the above-mentioned security solutions have to prevent an untrusted component from accessing or modifying the isolated components. To this end, a variety of prior work [4, 5, 14, 15, 24, 30, 32, 49, 58, 61, 65–67] faced a common challenge, i.e., *preventing the execution of various unsafe instructions in untrusted parts of the code (either in user space or kernel space).* Such unsafe instructions could compromise the integrity of the isolation mechanisms by modifying access permissions, disabling protections, gaining higher privilege, etc. To prevent the execution of such unsafe instructions, previous works have leveraged various approaches such as Control-Flow Integrity (CFI) [14, 24, 49] and binary scanning and binary rewriting [4, 5, 30, 32, 49, 61, 65, 67]. As currently existing CFI solutions [21, 25, 31, 35, 35, 45] have non-trivial performance overhead (> 10%), leveraging CFI to prevent the execution of unsafe instructions is expensive. Additionally, CFI cannot simply be leveraged in self-modifying or dynamically generated code [58]. As shown by previous works, binary scanning and binary rewriting approaches can filter unsafe instructions in static code [32, 61]. However, an efficient implementation of binary scanning and binary rewriting, especially for dynamically generated code, is challenging.

To clarify the challenges involved in binary scanning and binary rewriting for preventing the execution of unsafe instructions, consider the case of memory protection keys. In recent years, various processors including ARM [51], IBM Power [54], and Intel [55] provided hardware-assisted memory protection keys, an extension to page-based memory permissions. Accordingly, a software developer can associate a group of memory pages with the same protection key to create a memory protection domain. Subsequently, the software developer can update the access permission of all the pages in the same domain by updating the corresponding permission bits of the protection key. While ARM and IBM Power only allow the OS to modify the corresponding permission of a protection key, Intel MPK allows a user-space process to make this modification. Intel MPK stores the corresponding permission bits of all the protection keys in a new thread-local register, called protection key right register (PKRU). Modifying the permission bits of a protection domain requires writing into PKRU leveraging a new user-space instruction, called `WRPKRU`. The execution of the `WRPKRU` instruction is fast but an untrusted component can gain access permission to any protection domain by simply writing into PKRU through executing the `WRPKRU` instruction.

To ensure that all the occurrences of `WRPKRU` instructions are safe, various approaches such as Hodor [32] and ERIM [61] rely on binary inspection and runtime binary rewriting. One of the main challenges in ensuring the safety of `WRPKRU` occurrences through binary inspection is the implicit (unintended) occurrences of the instruction. Such implicit occurrences could be the result of the `WRPKRU` instruction forming across the boundary between two consecutive instructions or as a sub-sequence of a longer instruction. An attacker can perform a control-flow hijacking attack to jump into the point that an implicit `WRPKRU` instruction occurs. To address this challenge, Hodor leverages debug registers to trigger a hardware watchpoint once an explicit or implicit `WRPKRU` is about to be executed. A vetting mechanism at kernel level allows the execution to continue only for safe occurrences of the `WRPKRU` instruction. ERIM intercepts each executable page and scans through the page for unsafe instructions; then, it enables the execute permission iff no unsafe occurrences exist. Otherwise, ERIM implements a runtime binary rewriting approach to rewrite the implicit occurrences of unsafe instructions.

The above-mentioned challenges for restricting the occurrence of `WRPKRU` instructions to safe locations indicate the requirement for an efficient approach to filter unsafe instructions at runtime. This requirement is not limited to memory protection domains, `WRPKRU` instructions, or x86 processors. We observed that a number of isolation-based security solutions, on different processor architectures, have to prevent the execution of various unsafe instructions in untrusted parts of the code. In the rest of this paper, we refer to the unsafe instructions as *target* instructions that should be filtered. Depending on the isolation mechanism, the target instructions could be privileged or unprivileged instructions. In x86 processors, other security solutions [30, 65] limit the execution of target instructions such as privileged `MOV CR0`, `MOV CR3`, and `VMRUN`. Prior work on ARM processors [4, 5, 67] prevent the occurrence of target instructions such as `MSR`, `LDC`, and `MCR` in untrusted parts of the code. Recent works leveraging memory protection domains on RISC-V architecture [14, 15, 58] have to limit the execution of the `WRPKRU` equivalent instruction.

The previous works are limited to filtering the execution of certain target instructions. In this paper, we strive to provide a **generalized** solution for filtering target instructions. Such a generalized solution should satisfy the following requirements: 1) flexibility to be applicable to a variety of instructions, 2) efficiency to be applicable at runtime, and 3) fine-granularity to be able to filter various parts of the code. To this end, we propose FlexFilt, an efficient and flexible hardware-assisted capability for runtime filtering of target instructions at page granularity. FlexFilt provides the generalized instruction filtering capability via two mechanisms, i.e., instruction protection domains and flexible hardware-level filters. FlexFilt enables the software developer to create instruction protection domains by assigning the same protection key to a group of executable pages. At the hardware level, FlexFilt provides configurable filters to prevent the execution of various user-defined instructions. The hardware-level filters can then be associated with instruction protection domains and subsequently prevent the execution of target instructions in memory pages assigned to the corresponding domain. FlexFilt is an efficient hardware-assisted feature and incurs negligible performance overhead for filtering target instructions at runtime. FlexFilt satisfies all the previously mentioned requirements of a generalized instruction filtering solution. In addition to filtering user-space instructions, FlexFilt is capable of filtering privileged instructions (i.e., supervisor mode and hypervisor mode).

To demonstrate the feasibility of FlexFilt's design, we leverage the RISC-V open Instruction Set Architecture (ISA) [64] and implement FlexFilt on the RISC-V Rocket core [3]. To evaluate FlexFilt in a realistic environment, we provide the OS support for our hardware design and prototype our full design (including hardware, OS, and user-space software) on the Xilinx Zedboard FPGA [52]. In summary, our contributions are as follows:

- We propose FlexFilt, a flexible hardware-assisted feature, which enables a software developer to efficiently prevent the execution of various instructions at a page granularity.
- To demonstrate the feasibility of our design, we implement a practical prototype, consisting of the RISC-V Rocket core enhanced with FlexFilt and the Linux kernel support for FlexFilt, on an FPGA.
- To illustrate the effectiveness of FlexFilt compared to binary scanning approaches, we measure the overhead of scanning JIT compiled bytes generated by V8 JavaScript engine while browsing various webpages.

In the spirit of open science and to facilitate the reproducibility of our experiments, we will submit our work for artifact evaluation and open-source our full design.

2 MOTIVATION AND RELATED WORK

As mentioned before, a variety of previous works faced the challenge of preventing the execution of target instructions in untrusted parts of the code. Table 1 lists these works, their target instructions, and the approaches they used for filtering the instructions.

Table 1: Comparison of previous works that prevent the execution of target instructions at runtime.

Architecture	Mechanism	Target Instructions	Privilege Level	Filtering Approach
x86	ERIM [61]	WRPKRU, XRSTOR	User	Binary inspection and rewriting
	Hodor [32]	WRPKRU	User	Binary scanning and hardware watchpoints
	libmpk [49]	WRPKRU	User	CFI or relying on an approach like ERIM
	Xu et al. [66]	WRPKRU	User	Relying on approach like Hodor or ERIM
	Donky [58]	WRPKRU	User	Hardware-assisted call-gates
	IMIX [24]	Extended instruction (SMOV)	User	CFI
	Fidelius [65]	MOV CR0, MOV CR4, WRMSR, VMRUN, MOV CR3	Supervisor	Binary scanning
	Underbridge [30]	WRPKRU, MOV CR3	User & Supervisor	Binary scanning and rewriting
ARM	Silhouette [67]	MSR	User	Binary scanning
	TZ-PKR [4]	LDC, MCR	Supervisor	Binary scanning
	SKEE [5]	N/A	Supervisor	Binary scanning
RISC-V	Donky [58]	N/A	User	Hardware-assisted call-gates
	SealPK [14]	Extended instruction (WRPKR)	User	Hardware-assisted instruction filtering
	FlexFilt	Various instructions	User & Supervisor	Hardware-assisted flexible filters

2.1 Instruction Filtering in Processors

2.1.1 x86 Processors. Recently, with the availability of Intel MPK [55] on high-end commercial x86 processors, researchers focused on deploying MPK in a secure way. To ensure the security of Intel MPK for intra-process memory isolation, it is necessary to prevent an untrusted component from executing `WRPKRU` or `XRSTOR` instructions, which might lead to unauthorized access to protected memory pages.[1] As an example, consider a scenario where the software developer aims to only allow the execution of `WRPKRU` instruction in trusted parts of the code. In this example, we assume that the software developer writes two trusted functions, namely `good_code1` and `good_code2`. She modifies the permission bits of her memory protection domains through `WRPKRU` instructions only in the two trusted functions and she wants to prevent the execution of `WRPKRU` in other parts of the code. As shown in Table 1, various recent works leveraged different approaches to prevent the unsafe execution of `WRPKRU` (and `XRSTOR`) instruction. Hodor [32] leverages binary scanning and hardware watchpoints to prevent the execution of unsafe `WRPKRU` instructions. ERIM [61] relies on binary inspection and binary rewriting techniques to prevent an unsafe execution of a `WRPKRU` or an `XRSTOR` instructions. libmpk [49] and the work by Xu et al. [66] address the scalability issue of Intel MPK using software-based and hardware-based virtualization techniques, respectively. These virtualization techniques [49, 66] rely on CFI or previous approaches such as ERIM and Hodor to filter unsafe `WRPKRU` instructions. Donky [58] uses a hardware-assisted call-gate mechanism to secure the domain transitions of MPK, without the need for binary scanning or CFI. IMIX [24] assumes the mitigation approaches such as CFI and Code-Pointer Integrity (CPI) [41] to prevent an attacker from reusing the trusted code containing `SMOV`, an extended instruction for secure load and store.

The need to filter target instructions in x86 architecture is not limited to user-space instructions protecting MPK. Fidelius [65] proposes a software-based extension to protect the Virtual Machine (VM) against an untrusted hypervisor. Fidelius utilizes binary scanning to restrict the execution of instructions that might hijack the control flow (e.g., `VMRUN`) or switch the address space (e.g., `MOV CR3`). Underbridgee [30] retrofits Intel MPK for kernel space isolation. To prevent the bypassing of the isolation enforced by MPK, Underbridge leverages binary scanning and rewriting. Subsequently, Underbridge ensures that system servers do not contain any explicit or implicit `CR3` instructions that modify the page table base register.

2.1.2 ARM Processors. Researchers have faced the instruction filtering requirement on ARM processors too. Silhouette [67] provides a protected implementation of the shadow stack on embedded ARM processors. Silhouette scans the code to ensure that it does not contain an instruction, such as `MSR`, that can be used to modify the program state without the need for a store instruction. TZ-PKR [4] provides a real-time protection of the OS kernel by leveraging ARM TrustZone [50]. SKEE [5] implements a light-weight framework for a secure kernel-level execution environment on ARM architectures, without relying on a higher privileged layer. To prevent the kernel from executing target privileged instructions, both TZ-PKR and SKEE scan the kernel executables looking for certain control instructions, such as `MCR` and `LDC`. These instructions are replaced with hooks that jump to a switch gate.

2.1.3 RISC-V Processors. Multiple recent works [14, 15, 58] provided memory protection keys for RISC-V. In addition to the x86 implementation, Donky [58] provides the intra-process memory isolation feature for RISC-V and leverages hardware-assisted call-gates to secure its implementation. Similarly, SealPK [14, 15] implements the memory protection keys for RISC-V. SealPK provides a hardware-assisted feature allowing the software developer to restrict the execution of the `WRPKRU` instruction to a contiguous range of memory addresses (e.g., one trusted function). Unlike the flexible design of FlexFilt, SealPK's implementation is limited to allowing the execution of a fixed instruction in only one trusted function.

2.2 Shortcomings of Existing Approaches for Instruction Filtering

As discussed before, a large number of previous works [4, 5, 30, 32, 61, 65, 67] rely on binary scanning to prevent the execution of target instructions. In CISC architectures such as x86, one of the challenges in filtering target instructions is the implicit (unintended) occurrences of these instructions. The target instructions can be formed implicitly across the boundary between instructions or as a sub-sequence of a longer instruction. Filtering the target instructions in RISC architectures, including ARM and RISC-V, is simpler because these architectures use fixed-length instructions. Although both ARM and RISC-V support compressed instructions (16-bit instructions), instructions cannot be loaded at any offset in

[1] `XRSTOR` restores the full or partial state of a processor's state during a context switch. The `XRSTOR` instruction can modify the contents of the `PKRU` register (which stores the permission bits of all the domains) by setting a specific bit in the `eax` register before executing the instruction [61].

the memory (unlike x86). As FlexFilt monitors and filters instructions at the execution stage, we are not concerned with the implicit vs explicit occurrences of instructions.

Although binary scanning approaches are efficient for the static code, binary scanning and subsequently binary rewriting can be costly for dynamically generate code. In Section 6, we analyze the overhead of scanning the Just-In-Time (JIT) compiled pages while browsing various websites and discuss the challenges faced by prior work to prevent the execution of target instructions in the JIT code. FlexFilt provides flexible instruction filters that allow the software developer to prevent the execution of target instructions on any page, irrespective of whether the page contains static code or dynamically generated code (e.g., JIT compiled), without relying on binary scanning and binary rewriting.

2.3 Watchpoints and Hardware Monitors

Most modern architectures provide a number of hardware watchpoints or debug registers. A hardware watchpoint is a debugging mechanism that allows the software developer to monitor a number of programmer-specified memory locations. Whenever the monitored locations are accessed, the hardware triggers an exception and traps into the debugger. Unfortunately, due the limited number of hardware watchpoints in modern architectures (e.g., only 4 watchpoints in x86), it cannot effectively be used as a fine-grained standalone solution to filter instructions at runtime.

In addition to commercially available hardware watchpoints, researchers have proposed a variety of dedicated [29, 68] and flexible hardware monitors [16, 18–20]. iWatcher [68] provides a large yet limited number of programmable hardware watchpoints while Greathouse et al. [29] propose an approach that supports an unlimited number of watchpoints. Even with an unlimited number of watchpoints, the software developer still has to utilize binary scanning to identify all the possible occurrences of target instructions and monitor the execution of each occurrence using a watchpoint. To leverage FlexFilt, the software developer only needs to specify the list of target instructions and the trusted parts of the code. Flexible hardware monitors such as PUMP [20], FlexCore [18], Harmoni [19], and PHMon [16] are capable of performing a variety of monitoring tasks. FlexCore and Harmoni have similar capabilities as PUMP; however, PUMP is more flexible and it has been more extensively adopted [12, 36, 57]. Although it is feasible to leverage PUMP to prevent the execution of target instructions, specifying the tag checking and propagation rules for PUMP is a challenging task. FlexFilt requires fewer and less invasive hardware modifications compared to PUMP's invasive hardware modifications on all stages of the CPU, caches, and main memory. PHMon can filter the execution of target instructions in a specific range of memory addresses leveraging a number of match units. However, the number of memory regions to filter target instructions is limited to the number of match units. Hence, unlike FlexFilt, PHMon cannot enforce a fine-grained instruction filtering at page granularity.

2.4 Hardware-Assisted Instruction Stream Customization

A large number of modern processors provide the Âţops capability to convert a complex ISA to simpler and easier to execute stream of instructions. Additionally, several processors such as IBM's DAISY [22], TransmetaâĂŹs Crusoe and Efficieon [13], and NvidiaâĂŹs Denver [9] implement a dynamic binary translator hardware and a software layer for performance optimization. A number of prior works provide the hardware support for runtime instruction customization and leverage it for functionalities beyond dynamic optimizations, e.g., safety/security checking and enforcement, profiling, and dynamic code decomposition [11, 59, 60]. DISE [11] is a programmable macro engine that translates user-defined instruction streams to customized streams at the decoder level. Importantly, DISE does not support per-domain or address-based instruction customization, which is a fundamental requirement in our instruction filtering use cases. Context-Sensitive Decoding (CSD) [59] enables program instructions to be dynamically translated into a customized set of Âţops. CSD can turn on and turn off the custom translations in different address ranges specified through a set of Model-Specific Registers (MSRs). The limited number of registers available for specifying the address ranges for instruction customization is in contrast with the requirements of some of our use cases, e.g., filtering `WRPKRU` in various untrusted functions. As modifying the contents of MSRs requires transitions to the kernel level, frequent runtime modification of these MSRs for addressing the limited number of available MSRs or handling the JIT compiled code could lead to high performance overheads. Context-Sensitive Fencing (CSF) [60] proposes a microcode level defense against Spectre attacks [39] by leveraging CSD.

In principle, hardware-assisted dynamic instruction customization approaches, such as DISE and CSD, can be complementary to FlexFilt upon detecting a target instruction. For example, one might be interested in replacing a target instruction with a sequence of safe instructions, e.g., a `NOP` instruction. Assuming that the imminent execution of a target instruction is an indication of an attack that undermines the security of the system, we currently terminate the program execution rather than replacing the target instruction with safe instructions. However, in the absence of the dynamic instruction customization support, we can provide additional flexibility by performing other operations in the exception handler instead of terminating the process, e.g., trap to a debugger.

3 BACKGROUND

We leverage the RISC-V open Instruction Set Architecture (ISA) to design, implement, and evaluate FlexFilt. In this section, we provide the background information on the RISC-V ISA [64]. In this paper, our focus is on commonly used 64-bit RISC-V processors (RV64). RISC-V ISA dedicates four `opcodes` for custom instruction-set extensions. Instructions with these `opcodes`, called `custom` instructions, are reserved for customization and will not be used by future standard extensions. We leverage RISC-V `custom` instructions to configure FlexFilt. The RISC-V ISA has unprivileged [62] and privileged [63] ISA specifications. Currently, the RISC-V ISA provides three privilege levels, i.e., user/application, supervisor, and machine modes. The highest level of privilege belongs to the machine mode, which is a mandatory privilege level for any RISC-V core.

For RV64, RISC-V specifies two page-based virtual memory systems, i.e., Sv39 and Sv48. Sv39 and Sv48 provide a 39-bit and a 48-bit virtual address space, respectively, where in both cases the address space is divided into 4KB pages. The privilege spec of RISC-V ISA specifies the virtual address translation process and the format of the Page Table Entry (PTE). Each PTE holds the mapping between a

virtual address of a page and its corresponding address of a physical frame. Bits 3-1 of each PTE are the page permission bits, indicating whether a page is readable, writable, and executable, respectively. The top 10 bits of an Sv39 and Sv48 PTE (bits 63-54) are reserved for future use, e.g., to facilitate research experiments [63]. The previous works on memory protection keys for RISC-V including Donky [58] and SealPK [14] leverage these 10 unused bits to store the memory protection key information.

In addition to access permissions stored in PTE, RISC-V ISA specifies the Physical Memory Protection (PMP) capability. PMP provides a per thread view (for each hart) that enables the programmable machine mode to limit the physical addresses that are accessible by software. PMP divides the physical memory address into up to 16 configurable regions, where each region can be configured with specific access permissions. At hardware level, a PMP unit utilizes machine-mode CSRs to specify the memory access permission (read, write, and execute) of each region. At runtime, PMP checks are applied to all the accesses in user and supervisor modes. Various previous works [37, 38, 42, 43, 53] leverage PMPs for providing an additional security layer. One could be tempted to implement FlexFilt on top of PMP. However, each PMP region is specified by a contiguous range of memory addresses and there are only 16 PMPs available. For our design, we are interested in creating instruction domains at page granularity, which is not feasible with 16 available PMPs. Hence, we do not build our instruction domains on top of the existing PMP feature.

4 THREAT MODEL

FlexFilt can be leveraged in a variety of security use cases introduced by prior work (see Table 1). In our work, for each use case, we follow the common threat model in the prior work. For intra-process memory isolation approaches, we assume that the untrusted parts of the code might contain vulnerabilities that an adversary can exploit to inject or reuse arbitrary instructions including the target instructions (e.g., `WRPKRU`). We do not assume any restrictions about what an attacker would do after a successful attack. We assume that the safe occurrences of target instructions in trusted parts of the code are surrounded by call gates or trampolines similar to the ones described in [32, 61], which protect these occurrences against control-flow hijacking attacks.

As the OS is responsible for allocating the instruction domains and maintaining FlexFilt's information, we assume the OS kernel is (partially) trusted. We assume all hardware components, including our modifications, are trusted and bug free. Hence, rowhammer, side-channel, and fault attacks are beyond the scope of this work.

5 DESIGN

In this section, we discuss FlexFilt's design goals, the challenges involved in implementing FlexFilt, and our solutions to address those challenges. As discussed before, we identified a common requirement for a flexible runtime instruction filtering capability in a variety of previous works. Unlike prior works that provide a solution capable of filtering a small number of specific target instructions, we strive to provide a generalized solution for the *runtime filtering of target instructions*. Such a generalized solution should be flexible, efficient, and fine-grained. To be compatible with existing OS-supported memory protections, we implement FlexFilt at page granularity, i.e., each executable page can apply a combination of the configured instruction filters. This design choice allows us to leverage the already existing OS-managed structures such as PTE as well as hardware structures such as Translation Lookaside Buffer (TLB) in our implementation. While, providing a finer granularity for instruction filtering requires substantial modifications at both OS-level and hardware-level.

We need to provide the OS support as well as a software API to enable a software developer to use FlexFilt. In the rest of this section, we will first discuss our hardware design choices, followed by the OS support for FlexFilt, and then the software support to configure our flexible instruction filters.

5.1 Hardware Design

In this section, we discuss the hardware design of FlexFilt.

5.1.1 Instruction Protection Domains. To leverage the existing OS-level and hardware-level structures for memory protection, we implement FlexFilt at page granularity. Inspired by the design of memory protection keys, we devise *instruction protection keys*, which enable us to simply divide the software code into trusted and untrusted executable partitions. The software developer can assign the same instruction protection key to a group of executable pages, which subsequently creates instruction protection domains. The existing memory protection keys such as Intel MPK are only applicable to data memory accesses, not instruction addresses. Here, our focus is on associating fetched instructions to protection domains according to the corresponding address of each instruction.

Prior work [14, 15, 58] leverages the 10 unused bits of the RISC-V Sv39/Sv48 PTE to store the memory protection key. Similarly, we can utilize these 10 unused bits to store the instruction protection keys, which provides up to 1024 instruction protection domains. Supporting a large number of data *memory protection domains* is a necessity in various use cases, such as Persistent Memory Object (PMO) [66] and OpenSSL [49]. However, supporting a large number of domains is not required for instruction protection domains. According to our literature review, previous works with the instruction filtering requirements only needed two instruction domains, i.e., a trusted and an untrusted domain. However, providing only two instruction protection domains could be restrictive for some use cases (e.g., the combination of various protection mechanisms). Intuitively, we would not need 1024 instruction domains, even if we apply all the protection mechanisms proposed by a variety of the previous works into a single system. As a trade-off for the number of instruction domains, we utilize the 4 lower bits of the 10 unused bits in the PTE to store the instruction protection keys (ipkey). Accordingly, FlexFilt supports up to 16 instruction protection domains, where each domain filters target instructions in the domain's corresponding pages.

5.1.2 Flexible Filters. One of the main design goals of FlexFilt is providing **flexible** instruction filters, capable of filtering various target instructions. To achieve this goal, we design each filter in an inherently flexible way. We leverage a bit-granular match/mask mechanism, similar to the matching mechanism used in the prior work [16, 17]. This design choice enables us to filter one specific instruction or a group of instructions with one `Flexible Filter`. As an example, consider various branch instructions, including `BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, and `BGEU`, in the RISC-V ISA. These branch

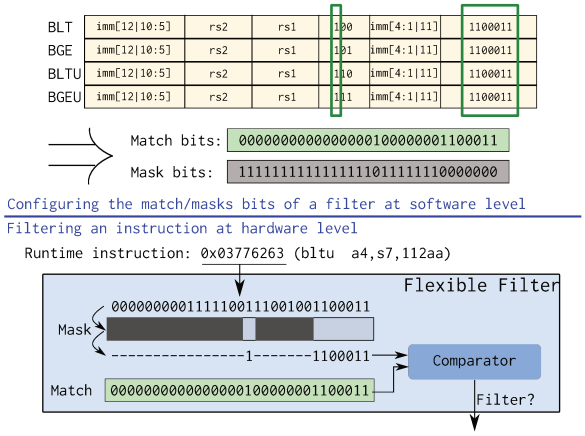

Figure 1: The `Flexible Filter` design, applied to a subset of RISC-V branch instructions.

instructions share the same opcode value (1100011) and they are distinguished based on the value of `funct3` bits. In this scenario, a flexible bit-granular instruction filter offers the user the option of filtering a specific branch instruction, a subset of the branch instructions, or all of the above-mentioned branch instructions.

Consider a scenario where the user is interested in filtering four of the previously mentioned branch instructions, i.e., BLT, BGE, BLTU, BGEU. Figure 1 shows the format of these instructions. The common bits of these instructions, specified with a green box, can be used for identifying them. In this example, the uncommon bits are `don't cares`. Accordingly, the software developer can simply describe the four branch instructions using `Match` and `Mask` bits. The `Match` bits specify the 32-bit value of an instruction identified as one of the four branch instructions and the `Mask` bits specify the `don't care` bits. At the hardware level, our `Flexible Filters` enforce such a matching/masking approach (the bottom part of Figure 1). The control logic of the `Mask` acts like a filter that blocks the masked parts of an instruction (specified by 1 bits in the `Mask` and shown by dark gray color in Figure 1) and passes through the rest of the instruction bits (shown by transparent gray color in Figure 1). The output of this control logic, which contains the `don't care` bits, is passed into a comparator module to be compared with the `Match`. If these two values match, then the `Flexible Filter` activates an output signal, indicating that current instruction should be filtered. For additional information on the filtering capabilities of our `Flexible Filters`, refer to Appendix A.

As mentioned before, an instruction protection domain is a group of executable pages assigned with the same instruction protection key. Each instruction protection domain prevents the execution of various user-defined target instructions in its corresponding pages. Ideally, we are interested in a flexible feature capable of filtering any number of target instructions in each domain. In reality, providing such a capability is not practical due to resource limitations and the substantial area and power overheads induced by such excessive flexibility. With a limited number of instruction filters for each domain, we consider two design options. First, each instruction domain has a fixed number of dedicated instruction filters. Second, there is a fixed number of shared instruction filters, and each instruction domain can apply a combination of these shared filters to its corresponding pages. Although the first option provides more flexibility in terms of filtering capabilities, it requires more hardware resources. Additionally, the instruction filter information for all the domains should be maintained by the OS during context switches. Considering the overheads involved with the first design option, we choose the second option in our design. We leave further investigation into the overheads involved in implementing the first design option as part of our future work.

By choosing the second design option, i.e., a fixed number of shared configurable instruction filters, the next design question we have to answer is the exact number of shared instruction filters. To choose the number of instruction filters, we examine the number of required filters in the previous works (listed in Table 1). Most of the previous works required to filter only one target instruction. In the worst case scenario, Fidelius [65] needed to filter the execution of three unique instruction types (five instructions in total). Considering the possibility of enforcing a combination of protection mechanisms, we choose to implement four shared instruction filters in our design.[2] Subsequently, each instruction domain can be configured to apply a combination of the shared instruction filters.

5.1.3 Microarchitecture Support. In our design, FlexFilt supports up to 16 instruction domains and provides four `Flexible Filters`, where a combination of these filters is applicable to each instruction domain. For each page, we specify the instruction domain by storing the ipkey in the 4 previously unused bits of PTE. For each instruction execution, in addition to checking the PTE permission bits (e.g., the X bit, which indicates that the page is executable), we need to determine if the instruction should be filtered. To this end, we first have to identify the corresponding domain of the instruction. At hardware level, the Instruction TLB (I-TLB) maintains the virtual to physical address translation of the instructions as well as their corresponding permission bits. We augment the I-TLB with a new field to store the associated ipkey of each virtual address. Whenever there is an I-TLB miss, the hardware Page Table Walker (PTW) walks the page table and fills the I-TLB with the missing information including the ipkey. As each instruction domain applies a combination of the `Flexible Filters`, we need to maintain the configured combination of each domain. To this end, for each domain, we associate a valid bit to each of the `Flexible Filters`. We store all the valid bits in a separate 64-bit register, called Instruction Protection Register (IPR).

Figure 2 demonstrates our modifications to the RISC-V Rocket core [3] to implement FlexFilt. The modified I-TLB stores the ipkey information (received from PTW) for each entry. On an I-TLB hit, the ipkey value gets transferred to the I-Cache (alongside the physical address) and subsequently on an I-Cache hit, the instruction and its associated ipkey gets stored in the Instruction Queue. Subsequently, the ipkey value gets transferred to the decode and then execute stage without any modifications. At the execute stage, FlexFilt uses the ipkey value to read the corresponding valid bits (4-bits) of the instruction domain from IPR. In the same cycle, each of the 4 `Flexible Filters` receives the 32-bit instruction in the execute stage, and performs the filtering operation based on its `Match` and `Mask` bits configuration (Figure 1). If the resulting filter signal of any of the `Flexible Filters` is high and at the same

[2]Note that with four instruction filters we can filter all the target instructions in Fidelius [65] based on their instruction type.

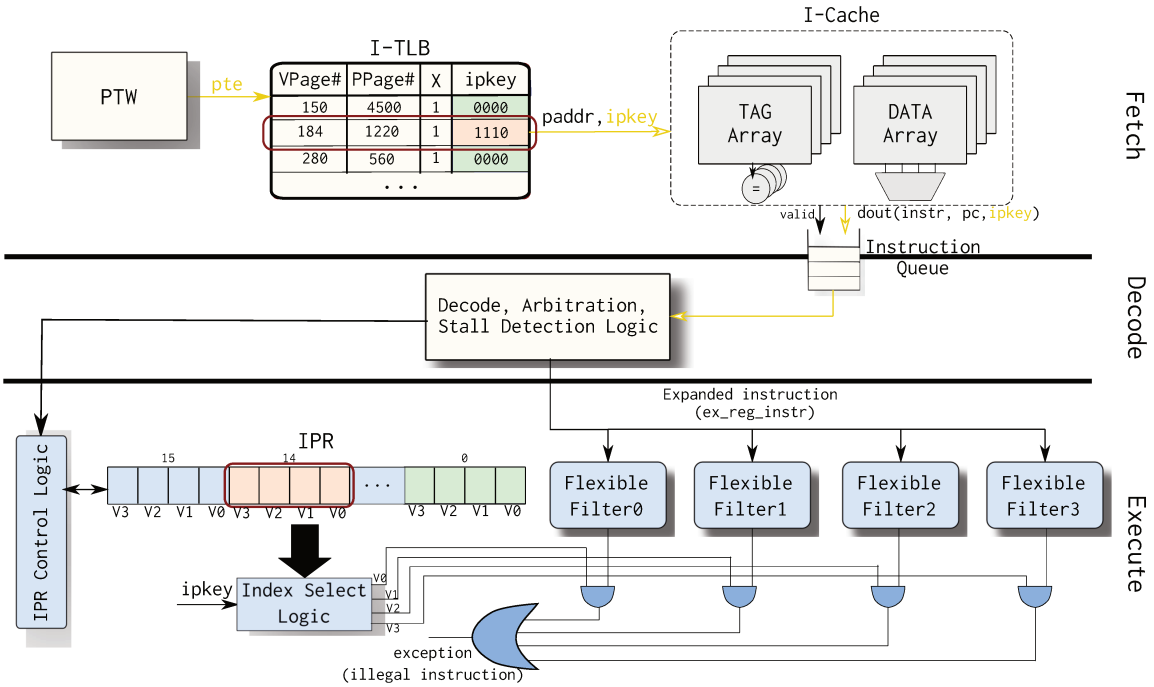

Figure 2: Simplified overview of the modifications to the RISC-V Rocket core to support FlexFilt. The blue, yellow, and gray colors show the new, minimally modified, and unmodified components, respectively.

time its corresponding valid bit is active, then FlexFilt prevents the execution of the instruction by causing an illegal instruction exception.

5.1.4 Unprivileged vs Privileged Target Instructions. While Figure 2 shows the main components of FlexFilt to filter unprivileged instructions, this implementation does not take the privilege level of the instructions into account. As discussed in Section 2, some of the previous works focused on preventing the execution of target instructions in the kernel space while others focused on preventing the execution in user space. To filter instructions in user space, we simply access the `priv` field provided in the `MStatus` CSR of the Rocket core and only apply the `Flexible Filters` on user-level instructions. To filter the target instructions at kernel level, we use a similar approach as PMPs. We add two pairs of new CSRs to store the physical address range for filtering kernel-level instructions. To specify the kernel-level target instructions, we implement four dedicated `Flexible Filters`. These dedicated filters are only applicable to kernel-level instructions executing in the physical address range specified with our newly added CSRs. The above-mentioned CSRs and dedicated filters can only be configured from the machine mode.

5.2 OS support

FlexFilt is capable of filtering target instructions in OS-managed processes as well as the kernel itself. In our design, we consider scenarios where each user-space process can filter different target instructions. To enable a per-process instruction filtering capability, we need to provide the OS support for FlexFilt. In this section, we discuss the Linux kernel modifications to support FlexFilt.

5.2.1 Instruction Protection Keys. We implement our instruction protection keys on top of the existing support for memory protection keys. The Linux kernel provides three new system calls, i.e., `pkey_mprotect`, `pkey_alloc`, and `pkey_free`, to support Intel MPK. `pkey_mprotect` is an extension to `mprotect` system call. In addition to updating the permission bits of the PTEs of specified pages, `pkey_mprotect` assigns a protection key to the PTE. `pkey_alloc` and `pkey_free` system calls enable a software developer to allocate and free a protection key, respectively. The kernel implements an allocation bitmap (16-bit for Intel MPK) to keep track of the allocated pkeys. The recent works on memory protection keys for RISC-V [14, 58] extend the Linux kernel support of memory protection keys to the RISC-V ISA. We modify the existing support for `pkey_alloc` and `pkey_mprotect` in the kernel to allocate an instruction protection key and associate the specified executable pages with an ipkey, respectively. We add a new flag to `pkey_alloc` and `pkey_mprotect` system calls to identify an instruction protection operation. Unlike the existing memory protection proposals on RISC-V, we only use 4 bits of the unused PTE bits to store our protection keys. Hence, we add an instruction allocation bitmap to keep track of 16 instruction protection domains.

5.2.2 Per Process OS Support. To enable a per process view for instruction protection domains, we maintain the domain information during context switches. We modify the `task_struct` in the Linux kernel to keep the configuration of each `Flexible Filter`, which includes the `Match` and `Mask` bits. Additionally, we maintain the `bitmap` of allocated ipkeys as well as `IPR` contents. In Section 7, we discuss the overhead of maintaining FlexFilt information during context switches.

5.2.3 Kernel-Level Instruction Filtering Support. The kernel-level `Flexible Filters` and their corresponding physical address range

Table 2: FlexFilt's Application Programming Interface (API).

Function	Invoked Custom Instruction
config_filter(uint32_t match, uint32_t mask, uint8_t priv, uint8_t index)	SETMATCH, SETMASK, and SETPRIV
config_instr_domain(uint64_t d_index, uint64_t v_index)	WRIPR

CSRs can only be configured from the machine mode. In the RISC-V environment, the Berkeley Boot Loader (BBL) [56] enables us to configure our kernel-level filters and their CSRs from the machine mode, prior to booting up the Linux kernel. As our kernel-level filters are applicable to all processes, we do not need to maintain their configuration during context switches.

5.3 Software Support

For configuring FlexFilt, we leverage the standard RISC-V `custom` instruction extension to define new instructions. Table 2 shows our software API and the unprivileged `custom` instructions that each API invokes to configure FlexFilt. We provide the `config_filter` function to configure each `Flexible Filter` by specifying its corresponding `Match`, `Mask`, and `privilege` bits. The software developer can leverage the `config_instr_domain` function to set the valid bit of a `Flexible Filter` for a specific instruction domain. We also provide five privileged `custom` instructions, which are accessible only at the supervisor level. We leverage these five instructions to maintain FlexFilt's information during context switches.

We leverage the `pkey_mprotect` system call to associate a group of executable pages, specified by `addr` and `len`, with an ipkey. Multiple functions with non-contiguous address ranges can be assigned with the same ipkey and subsequently create one instruction domain. To invoke the `pkey_mprotect` system call, we should obtain the address range of each instruction domain. In a deployed system, the software developer can annotate the source code to specify the sections of the program belonging to an instruction domain. Then, we can modify the loader to invoke `pkey_mprotect` based on the extracted information from annotations. Rather than modifying the compiler and the loader, as a proof of concept, we leveraged `LD_PRELOAD`. We leave the required modifications to the loader as part of our future work.

As an example, consider the scenario we described in Section 2. In this scenario, a software developer wants to allow the execution of `WRPKRU` instruction in two trusted functions (`good_code1` and `good_code2`) while preventing the execution of `WRPKRU` in other parts of the code. To specify the trusted instruction domain, we first allocate a new ipkey (via `pkey_alloc` system call). By assigning the same ipkey to both `good_code1` and `good_code2` functions, we associate them with the same trusted instruction domain, i.e., `domain1`. Then, we configure `Flexible Filters` to prevent the execution of the `WRPKRU` instruction in the default domain, i.e., `domain0`. To assign the allocated ipkey to the two trusted functions, we need to invoke the `pkey_mprotect` system call, which requires the address and length of the trusted functions as input arguments. To this end, as a proof of concept, we mark the two trusted functions with an attribute in the source code and use a linker script to page align these functions. To obtain the address range of these two functions and invoke `pkey_mprotect`, rather than modifying the loader, we use `LD_PRELOAD`. To simplify using `LD_PRELOAD` on `good_code1` and `good_code2` functions, we define them as extern function pointers. Subsequently, when the executable containing `good_code1` or `good_code2` is about to get executed, first the shared library (.so) that pre-loads these two functions gets loaded and the loader fills in the corresponding addresses of these functions. In our shared library, we use `dlsym` to obtain the address of `good_code1` and `good_code2` functions. Then, we use the obtained address as an argument for invoking `pkey_mprotect`. Subsequently, we call the original implementation of our trusted functions.

6 CASE STUDY

For prior works that rely on binary rewriting, filtering target instructions in dynamically generated code is more challenging than the static code. A popular use-case of dynamically generated code that adversaries are likely to take advantage of is in Just-In-Time (JIT) compilers. JIT compilers are not limited to dynamically generating user-space code; JIT compilation also occurs in the kernel, e.g., extended Berkeley Packet Filters (eBPF) VM has a JIT compiler. In this section, we provide an experimental study to demonstrate the advantages of leveraging FlexFilt for run-time instruction filtering of dynamically generated code through JIT compilation.

6.1 JIT compilation

JIT compilation dynamically compiles interpreted programming languages such as JavaScript into bytecode (an intermediate representation) or native machine code. A JavaScript engine (e.g., ChakraCore [47], SpiderMonkey [48], and V8 [27]) is responsible for compiling and executing the JavaScript code, memory management, and optimization. For illustration purposes, we use V8, which is Google's open-source JavaScript engine used in Chrome, Chromium, and Node.js. Note that other JIT compilers work in a similar manner. V8 first compiles the JavaScript code into a bytecode. Then, V8's optimizing compiler generates an optimized machine code from the bytecode.

6.2 V8 JIT Compilation Experiment

As discussed by prior works including ERIM [61] and Donky [58], isolation of the dynamically generated code through memory protection domains is of great importance. As a case study, we consider the scenario of leveraging Intel MPK for intra-process memory isolation of the Chromium browser. While browsing webpages, the V8 engine dynamically compiles the JavaScript code and translates it to optimized native machine code. To prevent reuse of JIT compiled code for unauthorized modification of a protection domain's permission bits, we need to ensure that the code does not contain any occurrences of `WRPKRU` instruction. A modified JIT compiler can prevent the emission of *explicit* unsafe instructions (e.g., V8 never emits `WRPKRU` explicitly); however, preventing misaligned/overlapping *implicit* instructions is challenging. For example, a JIT compiler might use constant data in the JavaScript code for generating an instruction, which inadvertently can lead to the creation of an implicit unsafe instruction. In the absence of ubiquitous compiler adjustments, our hardware-assisted approach can transparently prevent the execution of unsafe instructions at runtime.

An attacker can exploit an implicit occurrence of `WRPKRU` instruction using control-flow hijacking attacks and in turn gain access permission to a protection domain. To prevent such an exploitation,

Table 3: The measured size of executable bytes generated for browsing the Alexa top-10 websites, on average.

Website	Executable bytes generated when loading the frontpage	Executable bytes generated per second while browsing the page
Google.com	0	3,458
Youtube.com	266,798	2,620
Tmall.com	366,003	15,323
Baidu.com	0	1,532
Qq.com	159,565	2,043
Sohu.com	34,096	2,014
Facebook.com	20,938	9,712
Taobao.com	220,299	15,454
Amazon.com	92,442	3,098
360.cn	0	400
Geometric mean	3,432	3,258

the previous works continuously scan the newly generated code at runtime and rewrite the code if necessary. In this section, we analyze the overhead of scanning the dynamically generated code by measuring the number of generated bytes in native machine code while browsing various webpages. For this measurement, we built and ran the Chromium browser [26] on an Intel® Core™ i7-4700MQ processor @ 3.4GHz machine running Ubuntu 18.04.5 LTS. We used `v8_enable_disassembler=true` flag for building Chromium to enable disassembler support in V8. To measure the total number of generated bytes during JIT compilation, we ran Chromium with `--js-flags="--print-bytecode"` flag and browsed the Alexa top-10 websites [2].

Table 3 shows the total size of executable bytes generated by V8 engine while browsing the Alexa top-10 websites. For each website, we report two numbers: 1) the total size of executable bytes generated when loading the frontpage of the website, and 2) the number of bytes generated per second while browsing each website for 5 minutes. Note that for a website such as `Google.com` and `Baidu.com`, we do the browsing by searching various keywords without opening any of the search results. For each website, we repeated both the experiments, i.e., loading of the frontpage of the website and 5 minutes browsing, three times and reported the geometric mean. On average,[3] a binary scanning approach has to scan around 3,432 bytes and 3,258 bytes per second, respectively, for loading the Alexa top-10 pages and browsing them. In the worst case, 366,003 bytes (for `Tmall.com`) and 15,454 bytes per second (for `Taobao.com`) should be scanned. Considering 4KB pages, a binary scanning approach has to scan about 90 pages for loading `Tmall.com`, while for continuous browsing of `Taobao.com` around 4 pages should be scanned almost every second.

6.2.1 Comparison with Prior Works. Once the binary scanning finds an implicit occurrence of a target instruction, the prior works rely on a binary rewriting approach or hardware watchpoints to guarantee the safety of the target instruction execution. Binary scanning is a trivial task. The challenge is the efficient implementation of the binary rewriting. ERIM reports that the binary inspection takes between 3.5 and 6.2 microseconds per page for SPEC2006 benchmarks [34]. Unfortunately, ERIM does not report the performance overhead of their binary rewriting approach at runtime. For a continuous process such as web browsing, the binary scanning and binary rewriting should be implemented very efficiently, which is a challenging task. Typically, a binary rewriting approach incurs considerable performance overhead. For example, MULTIVERSE [7], a state-of-the-art x86 static binary rewriter reports 60.42% runtime overhead on SPECint2006 benchmarks while MAMBO [28], a dynamic binary rewriter for ARM reports an average overhead of 34% for SPEC2006 benchmarks on a Cortex-A15. Instead of dynamic binary rewriting, Hodor relies on a trusted loader to identify all the occurrences of the `WRPKRU` instruction and uses hardware watchpoints to examine the safety of these occurrences. If there are more occurrences of the `WRPKRU` instruction in a page than the number of watchpoints, Hodor relies on single-step execution, which incurs considerable performance overhead. Additionally, the current implementation of Hodor does not support JIT. Although Hodor can be extended to support JIT compiled code, it has to cause a page fault before the execution of each added page to inspect the page and configure the watchpoints. In contrast to ERIM and Hodor, FlexFilt examines each executed instruction at hardware level with negligible performance overhead (refer to Section 7.3) and prevents the execution of target instructions without the need for binary scanning, binary rewriting, or hardware watchpoints. Overall, scanning the dynamically generated code for target instructions is unnecessary if a protection mechanism can prevent the execution of target instructions reliably, which FlexFilt does through instruction protection domains and hardware-level `Flexible Filters`.

[3] As shown in Table 3, some websites had zero executable bytes generated. When loading the frontpage of these website, they did not result in any native bytes. As a workaround for calculating the geometric mean in the presence of samples with zero values, we converted each zero value to one.

7 EVALUATION

In this section, we discuss our implementation and evaluation framework, and demonstrate the feasibility of FlexFilt's design with our prototype. To this end, we have to demonstrate FlexFilt's correct functionality and enforcement of the developer's chosen security policy as well as acceptable performance, power, and area overheads. To verify the correct functionality of FlexFilt, we demonstrate FlexFilt's capability in preventing the execution of the `WRPKR` instruction in untrusted domains. To evaluate FlexFilt's performance overhead, we devise experiments to measure the overhead of configuring FlexFilt, the context switch overhead, and the overall execution time overhead. To estimate the area overhead of FlexFilt, we leverage the resource utilization of our FPGA prototype and reason about our power overhead according to our area overhead estimation. Finally, we provide a head-to-head comparison of the filtering capability and overheads of FlexFilt with PHMon, a hardware monitor.

7.1 Implementation and Evaluation Framework

We implemented FlexFilt on a RISC-V Rocket core [3] using the Chisel Hardware Description Language (HDL) [6]. The RISC-V Rocket core is a single-issue in-order processor with a 6-stage pipeline. We prototyped our hardware design on a Xilinx Zedboard FPGA [52]. We modified the Linux kernel (v4.15) to add the support for FlexFilt. We implemented our support for instruction protection domains on top of the existing implementation of memory protection keys for the RISC-V ISA using the open-source Linux kernel patches from Donky [58] and SealPK [14].

7.2 Functional Verification

To verify the correct functionality of FlexFilt, we implemented tests that created scenarios similar to the one described in Section 5.3. As an example, we leveraged the `WRPKR` extended instruction [14] to implement memory protection domains. Then, we prevented the execution of the `WRPKR` instruction except in the trusted functions specified by the user. To this end, we leveraged FlexFilt's API and the `LD_PRELOAD` approach. We cross-compiled the code using RISC-V GNU toolchain and ran the program on our FPGA prototype. As expected, FlexFilt allows the execution of `WRPKR` in trusted functions and prevents its execution anywhere else in the code by causing an illegal instruction exception.

To demonstrate FlexFilt's capability in preventing a security attack, we leveraged a buffer overflow vulnerability in a simple program to inject a `WRPKR` instruction, which modifies the permission bits of a memory protection domain in an untrusted function. As expected, FlexFilt was able to successfully prevent the execution of the injected `WRPKR` instruction in the untrusted domain at runtime.

To verify the correct functionality of the FlexFilt's kernel-level filtering capability, we configured FlexFilt in BBL [56] to limit the execution of our `custom` instructions used to maintain FlexFilt's information during context switches. As expected, FlexFilt allows the execution of these `custom` instructions in the context switch function and prevents their execution outside this function.

7.3 Performance Evaluation

In this section, we evaluate FlexFilt's performance overhead using microbenchmarks. Additionally, we report the context switch overhead to maintain FlexFilt's information and the overall execution time overhead of FlexFilt using standard benchmarks.

7.3.1 Microbenchmarks. FlexFilt provides four `Flexible Filters`. During the program execution, each `Flexible Filter` receives the current instruction at the execution stage and applies its configured filter on the instruction. As the filtering operation does not need any extra cycles, we expect FlexFilt to incur negligible performance overhead. At hardware level, all the `Flexible Filters` perform the filtering operation in parallel. Hence, regardless of the number of activated configured filters, we expect FlexFilt's performance overhead to remain the same. To examine the effect of number of activated configured `Flexible Filters` on the performance overhead, we ran the `mcf` benchmark from SPEC2000 benchmark suite [33] for active filter count ranging from 0 to 4. We repeated each experiment 3 times and considered the geometric mean of execution times as the performance metric. As expected, this experiment showed that the execution time overhead of FlexFilt did not change with the number of activated filters. We expect a similar behavior in other benchmarks too. With various number of filters, the total execution time stayed the same (the geometric mean of the execution time overhead across various configurations was 0.17% with a standard deviation of less than 1%).

We devised a microbenchmark to measure the overhead of configuring `Flexible Filters` as well as the overhead for applying a combination of filters to an instruction domain. Appendix B presents these measured overheads.

7.3.2 Context Switch Overhead. During the context switches, we maintain FlexFilt's information including the configuration of each `Flexible Filter` and the contents of `IPR`. As the amount of FlexFilt's information to maintain stays the same during context switches, we expect FlexFilt's context switch overhead to be the same across all applications. We measured the performance overhead of FlexFilt during context switches for 9 (out of 12) SPECint2000 [33] and 9 (out of 12) SPECint2006 [34] benchmarks with test inputs. We were not able to run the remaining benchmarks, i.e., eon, perlbmk, and vortex from SPECint2000 and mcf, perlbench, and sjeng from SPECint2006, on our baseline system due to runtime errors (e.g., out of memory error because our evaluation board only provides 256MB of DDR memory). In addition to the context switch overhead, we measured the total execution time overhead of each benchmark. We ran each benchmark three times and determined the geometric mean of the execution times. Table 4 shows the performance overhead of maintaining FlexFilt's information during each context switch. On average, FlexFilt increases the execution time of each context switch by 2.23% and 2.34% for SPECint2000 and SPECint2006 benchmarks, respectively. However, the time spent in context switches for most workloads is negligible (e.g., 0.50% on average for the SPECint2000 benchmarks). As expected, the context switch overhead is in a similar range for various benchmarks. As the number of context switches varies across different benchmarks, the total performance overhead of FlexFilt is different for each benchmark. However, FlexFilt's overall performance overhead is negligible (Table 4). In this experiment, we used 9 applications from SPECint2000 and 9 applications from SPECint2006 benchmark suites; however, as the amount of FlexFilt's information maintained during context switches is independent of the benchmark, we expect similar context switch overheads in any application.

7.4 FPGA Resource Utilization

In our FPGA prototype, the maximum frequency for the unmodified RISC-V Rocket core is 25MHz. The RISC-V Rocket core enhanced with FlexFilt operated with the same maximum frequency. Hence, our microarchitectural modifications did not negatively affect the critical path. Table 5 shows the FPGA resource utilization of an enhanced Rocket core with FlexFilt compared to the baseline Rocket core. Accordingly, as FlexFilt has less than 1% area overhead, we estimate the power overhead of FlexFilt to be negligible.

Table 4: Performance overhead of FlexFilt due to maintaining FlexFilt's information during context switches.

Benchmark Suite	Applications	Average Increase in Context Switch Execution Time	Overall Execution Time Overhead
SPECint2000	bzip2	2.97%	0.03%
	crafty	2.16%	0.04%
	gcc	1.70%	0.07%
	gap	2.49%	0.15%
	gzip	2.44%	0.05%
	mcf	2.26%	0.17%
	parser	2.14%	0.06%
	twolf	2.27%	0.00%
	vpr	1.88%	0.09%
	Geometric Mean	2.23%	0.09%
SPECint2006	astar	2.21%	0.09%
	bzip2	3.06%	0.04%
	gcc	3.63%	0.08%
	gobmk	2.04%	0.09%
	h264ref	0.82%	0.05%
	hmmer	3.23%	0.05%
	libquantum	2.95%	0.10%
	omnetpp	2.07%	0.01%
	xalancbmk	2.66%	0.09%
	Geometric Mean	2.34%	0.06%

Table 5: The FPGA utilization of the Rocket core enhanced with FlexFilt compared to the baseline Rocket core.

	Baseline		Rocket Core + FlexFilt	
	#Used	% Utilization	#Used	% Utilization
Total Slice Luts	32030	60.21	32584	61.25
Luts as logic	30907	58.1	31409	59.04
Luts as Memory	1123	6.45	1175	6.75
Slice Registers as Flip Flop	16506	15.51	17056	16.03

7.5 Comparison with a Hardware Monitor

In Section 2.3, we discussed prior hardware monitors and their potential capabilities in preventing the execution of target instructions at runtime. To the best of our knowledge, among the prior hardware monitors, i.e., PUMP [20], FlexCore [18], Harmoni [19], and PHMon [16], only PHMon is open-source [1]. In this section, we provide a quantitative comparison between FlexFilt and PHMon for filtering target instructions.

PHMon requires a Match Unit (MU) to prevent the execution of target instructions in each Contiguous Memory Range (CMR). As a head-to-head comparison, we consider a scenario where we need to prevent the execution of a target instructions, e.g., a `custom` instruction, in an untrusted domain spanning through four separate CMRs. To filter the target instruction using FlexFilt, we define an instruction domain by assigning the same instruction protection key to the corresponding pages of the four CMRs. Then, we configure one of our `Flexible Filters` with the target instruction and enable this filter for our defined instruction domain. To filter the target instruction using PHMon, we use 4 MUs and program each MU to monitor the execution of the target instruction in the address range of one of the CMRs. Both FlexFilt and PHMon can prevent the execution of the target instruction with negligible performance overhead. However, FlexFilt has considerably lower area overhead compared to PHMon. According to the resource utilization on our Zedboard FPGA, FlexFilt and PHMon (configured with 4 MUs) increase the number of Slice LUTs by ∼1% and 27%, respectively, over the baseline Rocket core. PHMon's area overhead increases (almost) linearly with the number of MUs. To support 16 domains of CMRs, PHMon requires 16 MUs, which is expected to result in ∼93% increase in the number of Slice LUTs over the baseline Rocket core.[4] In contrast to PHMon, FlexFilt is applicable at page granularity and FlexFilt's overheads are regardless of the number of CMRs. Each of FlexFilt's 16 instruction domains can be applied to any number of contiguous or noncontiguous pages.

8 DISCUSSION AND FUTURE WORK

We configure FlexFilt once a process gets loaded (or during `LD_PRELOAD`). To prevent further modifications to FlexFilt's configuration, the software developer can leverage one of the `Flexible Filters` to prevent the execution of the configuration `custom` instructions. This `Flexible Filter` can be sealed at hardware level from further modifications. As a result, once the process is loaded and FlexFilt is configured, any further execution of the configuring `custom` instructions causes an exception.

In previous sections, we discussed FlexFilt's capability in filtering the execution of target instructions in untrusted parts of the code. FlexFilt allows the execution of target instructions in trusted parts of the code. However, an adversary might leverage the vulnerabilities in untrusted parts of the code (e.g., buffer overflow) to launch a control-flow hijacking attack and execute the target instructions. To prevent such attacks, we can protect the entrance and exit points of trusted functions using trampolines or call gates.

In addition to preventing the execution of known unsafe instructions in untrusted code, FlexFilt enables us to prevent the execution of previously safe instructions as they turn into unsafe instructions (e.g., `CFLUSH`) due to evolving security attacks. For example, cache-based side channel attacks such as FLUSH+RELOAD and FLUSH+FLUSH frequently use the `CFLUSH` instruction as part of their attacks. FlexFilt can prevent the execution of `CFLUSH` in untrusted parts of code to defend against such side channel attacks.

Although FlexFilt requires hardware modifications to an existing processor, our FPGA resource utilization results indicate that these changes are small compared to the in-order Rocket processor (∼ 1% area overhead). In this paper, we provide a proof-of-concept implementation of our design in an open-source RISC-V environment. Implementing FlexFilt in other architectures requires hardware modifications, which could potentially be challenging (e.g., depending on the microarchitecture complexity) as these architectures are not openly available. The memory protection key support already exists in modern x86 and ARM architectures, which can be extended to implement instruction protection keys. Additionally, FlexFilt requires four flexible filters, which have to be integrated into a processor's pipeline.

In Appendix A, we discuss the filtering capabilities of our `Flexible Filters` in detail. Although our `Flexible Filters` enable a software developer to filter instructions at bit granularity, our current design cannot prevent the execution of target instructions based on the contents of a target instruction's operands or the contents of a memory address accessed by the target instruction. We can enhance the filtering capability of our design by expanding our bit-granular matching/masking mechanism to be applied to the contents of `rs1`, `rs2`, `rd`, and the corresponding memory address according to the the type of the target instruction. However, such a design will increase the width of our filters from 32 bits to 288 bits and requires FlexFilt to be applied at the write-back stage of the processor's pipeline. As part of our future work, we will investigate the area overhead/flexibility trade-off by considering design knobs such as four additional filters for contents of rs1, rs2, rd, and memory address versus one additional configurable filter applicable to either.

9 CONCLUSION

In this paper, we presented an efficient and flexible hardware-assisted feature, called FlexFilt, for runtime filtering of user-specified instructions at page granularity. Our flexible hardware-assisted feature can be used in a variety of security use cases that need to prevent the execution of certain unsafe instructions in untrusted parts of the code. We demonstrated the advantage of FlexFilt over binary scanning/binary rewriting approaches by measuring the number of JIT bytes generated while browsing various webpages with Chromium. We implemented a practical FPGA prototype of our design and provided the Linux kernel support for it.

[4] We estimate this number based on interpolation as we were able to fit PHMon with the maximum of 5 MUs into our Zedboard FPGA.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2020. PHMon. [online] https://github.com/bu-icsg/PHMon. (2020).
[2] Amazon. 2020. The top 500 sites on the web. [online] https://www.alexa.com/topsites. (2020).
[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, and others. 2016. The Rocket Chip generator. *EECS Department, UCB, Tech. Rep. UCB/EECS-2016-17* (2016).
[4] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 90–102.
[5] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *Proceedings of Network & Distributed System Security Symposium (NDSS)*, Vol. 16. 21–24.
[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of ACM Design Automation Conference (DAC)*. 1212–1221.
[7] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, and others. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *Proceedings of Network & Distributed System Security Symposium (NDSS)*.
[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged {CPU} features. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 335–348.
[9] Darrell Boggs, Gary Brown, Nathan Tuck, and KS Venkatraman. 2015. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro* 35, 2 (2015), 46–55.
[10] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. IEEE, 56–71.
[11] Marc L Corliss, E Christopher Lewis, and Amir Roth. 2003. DISE: A programmable macro engine for customizing applications. In *Proceedings of Annual International Symposium on Computer Architecture (ISCA), 2003. Proceedings.* IEEE, 362–373.
[12] Arthur Azevedo De Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-policies: Formally verified, tag-based security monitors. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 813–830.
[13] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*. IEEE, 15–24.
[14] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. 2020. Sealable Protection Keys for RISC-V. *arXiv preprint arXiv:2012.02715* (2020).
[15] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. 2021. SealPK: Sealable Protection Keys for RISC-V. In *Proceedings of Design, Automation and Test in Europe (DATE)*. 1–4.
[16] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. PHMon: A programmable hardware monitor and its security use cases. In *Proceedings of USENIX Security Symposium (Security)*. 807–824.
[17] Leila Delshadtehrani, Schuyler Eldridge, Sadullah Canakci, Manuel Egele, and Ajay Joshi. 2017. Nile: A programmable monitoring coprocessor. *IEEE Computer Architecture Letters* 17, 1 (2017), 92–95.
[18] Daniel Y Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G Edward Suh. 2010. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 137–148.
[19] Daniel Y Deng and G Edward Suh. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 1–12.
[20] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. 2015. Architectural support for software-defined metadata processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 487–502.
[21] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *Proceedings of USENIX Security Symposium (Security)*. 131–148.
[22] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. 2001. Dynamic binary translation and optimization. *IEEE Trans. Comput.* 50, 6 (2001), 529–548.
[23] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. Jitguard: hardening just-in-time compilers with SGX. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2405–2419.
[24] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-Process Memory Isolation EXtension. In *Proceedings of USENIX Security Symposium (Security)*. 83–97.
[25] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 585–598.
[26] Google. 2020. The Chromium Projects. [online] https://www.chromium.org/Home. (2020).
[27] Google. 2020. What is V8? [online] https://v8.dev/. (2020).
[28] Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. 2016. MAMBO: A low-overhead dynamic binary modification tool for ARM. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–26.
[29] Joseph L Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. 2012. A case for unlimited watchpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 159–172.
[30] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing performance and isolation in Microkernels with efficient intra-kernel isolation and communication. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 401–417.
[31] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*. 173–184.
[32] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 489–504.
[33] John L Henning. 2000. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer* 33, 7 (2000).
[34] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
[35] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1470–1486.
[36] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Towards a fully abstract compiler using Micro-Policies: Secure compilation for mutually distrustful components. *arXiv preprint arXiv:1510.00697* (2015).
[37] Henrik Karlsson. 2020. *OpenMZ: a C implementation of the MultiZone API.* Master's thesis. School of Electrical Engineering and Computer Science (EECS), KTH Royal Institute of Technology.
[38] Haeyoung Kim, Jinjae Lee, Derry Pratama, Asep Muhamad Awaludin, Howon Kim, and Donghyun Kwon. 2020. RIMI: instruction-level memory isolation for embedded systems on RISC-V. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 1–9.
[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, and others. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1–19.
[40] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 437–452.
[41] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.
[42] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 1–16.
[43] Samuel Lindemer, Gustav Midéus, and Shahid Raza. 2020. Real-time Thread Isolation and Trusted Execution on Embedded RISC-V. In *Proceedings of the International Workshop on Secure RISC-V Architecture Design Exploration (SECRISC-V)*.
[44] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 49–64.

[45] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and efficient CFI enforcement with intel processor trace. In *Proceedings of the IEEE International Symposium on High performance computer architecture (HPCA)*. 529–540.
[46] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1607–1619.
[47] Microsoft. 2020. ChakraCore. [online] https://github.com/Microsoft/ChakraCore. (2020).
[48] Mozilla. 2020. SpiderMonkey: The Mozilla JavaScript runtime. [online] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. (2020).
[49] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of USENIX Annual Technical Conference (ATC)*. 241–254.
[50] ARM. 2009. ARM security technology, building a secure system using TrustZone technology. [online] http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. (2009).
[51] ARM. 2018. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. (2018).
[52] Digilent's ZedBoard Zynq FPGA. 2020. Development board documentation. [online] http://www.digilentinc.com/Products/Detail.cfm?Prod=ZEDBOARD/. (2020).
[53] Hex-Five. 2020. MultiZone Hex Five Security. [online] https://hex-five.com/. (2020).
[54] IBM Corporation. 2017. Power ISA version 3.0b. (2017).
[55] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developers Manual. (2019).
[56] RISC-V. 2021. RISC-V Proxy Kernel and Boot Loader. [online] https://github.com/riscv/riscv-pk. (2021).
[57] Nick Roessler and André DeHon. 2018. Protecting the stack with metadata policies and tagged hardware. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 478–495.
[58] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *Proceedings of USENIX Security Symposium (Security)*. 1677–1694.
[59] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2018. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 624–637.
[60] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 395–410.
[61] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of USENIX Security Symposium (Security)*. 1221–1238.
[62] Andrew Waterman, Krste Asanovic, and SiFive Inc. 2019. *The RISC-V instruction set manual, volume i: unprivileged ISA, Document Version 20191213.* Technical Report.
[63] Andrew Waterman, Krste Asanovic, and SiFive Inc. 2019. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified.* Technical Report.
[64] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The RISC-V instruction set manual, volume i: Base user-level ISA. *UCB, Tech. Rep. UCB/EECS-2011-62* (2011).
[65] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 441–453.
[66] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 680–692.
[67] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *Proceedings of USENIX Security Symposium (Security)*. 1219–1236.
[68] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. 2004. iWatcher: efficient architectural support for software debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 224–235.

Table 6: Opcode-based grouping of RV64I instructions [62].

Instruction group	Opcode	Number of instructions
LUI	0110111	1
AUIPC	0010111	1
JAL	1101111	1
JALR	1100111	1
BRANCH	1100011	6
LOAD	0000011	7
STORE	0100011	4
ALUI	0010011	16
ALU	0110011	15
FENCE	0001111	1
ECALL/EBREAK	1110011	2

Table 7: Cycle counts for FlexFilt configuration.

Mechanism	Function or Instruction	#Cycles
API	config_filter	46
	config_instr_domain	69
Custom Instruction as Inline Assembly	`SETMATCH`	4
	`SETMASK`	5
	`SETPRIV`	4
	`WRIPR`	4

A FLEXIBLE FILTERING CAPABILITY

Our `Flexible Filters` enable a software developer to filter instructions at bit granularity. In addition to filtering a specific instruction or a subset of an instruction,[5] `Flexible Filters` can be used to filter a group of instructions. To clarify this capability, we examined the RV64I base instruction set, which consists of 55 instructions [62]. As shown in Table 6, these instructions can be divided into 11 groups, based on their `opcodes`. A software developer can configure a single `Flexible Filter` to filter any of the above-mentioned group of instructions. Two or more groups of instructions can be merged together and form a larger instruction filtering group. As an example, consider the scenario where a security developer defines secure versions of load and store instructions. Then, she specifies a trusted portion of the code, where she replaces all the load and store instructions with their secure counterparts. To ensure that the trusted code does not execute an ordinary store or load instruction at runtime, we can leverage FlexFilt. We group all the `LOAD` and `STORE` instructions together (11 instructions with the `opcode = 0-00011`) and configure one `Flexible Filter` to prevent the execution of all the instructions in this group.

B FLEXFILT'S CONFIGURATION OVERHEAD

As discussed in Section 7.3, we devised a microbenchmark to measure FlexFilt's configuration overhead. Table 7 shows the average number of cycles to configure a `Flexible Filter` and an instruction domain. As discussed in Section 5.3, we provide a software API to configure FlexFilt. The software API creates a wrapper function around the `custom` instructions to facilitate their use. As expected, using the software API is more costly compared to leveraging the `custom` instructions as inline assembly. For example, leveraging `config_filter` function to configure the `Flexible Filters` takes

[5] For example, filtering a `ret` instruction, which is defined as a `JALR` instruction with `rd = x0`, `rs1 = x1`, and `imm = 0`.

46 cycles, on average, while using its corresponding `custom` instructions (`SETMATCH`, `SETMASK`, and `SETPRIV`) as inline assembly takes less than 15 cycles.