2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00039

SpecMPK: Efficient In-Process Isolation with Speculative and Secure Permission Update Instruction

Debpratim Adak†, Huiyang Zhou†, Eric Rotenberg†, Amro Awad‡
North Carolina State University, Raleigh, USA†, University of Oxford, Oxford, United Kingdom‡
{dadak, hzhou, ericro}@ncsu.edu†, amro.awad@eng.ox.ac.uk ‡

***Abstract*—In today's digital landscape, software applications are susceptible to various threats arising from vulnerabilities in unsafe programming languages (C, C++) and speculative out-of-order cores in high-performance computers. Researchers recommended enhancements in both software and hardware for protection against such attacks.**

In-process isolation is a promising way to mitigate memory-related attacks. It compartmentalizes critical data and pointers in a separate memory region and enforces access control to this memory region. Any operation to such memory locations may require a permission adjustment before and after the operation, depending on the required access control. Memory Protection Keys, a recent architecture support, has been adopted by multiple processor vendors to allow access control changes in the user space, leading to lower performance overhead than the conventional system calls (e.g., mprotect). Still, this technology incurs significant performance overhead since the permission update instruction is serialized.

Our research demonstrates significant performance improvement by allowing speculative permission updates. However, speculative execution of the permission update instruction may upgrade access permission transiently, leading to potential speculative execution attacks. To prevent such attacks, we propose Speculative Memory Protection Keys (SpecMPK), a lightweight microarchitecture enhancement to examine permission change and block transiently upgraded memory instructions until they become non-squashable.

SpecMPK significantly improves performance compared to a serialized domain switch instruction. This work shows an average 12.21% performance improvement for selected SPEC workloads requiring frequent domain switches for various memory safety schemes using memory protection keys.

I. Introduction

With an ever-increasing connectivity of computing systems, cybersecurity has become a top priority for both users and vendors. However, a challenging trade-off between time to market of software, efficiency, and rigorous security testing is usually taken by vendors. Thus, software bugs and vulnerabilities due to improper, even unintentional, programming practices usually open a wide range of exploitation opportunities for remote attacks. A recent study by Microsoft revealed that 70% of the patches for common vulnerabilities and exposures (CVEs) are due to memory safety issues [6], [7]. Memory safety vulnerabilities refer to those that result from mishandling of memory management operations, e.g., memory allocation, deletion, and accesses. However, due to efficiency and legacy reasons, a large percentage of the software stack, starting from kernel code and up to web browsers, is written using memory-unsafe languages.

Unlike memory-safe languages, such as Java, languages such as C and C++ rely on the program itself to check the bounds of memory accesses within its address space, thus allowing programs to freely create pointers and directly access any structures decoupled from the semantics of the access itself. Thus, daily-encountered code bugs as simple as missing array bound checking could facilitate a wide range of attacks [13], [18], [25]. With the increasing complexity of applications, the code size is bound to constantly increase as well, and thus the attack surface could be uncovered by any vulnerability within the same address space is increasingly large. Accordingly, intra-process isolation is an imperative approach for mitigating such an expanding attack surface [14], [21], [26], [29], [31], [33], [51]. Following the least-privilege principle, in-process isolation restricts the memory space a particular portion of the program could access and the type of access allowed (if at all).

A conventional approach for in-process isolation would be to explicitly change page(s) access permissions upon leaving/entering a new domain (i.e., code portion) within the same address space [14], [21], [26], [29], [31], [33], [51]. Intuitively, the smaller the domain is, the smaller the attack surface it is. Such type of isolation is referred to as *domain-based*. On the contrary, in an *addressed-based* isolation, each memory access from an untrusted code is preceded by bound check instruction to ensure it does not access the isolated memory region.

Over the years, many techniques using both hardware and software have been made available for intra-process isolation in modern systems: Address Space Layout Randomization (ASLR), MPX, Memory Protection Keys (MPK), *mprotect*. We identify three properties crucial for isolation techniques: fast interleaved access [20], secure isolation, and enabling least privilege principle. These properties define the capability to access protected and unprotected pages alternately with a low-performance overhead, prevent accesses to the isolated region using untrusted access instructions both speculatively and non-speculatively, and facilitate multiple protected regions isolated from one another, respectively. However, out of all these schemes, only MPK supports all three properties. This technology was introduced by Intel, and it is currently adopted by both ARM [4] and AMD [5]. For this work, we use the

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00039

terminology used by Intel for the ease of discussion.

The MPK processor feature offers lightweight mechanisms that allow associating memory pages with a particular color (also called protection key or `pKey`). The access permissions of a color, hence the group of pages associated with it, can be changed by simply writing to a register dedicated to holding permissions of colors. Most prominently, MPK follows the aforementioned approach through a per-CPU user-accessible register called PKRU. A special, despite unprivileged, instruction (WRPKRU) can update PKRU directly without any privileged operation. PKRU holds the permissions for the maximum number of colors supported in the system (e.g., 16 colors in MPK). Each page's color is indicated as in the page table entry (PTE) and hence will be known upon address translation.

Although MPK is increasingly positioned for deployment in safeguarding against a wide range of attacks [14], [27], [29], [33], [40], [41], [51], and has better performance than conventional system calls (e.g., *mprotect*), we identify it can incur significant performance overheads compared to non-secure applications when used as frequently as sought in many use cases. Accordingly, in this paper, we investigate and demystify the root cause of the performance overhead when MPK is used for cases that require frequent domain changes. Interestingly, we find that the serializing effect of updating PKRU (i.e., WRPKRU instruction) leads to significant stalls in the pipeline, contributing to the majority of the overheads. Compared to a hypothetical implementation that allows non-serializing execution of WRPKRU instructions, the current implementation can lead to an average performance overhead of 11.17% (up to 32.63%). Unfortunately, allowing non-serializing or speculative execution of WRPKRU is challenging for the following reasons. First, the current WRPKRU instruction has an implicit destination register (PKRU), which must be read at execution time for each memory instruction, needing extra ports (hence more power and area) to the register file. Second, allowing speculative execution of PKRU updates could enable otherwise circumvented speculative execution side channel attacks. Therefore, our goal is to allow performance-efficient usage of MPK while preserving the security guarantees to prevent any leakage due to speculative execution.

To address the aforementioned challenges, we propose ***Speculative Memory Protection Keys (SpecMPK)***, a novel microarchitectural support that allows speculative execution of permission updates in MPK. SpecMPK builds upon the following observation. A WRPKRU update that disables access to a certain `pKey`(s) would not be followed by memory accesses to the affected regions. Otherwise, exceptions would be thrown. Therefore, we expect that the memory instructions after the WRPKRU *disable* update are safe to execute speculatively as long as we have an appropriate mechanism to detect any possible side channels. Since most memory instructions do not access protected regions, SpecMPK benefits from executing such instructions speculatively.

Contribution We make the following contributions to enable speculative execution of MPK.

- We propose a lightweight microarchitecture enhancement, SpecMPK, to enable speculative updates of the PKRU register.
- We identify vulnerabilities related to speculative permission updates, and SpecMPK not only detects these vulnerabilities but also prevents memory access to mitigate potential microarchitectural side channels.
- We thoroughly analyze the performance and security aspects of SpecMPK. We evaluate SpecMPK using SPEC2017 workloads with return address protection and SPEC2006 workloads for the code pointer integrity protection using software instrumentation. SpecMPK achieves 12.21% speed up over serialized WRPKRU for these workloads.

II. Background

A. MPK

Memory Protection Keys (MPK) extension allows changing access permissions for a group of pages without expensive TLB shootdown, unlike *mprotect*. By associating each page with a key (aka color) through specific bits in the page table entry (PTE), changes to permissions for the corresponding color will be reflected immediately on such associated page(s). Currently, MPK supports 16 keys, hence 4 bits need to be reserved in each PTE to indicate the key a page is associated with [3]. Such 4-bit field is denoted ***pKey*** hereinafter. Additionally, the permissions of all `pKeys` are maintained in a per-CPU user-accessible register called ***PKRU***. As we have 16 `pKeys` in total, and each `pKey` has one bit for *Access Disabled (AD)* and one for *Write Disabled (WD)*, a total of 32 bits are required as the size of PKRU. As the name indicates, AD determines if access to the page is allowed or not, while WD determines if a write is allowed or not. If access is allowed, then read access is allowed irrespective of the WD value. Figure 1 depicts an example demonstrating access permission checking in the presence of MPK.

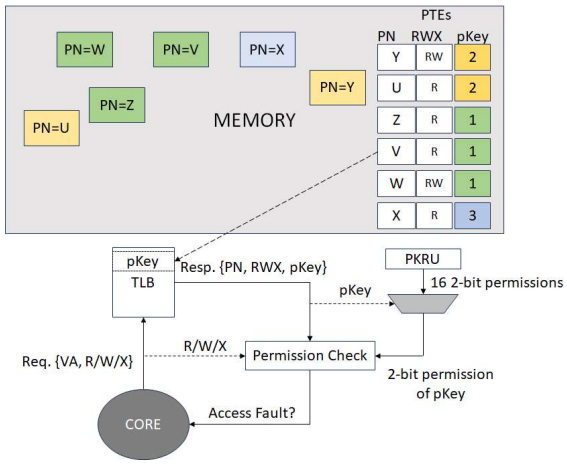

Fig. 1: Permission check for a page with MPK enabled

1) Working principle: The working principle of MPK can be broken down into three steps: key assignment, protection check, and permission update.

- **`pKey` Assignment:** To associate a page (or set of pages) with a `pKey`, Linux provides `pkey_mprotect` syscall that takes in address range and `pKey` as arguments. It updates the PTE(s) of the page(s) within the provided address range to reflect the assigned key (i.e., `pkey`) as shown in Figure 1.
- **Protection Check:** On each memory instruction access, the TLB additionally returns the `pKey` of the accessed page per the PTE entry's `pKey` field. As shown in Figure 1, the returned `pKey` value will be used as a selection input to choose the corresponding 2-bit permission bits {AD,WD} from the 16 pairs in PKRU. The normal access permissions (i.e., RWX) in PTE will be checked against the access, in addition to those indicated in the {AD,WD} pair obtained from PKRU; the most strict will be enforced, and hence allowing run-time user-controlled permission change without the security risks of direct control of page table.
- **Permission Update:** Permission updates occur through the use of the WRPKRU instruction, which involves writing to the PKRU register. Notably, WRPKRU does not *explicitly* use any source register but rather copies the contents of the *EAX* register to the *PKRU* register.

2) Implicit PKRU operand: The *PKRU* register is never explicitly used by any instruction, neither as a source nor as a destination. In the case of a RDPKRU/WRPKRU instruction, *PKRU* is used as an implicit source/destination operand, while all memory access instructions employ *PKRU* as an implicit source operand to validate the legitimacy of the access.

3) Execution of WRPKRU: The WRPKRU instruction is executed non-speculatively, and memory accesses are stalled until all prior WRPKRUs retire [40]. As a result, frequent use of WRPKRU instruction leads to significant performance degradation.

B. Memory Corruption and Overread Vulnerability

Unsafe languages such as C and C++ allow out-of-bound accesses due to insufficient bound checking at run-time, causing memory corruption, and buffer overread vulnerability. Such memory accesses potentially enable attackers to corrupt memory locations storing control-specific data; it also allows stealing confidential data.

*1) **Memory Corruption**:* in memory corruption attacks, the attacker overwrites the content of locations in the stack or heap to execute arbitrary operations. Three prominent memory corruption attacks are Jump-Oriented Programming (JOP) [13], Return-Oriented Programming (ROP) [42], and Data-Oriented Programming (DOP) [25]. These attacks primarily follow three steps as follows.

- The attacker corrupts memory locations that dictate control and data flow for the application. These include function pointers, return addresses, and data pointers.
- The attacker creates gadgets that are a sequence of machine instructions from the application's code region.
- These gadgets are stitched together to execute an operation.

It is proven that these attacks are Turing complete, meaning by stitching together multiple gadgets, the attacker can execute any algorithm. Protection against these attacks includes instrumenting the application to enable control flow integrity.

*2) **Buffer Overread**:* Buffer overread vulnerabilities have the potential to allow attackers to read data from locations that may contain sensitive or confidential information. Such vulnerabilities often occur due to inadequate bound checking in the code. An example of this is the Heartbleed attack [18], which exploits a buffer overread vulnerability to access session keys in the OpenSSL library.

To mitigate buffer overread vulnerabilities, one approach is to store confidential data in a secure memory region. Enabling access to such secure memory region is done explicitly through granting access permissions only when necessary while remaining disabled for the rest of the execution.

C. Speculative Attacks

Speculative attacks occur as a result of vulnerabilities in out-of-order processors. Out-of-order processors improve performance by executing instructions speculatively to achieve instruction- and memory-level parallelism. To do so, hardware components such as branch/memory-dependence/value predictors allow speculatively proceeding when otherwise the pipeline would have been stalled. While the execution of instructions could proceed speculatively, committing changes (i.e., instructions retiring) occurs in order and only in the absence of any exceptions and/or the speculation has been resolved successfully. Reorder Buffer (ROB) is responsible for retiring instructions in the correct order. Misprediction by any of the predictors causes dependent/following instructions to be squashed. Although squashed (also called transient) instructions are prevented from committing changes, they could leave microarchitectural effects that enable attackers to infer the value of a speculatively accessed location, for instance. Most commonly, cache side channel attacks allow an adversary to infer the value of a speculatively accessed location by leaving a microarchitectural effect that depends on such value; for instance, speculatively using the *secret* value as an address of a subsequent load/store and hence leave an effect on a particular cache set of which knowing the set itself could reveal the secret value – the secret was used as address and hence part of the indexing function to decide the set number. Different types of instructions that potentially consume a speculative value could leave a value-dependent microarchitectural effect; this effect lasts beyond the speculation resolution and is hence referred to as *transmitting instructions* [63]. Figure 2 demonstrates how speculative execution works and the terminology.

Two recent attacks are prominent examples of the extent such speculative execution vulnerabilities can compromise the system: Spectre [30] and Meltdown [35]. The Spectre vulnerability exploits speculative execution of the *transmitting*

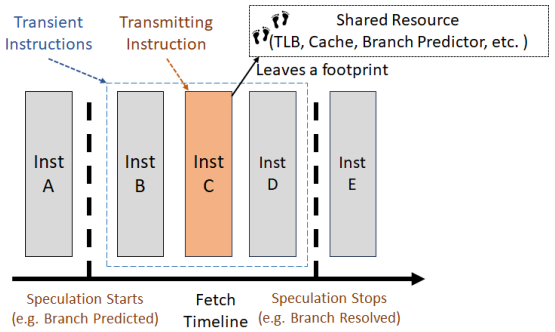

Fig. 2: Side channel due to speculative updating of the microarchitectural state.

instruction due to branch misprediction. On the other hand, meltdown vulnerability arises due to rogue data cache load, which exploits the delayed handling of memory protection faults by the CPU. Researchers have explored various Specter and Meltdown variants in today's CPUs from different manufacturers [16], [28], [30], [32], [33], [36], [45], [52], [57].

Since the Spectre attack is a result of branch misprediction, NDA [56] had identified this attack as *control-steering* attack. Conversely, attacks similar to meltdown, allowing speculative execution of instructions past a faulty instruction triggered by faulting load [35], transaction abort [50], interrupt delivery, exception [59], memory dependence violation [28], [50], [59] etc, is termed as *chosen-code* attack. We use similar terminology to discuss this work.

III. Motivation

Isolating sensitive data within different stack and heap regions of a process's memory is a common technique to prevent memory corruption and buffer overread vulnerabilities [14], [26], [27], [29], [33], [40], [41], [51], [61]. Compilers or application developers may distinguish sensitive data elements through either static analysis (e.g., code pointers) or prior knowledge (e.g., session key in the OpenSSL library).

Ideally, in-process isolation needs to provide memory safety protection at low-performance overhead. We motivate SpecMPK based on:

1) the advantage of MPK among the existing in-process isolation techniques and
2) our performance and security analysis of MPK.

A. *Existing In-process Isolation Techniques*

In this section, we discuss various in-process isolation techniques, both *domain-based* and *address-based*, that either exist in current systems or have recently been proposed by the academic community. MPK, *mprotect*, IMIX [20], SEIMI [54] are examples of the *domain-based* in-process isolation. On the other hand, MPX supports *address-based* isolation. These isolation techniques require dedicated hardware support. On the contrary, ASLR and Software-based Fault Isolation (SFI) [38], [46], [53] are examples of software-based isolation techniques. We have summarized various properties of the existing in-process isolation schemes in Table I.

TABLE I: Properties of Various Isolation Techniques

Isolation Method	Fast Interleaved Access	Secure	Least-Privilege Capability
MPK	✓	✓	✓
Mprotect	X	✓	✓
MPX	✓	X	✓
ASLR	✓	X	✓
IMIX [20]	✓	✓	X
SEIMI [54]	✓	✓	X
SFI [46]	✓	X	✓

Isolation through ***mprotect*** makes use of the protection bit in the page table entry to prevent unauthorized access. Accessing these protected pages requires modifying permissions in the page table when switching domains, resulting in an incoherent TLB state with the page table, requiring TLB shootdowns. While it offers secure isolation, it imposes a substantial performance overhead. Furthermore, updating protection for sparsely distributed pages requires multiple *mprotect* syscalls.

Conversely, **MPK**-based isolation mitigates the performance impact by enabling permission updates in user space. By never writing to the page table, it eliminates the need for TLB shootdowns. Additionally, a single WRPKRU instruction can update permissions for all protection keys and all pages associated with each updated `pKey`, making it efficient for sparsely distributed pages. With the use of multiple `pKeys`, MPK permits isolation with the least-privilege principle. However, MPK's performance overhead is majorly influenced by the domain switching frequency, as the WRPKRU instruction is a serializing instruction.

IMIX [20] marks a group of pages as protected in the page table and proposes a special instruction *smov* to access those pages. Access to the protected pages with any other instructions would raise an exception. This method's performance overhead is minimal as there is no requirement for permission change in the page table for accessing the isolated pages. However, since this method fails to distinguish among isolated pages, it is incapable of least-privilege isolation.

MPX introduces special bound check instructions to prevent buffer overflow vulnerability. Memory access to an unprotected region must be preceded by the bound instructions to prevent unauthorized access to the protected region. However, the bound-check can be bypassed speculatively [16], [37], making this isolation scheme non-secure. Another drawback of this technique is that code sections that can not be instrumented, such as third-party library code, can access the protected region [20].

ASLR is a software-assisted isolation technique that enables memory safety by hiding the memory layout through randomization. Nonetheless, multiple prior works were successful in revealing the layout through side channel [15], [19], [22], [24], [65]. Data-layout and finer-grained code randomization improve the resiliency of the ASLR. However, speculative probing [22] successfully uses speculative execution vulnerability to find locations of the function and data of interest for the attacker. In addition, **SEIMI** [54], a recently proposed software-assisted isolation, utilizes Supervisor Mode Access Prevention for the purpose of in-process isolation. However, it relies on hardware virtualization support. **SFI** [38], [46],

[53] masks the addresses of the memory access instruction to ensure access happens only to designated memory segments. However, due to masking, it fails to detect invalid memory accesses [20], [31]. Similar to MPX, SFI also fails to isolate protected regions within an un-instrumented code such as third-party libraries.

Use-cases of MPK: As MPK is able to provide *secure and least privilege capable in-process isolation* while incurring much lower performance overhead compared to *mprotect*, it is gaining popularity and the research community recently proposed multiple *MPK-assisted* software protections for isolating safe memory regions. These applications include isolating the safe region for the code pointer integrity [33], shadow stack to protect return address [14], heaps of the libraries written in memory-unsafe language within a memory safe software [29], [41], untrusted libraries [21] used from unknown sources, confidential data [27] to prevent data leakage due to speculative execution attacks, as well as the session keys in OpenSSL software [40], [51].

Note that although in-process isolation is a popular way to achieve memory safety, there are other alternatives, including the recently proposed hardware schemes [1], [47], [49], [55], [67], [68] for bound-check or pointer integrity. Compared to them, MPK offers high flexibility to support various use cases as mentioned above.

B. *Performance analysis of MPK*

Although MPK provides significant speed-up over *mprotect*, it experiences substantial slowdown compared to non-secure applications, specifically for the use cases with frequent domain switching. For example, safeguarding code pointers incurs a 12.4% overhead [20], while shadow stack protection introduces a 61.3% overhead on average [14]. PKRU-Safe [29] reports an average slowdown of 11.55% for a certain class of workloads and this slowdown is primarily attributed to the serialization of the WRPKRU instructions.

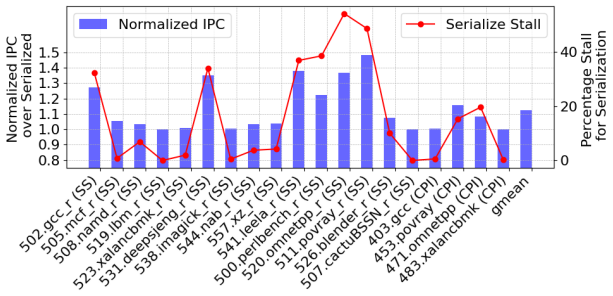

Fig. 3: Speedups due to speculative execution of WRPKRU instructions and the percent stall cycles at rename stage due to WRPKRU serialization. Workloads labeled as SS and CPI indicate shadow stack and code pointer integrity protection, respectively.

Additionally, having a limited number of `pKeys` also introduces performance overhead for the applications needing more than 16 `pKeys`. Safeguarding confidential session keys through isolation in the OpenSSL crypto library generally needs more than 16 `pKeys`, requiring frequent unmapping and remapping of the `pKeys`, resulting in a performance overhead of 4.2% [51]. Consequently, there have been multiple proposals to address this limitation [17], [40], [44], [64]. In contrast, researchers have yet to propose techniques to reduce the performance impact of the serializing WRPKRU instructions.

In this paper, we study the security and performance aspects of speculatively executing WRPKRU instruction. To our knowledge, this is the first work to investigate and propose speculative execution of the WRPKRU instruction. To assess WRPKRU serialization overhead, we compiled SPEC2017 and SPEC2006 workloads with shadow stack [14] and CPI [33], [51] and evaluated the performance overhead of serialization using our simulation infrastructure with gem5 (see VI-A). Figure 3 shows the speed up upon allowing WRPKRU instructions to execute speculatively. Additionally, it shows that a significant stall is introduced in the rename stage as a result of serialization, which in turn leads to the pipeline frontend stalls.

As shown in Figure 3, up to 48.43% (12.58% on average) performance improvement can be achieved by relaxing WRPKRU's serialization requirement. Consequently, our goal is to enable out-of-order execution around WRPKRU instructions, however, without compromising security.

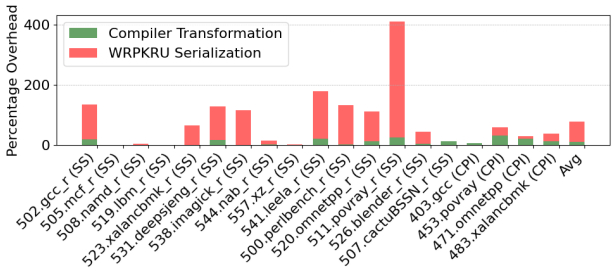

Fig. 4: Performance overhead breakdown.

Furthermore, we executed these applications on an Intel Cascade Lake processor. Figure 4 shows the overhead breakdown from compiler transformation and WRPKRU serialization, isolating compiler transformation by replacing WRPKRU with NOP. WRPKRU serialization adds a substantial 69.76% overhead on average, compared to 10.28% from compiler transformation. The difference in performance overhead between the native hardware and gem5 can be attributed to gem5's inaccuracies [9] and using simpoints that represent only the first 100 billion instructions.

C. *Security Requirements for MPK*

To better understand the challenges of optimizing MPK implementation, let's first discuss the security aspects that must be considered, specifically when WRPKRU is executed speculatively.

```
1 if (condition) {       --- mispeculated
2     WRPKRU (enable access for array1)
```

```
3     Y = array2[array1[X]*4096]
4     WRPKRU (disable access for array1)
5 }
```

Listing 1: Example of a vulnerability when WRPKRU is executed speculatively

Speculative WRPKRU Vulnerability: Both *control-steering* and *chosen-code* attacks can potentially lead to a side channel when the WRPKRU instruction is executed speculatively. Side channel induced by the *control-steering* attack leverages branch misprediction in a manner similar to the Spectre vulnerability. As demonstrated in Listing 1, the enabling and disabling of access for the array $array1$ occurs within the if block. When the branch is mistakenly predicted as taken, the WRPKRU instruction in line 2 speculatively grants access to array1, thereby enabling the transient execution of line 3, which subsequently establishes a side channel. Similarly, Spectre-BTI [30] can perform a similar attack by training the indirect target buffer to jump to a code section containing a WRPKRU instruction with access-enable permission for the `pKeys` that would otherwise have access-disable permission in the correct control path. In the event of a *chosen-code* attack, a transient WRPKRU instruction past a faulty instruction causing a squash leads to a side channel when it relaxes access permission.

Speculative Permission Upgrade for Write-Disable Pages: Preventing memory corruption involves restricting write access to the secure memory region in untrusted domains. Elevating permissions speculatively from write disable to write enable is not susceptible to memory corruption, as store updates due to mispeculation are eventually squashed and never retire. Additionally, this speculative elevation does not result in a side channel, as the store is committed to memory only after retirement. Despite this, due to speculative store-to-load forwarding optimization, a store may forward a corrupted value to a load while both instructions are yet to retire, causing a *speculative buffer overflow* attack. This scenario could potentially allow an attacker to enable control-flow hijacking speculatively. It has been demonstrated that speculative memory corruption may enable attackers to bypass Spectre defenses that use either lfence or WRPKRU to protect the Spectre gadget [28].

D. *Putting It All Together*

Due to the distinct advantage of MPK-assisted in-process isolation, the software research community recently proposed myriads of protection schemes using this method. However, to reduce the performance gap between the secure and insecure applications, MPK's requirement of WRPKRU serialization must be relaxed. To avoid side channels as a result of the speculative WRPKRU instruction, such a microarchitecture must ensure speculative execution of only the safe memory access instructions (i.e., not impacted by speculative permission upgrade) and non-speculative execution of the unsafe access instructions. As we target special classes of workload, our design must ensure the area overhead is insignificant for a general-purpose design. In contrast, the recent developments in the microarchitecture, aiming at preventing speculative execution attacks [10], [34], [43], [59], [62], [63], suffer from significant performance and area overhead. On the other hand, the early resolution of the speculativeness of the WRPKRU is complex as it requires delaying interrupt handling as well as adding new ports to the highly ported reorder buffer to update the pointer to the latest instruction with the potential of a pipeline squash [59]. In this work, by identifying potential unsafe load instructions by aggregating all inflight permission updates with minimal hardware overhead and executing them at retirement, we achieve significant performance improvement for MPK, near-identical performance compared to a microarchitecture that allows speculative execution of WRPKRU instruction without the protection against side channel vulnerabilities related to speculative permission upgrade.

IV. Threat Model

In this study, we assume that the adversary possesses the capability to exploit a memory corruption vulnerability to divert control and data flow. Furthermore, the adversary can leverage buffer overread and speculative attacks to steal confidential data. The speculative attack is potentially capable of utilizing contentions within shared resources, such as the cache and TLB.

Our assumption is that the software incorporates MPK as a preventive measure against such attacks. We also presume that the application developer appropriately employs safe memory regions to store data and associated metadata, thereby protecting against these types of attacks. Following the principle of least privilege and in accordance with the required access control, the software correctly defines permissions within trusted and untrusted domains. We assume that this architecture executes WRPKRU speculatively. Therefore, we must address the vulnerability associated with the speculative execution of the WRPKRU instruction.

V. SpecMPK

Our design of SpecMPK is built upon a baseline superscalar microarchitecture that follows the MIPS R10K style [60]. It has a Physical Register File (PRF) containing both committed and speculative register state, a Free List (FL) indicating which physical registers are free, a Rename Map Table (RMT) facilitating renaming logical source registers to physical source registers, and an Active List (AL) containing pertinent information for all in-flight instructions between the rename and retire stages. Unlike the MIPS R10K, the AL has current mappings instead of previous mappings of logical destination registers, for updating an Architectural Map Table (AMT); the effect is the same in that the AL and AMT manage the committing and freeing of physical registers, and facilitate misprediction and exception recovery.

A. *SpecMPK Design Overview*

Our proposed SpecMPK design builds on the following observations. Firstly, we note that after MPK permission updates, most memory instructions involve different `pKeys` and they are unaffected by the update. For example, when

MPK is used for stack protection, permission updates affect the shadow stack holding return addresses, while subsequent accesses typically target other regions. Secondly, memory instruction execution is safe if both the committed PKRU and all the outstanding PKRU updates have enabling permission for the associated `pKey`.

SpecMPK enables speculative execution of the WRPKRU instruction with the following design principles:

1) WRPKRU instructions execute in-order in relation to each other.
2) Memory access instructions must be executed after all the prior WRPKRU instructions are executed.
3) A load instruction is stalled if at least one of the WRPKRU instructions within the *WRPKRU-window*, which means the window between the committed WRPKRU instruction and the load we currently execute, sets the permission to Access-Disable for its `pKey`. Figure 5 demonstrates an example of the *WRPKRU-window*.
4) A store instruction executes speculatively even if *WRPKRU-window* has Write-Disable updates for its `pKey`; however, such a store instruction is prohibited from store-to-load forwarding.

We accomplish the first two objectives by renaming the PKRU register. However, the latter two objectives require inspecting PKRU updates within the *WPKRU-window*, which is accomplished using the committed PKRU register along with two *Disabling Counters* that count the numbers of speculative WRPKRU instructions with Access-Disable and Write-Disable updates.

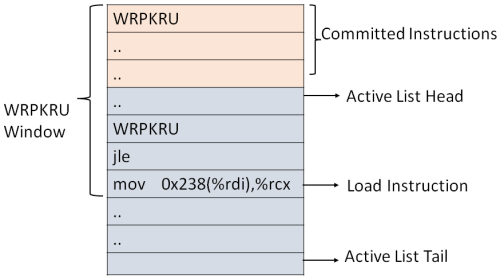

Fig. 5: Example of a *WRPKRU-window* depicting a dynamic instruction window containing a series of committed instructions followed by speculative instructions within the Active List.

B. SpecMPK Design Details

1) ***PKRU Rename****:* The PKRU register needs to be used as a source register for memory instructions and the RDPKRU instruction. Memory instructions require the latest PKRU update to validate the most recent permissions associated with each `pKey`. The WRPKRU instruction employs the PKRU as the destination register. RDPKRU instruction enables the reading of the PKRU register by copying the PKRU register to the EAX register. All of these instructions incorporate the PKRU register as implicit operands.

To rename the PKRU register, SpecMPK uses a reorder buffer that holds the speculative PKRU values and an architecture PKRU register storing the committed value. At the time of retirement of a WRPKRU instruction, the oldest value in the reorder buffer is copied to the architecture PKRU register. We manage dedicated register files for PKRU for the following reasons (1) During execution, memory access instructions need to read the committed PKRU (see Section V-C) to identify unsafe loads and stores. Renaming similar to MIPS R10k requires access instructions to store the committed PKRU mapping in its payload. However, in case of a new PKRU commit before the issue of an access instruction would free the previously committed register. In this case, instruction at the time of register-read will read from a freed PKRU physical register, resulting in a security vulnerability. But, with the use of dedicated register files, architecture PKRU stores the committed value, which the access instructions can read at the time of execution. (2) Additionally, with the addition of a dedicated PRF, we avoid additional ports in the highly ported PRF, avoiding area/power cost for the general-purpose design.

The subsequent list enumerates all the new components in the microarchitecture that facilitate the renaming of the PKRU register.

- $\mathrm{ROB}_{\mathrm{pkru}}$**:** A reorder buffer for the PKRU register that holds all in-flight PKRU updates (PKRU values) in order.
- $\mathrm{ROBHead}_{\mathrm{pkru}}$**:** The head pointer for the $\mathrm{ROB}_{\mathrm{pkru}}$.
- $\mathrm{ROBTail}_{\mathrm{pkru}}$**:** The tail pointer for the $\mathrm{ROB}_{\mathrm{pkru}}$.
- $\mathrm{ARF}_{\mathrm{pkru}}$**:** The architectural PKRU register that holds the committed PKRU value.
- $\mathrm{RMT}_{\mathrm{pkru}}$**:** Contains a valid bit and a $\mathrm{ROB}_{\mathrm{pkru}}$ tag to enable renaming.

2) ***Handling of the WRPKRU instruction****:* The WRPKRU instruction requires specific modifications in both the rename and retire stages, as it utilizes PKRU as the destination register. When a new WRPKRU instruction is encountered, the rename stage performs the following actions: it renames the PKRU to $\mathrm{ROBTail}_{\mathrm{pkru}}$ if an available entry exists in the $\mathrm{ROB}_{\mathrm{pkru}}$, sets the valid bit and updates the tag to $\mathrm{ROBTail}_{\mathrm{pkru}}$ in $\mathrm{RMT}_{\mathrm{pkru}}$, and increments $\mathrm{ROBTail}_{\mathrm{pkru}}$ to point to the next entry in $\mathrm{ROB}_{\mathrm{pkru}}$.

In the retire stage, the PKRU value from the entry pointed to by $\mathrm{ROBHead}_{\mathrm{pkru}}$ is copied to $\mathrm{ARF}_{\mathrm{pkru}}$. If the head pointer matches the tag in $\mathrm{RMT}_{\mathrm{pkru}}$, the valid bit is reset. The sequence of operations for rename and retire for a WRPKRU instruction is illustrated in Figure 6.

Although WRPKRU doesn't need to use PKRU as a source register operand, the rename stage introduces the PKRU register as a source operand to this instruction to prevent out-of-order execution of WRPKRU with respect to each other. This measure is crucial for preventing speculative permission augmentation, as elaborated in Section V-C. The renaming of PKRU as a source operand is explained in the next subsection.

3) ***Handling of the RDPKRU and memory instructions****:* Both RDPKRU and memory instructions require PKRU as a source register operand. The renaming of the PKRU source operand is contingent upon the valid bit in $\mathrm{RMT}_{\mathrm{pkru}}$. When the valid bit is set, the PKRU operand is renamed to the most recent entry in $\mathrm{ROB}_{\mathrm{pkru}}$, with the $\mathrm{RMT}_{\mathrm{pkru}}$ storing the tag,

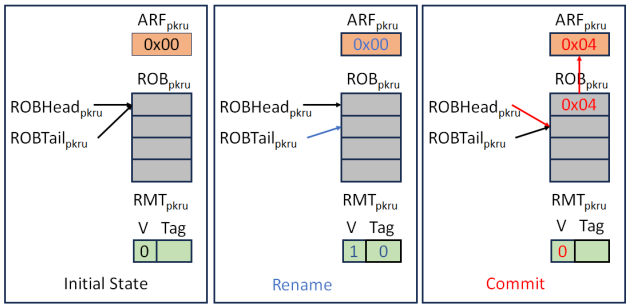

Fig. 6: The illustration of PRKU renaming. The leftmost figure portrays the initial state, the middle figure showcases the alterations resulting from renaming a WRPKRU instruction, and the last figure represents the state after committing a WRPKRU instruction.

which serves as the index to the most recent entry. A valid bit of zero indicates the absence of in-flight WRPKU instruction prior to the current instruction. In this scenario, PKRU is renamed to $\mathrm{ARF}_{\mathrm{pkru}}$ to signify that the most recent PKRU is the committed one.

Furthermore, memory instructions also utilize $\mathrm{ARF}_{\mathrm{pkru}}$, *AccessDisableCounter*, and *WriteDisableCounter* as source operands to mitigate potential side channels, as described in Section V-C. Table II provides an overview of all the new source operands introduced in SpecMPK.

TABLE II: Additional Source Operands in SpecMPK

Instruction Type	New Source Operands
Load	$\mathrm{ROB}_{\mathrm{pkru}}$, $\mathrm{ARF}_{\mathrm{pkru}}$, *AccessDisableCounter*
Store	$\mathrm{ROB}_{\mathrm{pkru}}$, $\mathrm{ARF}_{\mathrm{pkru}}$, *AccessDisableCounter* *WriteDisableCounter*
WRPKRU	$\mathrm{ROB}_{\mathrm{pkru}}$

Note that, neither memory instructions nor WRPKRU instructions read the PKRU data from the $\mathrm{ROB}_{\mathrm{pkru}}$. The primary objective of employing the $\mathrm{ROB}_{\mathrm{pkru}}$ *dependence* is to guarantee the serialized issue of memory instructions and WRPKRU instructions in relation to preceding WRPKRU instructions.

C. Speculative Attack Prevention

Preventing speculative permission upgrades through WRPKRU necessitates knowledge of PKRU update within the *WRPKRU-window*. In Figure 7, we illustrate three possible scenarios of speculative permission updates using WRPKRU regarding a single `pKey`. These scenarios are as follows.

- The latest PKRU update disables the access permission.
- Committed PKRU specifies Access-Disable while the recent PKRU reflects Access-Enable permission.
- Both the committed and the most recent PKRU contains Access-Enable permission. However, one of the older in-flight WRPKRU disables the access.

In all three scenarios, memory accesses mapped to this specific `pKey` must be stalled to prevent potential speculative attacks. To identify speculative permission upgrade, we utilize a pair of *Disabling Counters* for each `pKey`; *AccessDisableCounter*, and *WriteDisableCounter*. These counters monitor the total number of WRPKRU instructions in the speculative instruction window with Access-Disable and Write-Disable permission, respectively.

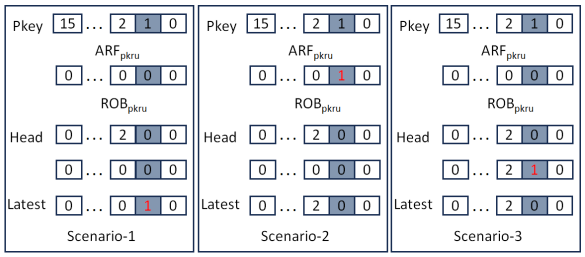

Fig. 7: Possible scenarios where load instruction could potentially create a side channel. These load instructions access pages which are mapped to `pKey` 1.

*1) **Disabling Counters**:* The process of *Disabling Counters* involves two counters that count the number of instructions within the speculative window with either Access-Disable or Write-Disable permission for each `pKey` (shown in Figure8). These counters are incremented in the execution stage through WRPKRU instructions. Following the execution of WRPKRU instructions in this stage, the counters are updated as the PKRU value becomes available. Specifically, when the Access-Disable bit is set for a `pKey`, the *AccessDisableCounter* is incremented for the corresponding `pKey`. Furthermore, if the Write-Disable bit is set, the *WriteDisableCounter* is incremented. Importantly, these counters are never incremented out-of-order, as the microarchitecture enforces this by creating dependencies among WRPKRU instructions, using PKRU as a source operand.

At the time of retirement or squash, the same WRPKRU instruction that increments any of the counters also decrements it. For this purpose, `pKey` bitmaps for the *Disabling Counters* are stored in $\mathrm{ROB}_{\mathrm{pkru}}$. The size of the counter for each `pKey` depends on the size of the $\mathrm{ROB}_{\mathrm{pkru}}$. The bit-width required for each `pKey` for each counter must be $\lfloor log2(ROB_{pkru}\ size) \rfloor + 1$ to prevent any stalls resulting from the counters.

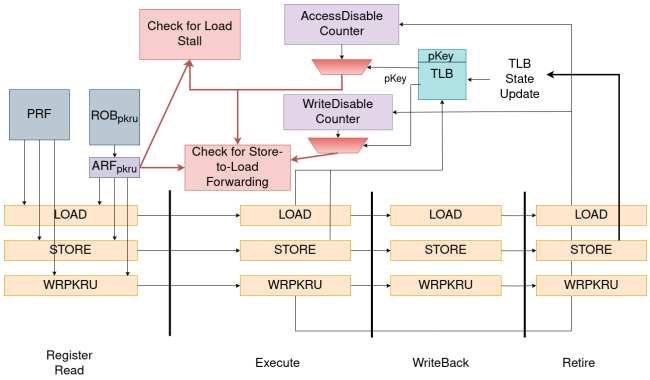

Fig. 8: Additional microarchitectural components of SpecMPK and their interaction with backend pipeline stages.

*2) **Stall conditions for memory instructions**:* In the presence of speculative permission upgrades, memory instructions face stalls. To assess the permission for these memory instructions, the retrieval of the pKey from the TLB is necessary. Therefore, these instructions need to be issued before stalling them selectively.

The load execution stage, with the obtained pKey, inspects the *AccessDisableCounter* and $\mathrm{ARF}_{\mathrm{pkru}}$ to evaluate the stall condition. The counter specifies any WRPKRU within the window with Access-Disable permission, while the ARFpkru provides the committed permission. Scenario-2, depicted in Figure 7, requires knowledge of the committed permission. Loads are stalled if either the *AccessDisableCounter* is greater than zero or the committed permission is Access-Disable. Stalled loads are replayed at the time of retirement.

On the contrary, store instructions examine both the counters and the $\mathrm{ARF}_{\mathrm{pkru}}$. Unlike load instructions, store instructions are not stalled since these instructions commit to memory after retirement, but rather prevented from store-to-load forwarding. This approach also facilitates address generation, enabling younger load instructions to learn the physical address of older store instructions and thereby reducing squash resulting from memory dependence speculation. The condition for disabling store-to-load forwarding is met if any of the *Disabling Counters* is greater than zero or if the committed PKRU indicates either Write-Disable or Access-Disable permission. If a younger load instruction matches the address with such a store instruction, it only executes after reaching the head of the Active List.

We denote the checks for stalling the load and disabling store-to-load forwarding as *PKRU Load Check* and *PKRU Store Check*, respectively. None of the memory instructions can raise an exception when issued for the first time, as these instructions do not read the renamed PKRU due to the potential staleness of the renamed register (discussed in Section V-C6).

*3) **WAR hazard for the Disabling Counters**:* This microarchitecture does not employ renaming for the *Disabling Counters*, potentially leading to a write-after-read hazard in specific scenarios involving the counters and consumer instructions, which are loads and stores. In this situation, an older memory instruction might wait in the issue queue if any of its source operands are not yet available. Meanwhile, a younger WRPKRU instruction updates the counters. If *PKRU Load Check* returns success for the load instruction before a younger WRPKRU increments the counters for the associated pKey, the instruction must needlessly wait until it becomes non-squashable. Such a stall could incur performance overhead. In the case of a store, store-to-load forwarding would be disabled conservatively. We adopt this conservative approach because it cautiously stalls only the memory instructions that access the safe region while allowing access to regular memory regions without any stall.

*4) **Protection Fault Exception**:* A load instruction must raise a protection fault if the most recent PKRU update disables access to the associated pKey. For store instructions, triggering a protection fault occurs when either Access-Disable or Write-Disable permission is applied. If *PKRU Load Check* and *PKRU Store Check* do not fail for load and store instructions respectively, such memory accesses should not violate a protection fault since the *Disabling Counters* takes into account the most recent updates.

Conversely, stalled loads are re-issued upon reaching the head of the Active List, with the load execution lane reading the $\mathrm{ARF}_{\mathrm{pkru}}$, which is also the most recent PKRU. Such a load instruction raise a protection fault if the $\mathrm{ARF}_{\mathrm{pkru}}$ specifies Access-Disable permission for the associated pKey.

However, store instructions that fail the *PKRU Store Check* are not re-issued; instead, such store instructions explicitly verify permission by accessing the TLB to retrieve the pKey and associated access rights from the $\mathrm{ARF}_{\mathrm{pkru}}$ register after reaching the Active List head.

*5) **TLB state update**:* Side channel due to microarchitectural state changes in TLB is similar to the cache side channels. Gras et al. [23] demonstrates the potential for stealing cryptographic keys through a side channel, created by the TLB. To mitigate such attacks arising from mispeculation, we adopt a strategy analogous to the one outlined in Section V-C2. The TLB state for a load instruction undergoes an update only if *PKRU Load Check* succeeds. Similarly, for the store instructions, the TLB state is updated only if the *PKRU Store Check* holds true. TLB state is updated when stalled load instructions are re-issued upon reaching the Active List head. Conversely, stores with disabled store-to-load forwarding update the TLB state during the re-evaluation of protection.

The permission of pKey in $\mathrm{ARF}_{\mathrm{pkru}}$ and in the *Disabling Counters* determines whether to stall memory instructions that cause a TLB miss, preventing any updates to the TLB state. Since the pKey of a page not residing in the TLB is unknown beforehand, we conservatively stall memory accesses causing a TLB miss. Once these instructions reach the active-list head, they are re-executed, and the TLB miss is addressed at that point.

*6) **Revisiting RDPKRU Instruction**:* The RDPKRU instruction may be susceptible to the potential staleness of the PKRU tag when the $\mathrm{ROB}_{\mathrm{pkru}}$ located at the $\mathrm{ROBHead}_{\mathrm{pkru}}$ commits to the $\mathrm{ARF}_{\mathrm{pkru}}$. To address this issue, we propose serializing the RDPKRU instruction. With serialization, PKRU register is renamed to $\mathrm{ARF}_{\mathrm{pkru}}$ for this particular instruction.

RDPKRU serves the purpose of selectively changing permissions for a set of pKeys. For example, the Pkey_set function in the GNU library takes a pKey and the associated new permission as arguments and updates PKRU accordingly. The function initially reads the old permissions from the PKRU and then modifies the permission corresponding to the pKey specified in the argument. However, this reliance on RDPKRU could be mitigated by the compiler using a data structure to store permissions, thereby reducing the need for frequent RDPKRU instructions.

VI. Evaluation Methodology

A. *Experimental Setup*

We use gem5 [12] O3 CPU model and system emulation mode to evaluate SpecMPK. We configure the gem5's CPU model based on recent high-performance processors as shown in the table III.

TABLE III: Simulation configurations

CPU	
ISA	x86-64
Issue/decode/Commit width	8 instructions
AL/LQ/SQ/IQ/PRF Size	352/128/72/160/280
$\mathrm{ROB}_{\mathrm{pkru}}$ size	8
Branch Predictor	
BTB	4096 entries
RAS	32 entries
Direction Predictor	LTAGE
Memory	
L1 Inst Cache	32kB, 8-way, 5-cycle roundtrip latency
L1 Data Cache	48kB, 12-way, 5-cycle roundtrip latency
L2 Cache	512kB, 8-way, 15-cycle roundtrip latency
L3 Cache	2MB, 16-way, 40-cycle roundtrip latency
DRAM Device	DDR4_2400_16x4

B. *Workloads*

We perform two case studies using SPEC2017 and SPEC2006 workloads involving two different memory protection schemes: shadow stack (SS) and code pointer integrity (CPI). The SS scheme provides protection against control flow hijacking, a well-established attack that utilizes return-oriented programming to divert from the correct execution path. This protection scheme copies return addresses in the shadow stack. The CPI method identifies all code pointers and stores them in a safe region [33]. The shadow stack in the first scheme and the safe region in the second one are protected using MPK to disable malicious updates. We used an open-source compiler developed by [14] and [51] for the compilation of shadow stack and code pointer integrity, respectively. Readers are encouraged to go through the original works to learn the details of these protection schemes.

*1) **Shadow Stack**:* Shadow Stack protection [14] stores return addresses in a protected shadow stack. This scheme adds a function prologue and epilogue: in the prologue, the return address is saved to the shadow stack with write permission temporarily enabled, then immediately reverted to read-only. The R15 register points to the latest shadow stack location, which is popped in the epilogue and checked against the return address in the regular stack. If they don't match, the function crashes, effectively blocking the ROP attack.

*2) **Code Pointer Integrity**:* Code Pointer Integrity [33] identifies sensitive code pointers through static analysis. It then isolates these sensitive pointers in a safe region and modifies the code so that instructions access this safe region instead of the regular region when accessing these objects. Accesses to the safe region are sandwiched by enabling and disabling permission updates specific to that region. We use the relaxed variant of CPI, which is code pointer separation.

VII. Evaluation

For evaluation, we first find simulation points using the simpoint profiling tool [48] for the first 100 billion instructions with an instruction interval of 100 million instructions. The top five simpoint intervals are simulated in detail with the gem5's O3 CPU model, and the final IPC is computed based on the weight of the corresponding interval. In addition to SpecMPK, we also simulate two distinct microarchitectures for WRPKRU: serialized and NonSecure speculative. In the serialized version, the execution of WRPKRU instruction is serialized. In the NonSecure speculative case, the PKRU register is renamed through the PRF. Therefore, this version

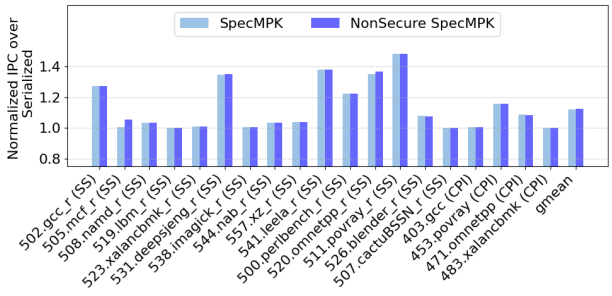

Fig. 9: Normalized IPC over serialized WRPKRU microarchitecture. Workloads labeled as SS and CPI indicate shadow stack and code pointer integrity protection, respectively.

avoids stalls caused by frequent WRPKRU occurrences in the dynamic instruction stream. We denote this microarchitecture as NonSecure SpecMPK. On the contrary, SpecMPK may lead to a stall in the pipeline when the $\mathrm{ROB}_{\mathrm{pkru}}$ becomes full. In the NonSecure SpecMPK microarchitecture, memory instructions only check the most recent PKRU update, therefore potentially exposing side channels.

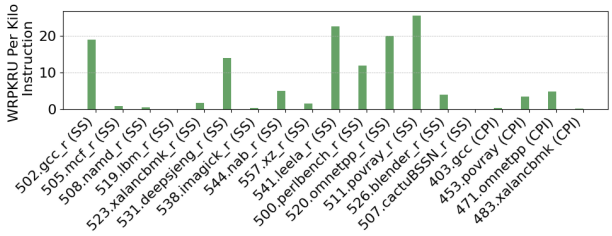

Fig. 10: Frequency of WRPKRU instructions in the dynamic instruction stream.

Figure 9 illustrates the normalized IPC by both SpecMPK and NonSecure SpecMPK. Since SpecMPK does not observe frontend serialization stalls, it achieves an increased instruction issue rate over serialized WRPKRU microarchitecture. The stall in the retirement stage due to the protection check for the store instructions failing *PKRU Store Check* and execution of load instructions failing *PKRU Load Check* is insignificant. Therefore, the speedup achieved by SpecMPK is similar to

that of NonSecure SpecMPK across the simulated workloads. This performance enhancement aligns with the frequency of WRPKRU instructions, as depicted in Figure 10. On average, SpecMPK achieves a 12.21% speedup over the baseline serialized model (maximally 48.42%). Workloads characterized by high WRPKRU per kilo instructions experience substantial speedups.

*1) **Sensitivity Analysis**:* The SpecMPK microarchitecture stalls the pipeline frontend if $\mathrm{ROB}_{\mathrm{pkru}}$ is full. Therefore, the performance benefit of SpecMPK is sensitive to the size of the $\mathrm{ROB}_{\mathrm{pkru}}$. The ideal size of the $\mathrm{ROB}_{\mathrm{pkru}}$ depends on the instruction window length of the CPU, which is equal to the number of entries in the Active List.

To evaluate the performance of SpecMPK we vary the size of $\mathrm{ROB}_{\mathrm{pkru}}$ while keeping the size of the Active List constant. Additionally, we compare the performance against NonSecure SpecMPK to evaluate the smallest $\mathrm{ROB}_{\mathrm{pkru}}$ that achieves performance close to NonSecure architecture.

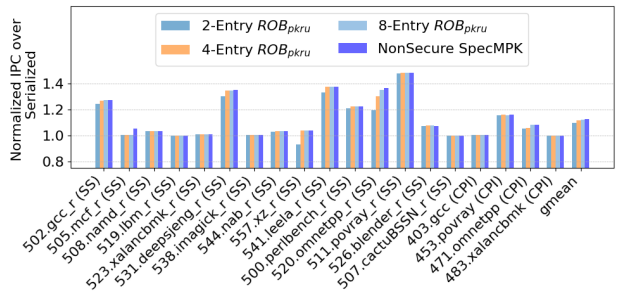

Fig. 11: Normalized IPC for various sizes of $\mathrm{ROB}_{\mathrm{pkru}}$

In this experiment, we explore three distinct ratios between the number of entries for $\mathrm{ROB}_{\mathrm{pkru}}$ and the Active List: $1/96$, $1/48$, and $1/24$. As we reduce the ratio, the probability of stall becomes higher since it reduces the minimum number of WRPKRU instructions required to stall the pipeline. Figure 11 demonstrates the performance for these configurations with ratios $1/96$, $1/48$, and $1/24$ corresponding to 2,4 and 8 entry $\mathrm{ROB}_{\mathrm{pkru}}$ respectively. Workloads with higher WRPKRU per kilo instructions display lesser performance improvements at lower ratios. While *502.gcc_r (SS)*, *500.perlbench_r (SS)*, *531.deepsjeng_r (SS)*, *541.leela_r (SS)*, *526.blender_r (SS)*, *453.povray_r (CPI)* benchmarks achieve performance similar to the NonSecure SpecMPK at the 1/48 ratio, *520.omnetpp_r (SS)* and *471.omnetpp_r (CPI)* require the ratio to be 1/24 to match the performance of NonSecure-SpecMPK. The remaining workloads primarily contain very few WRPKRU instructions in the dynamic instruction stream, resulting in minimal performance changes across various configurations.

VIII. Hardware Overhead

SpecMPK requires the addition of the following microarchitecture components: $\mathrm{ARF}_{\mathrm{pkru}}$, $\mathrm{ROB}_{\mathrm{pkru}}$, *AccessDisableCounter*, and *WriteDisableCounter*. Additionally, the StoreQueue requires additional bits to specify whether store-to-load forwarding is disabled for each entry. SpecMPK stores two bitmaps in ROB_{pkru} to specify the `pKeys` that need to decrement the counters in the event of either a commit or a squash. For the configuration in Table III, our design requires 93B of sequential logic, which is approximately 0.19% of the L1 data cache. We have also synthesized the RTL module that includes the counters, $\mathrm{ROB}_{\mathrm{pkru}}$ and $\mathrm{ARF}_{\mathrm{pkru}}$ with associated combinational logics for updating them. The synthesis was done using a 45nm node, and it reported an area overhead of $5887.91 \mu m^2$ with 3103 logic cells. Additionally, CACTI reports 2.02% dynamic and 0.39% leakage power overhead due to the new components when compared to access to the 48kB L1 data cache.

IX. Security Analysis and Discussion

In this section, we begin by articulating the three properties ensuring SpecMPK's security guarantees. We then analyze how SpecMPK protects against the memory vulnerabilities like MPK. Next, we explain how the same properties ensure SpecMPK does not introduce new speculative execution vulnerabilities. We additionally demonstrate proof-of-concept attacks on NonSecure SpecMPK that result in side channels due to transient permission upgrades and explain how SpecMPK mitigates them. At last, we discuss the non-security use cases of MPK and explain why SpecMPK would not affect the functionality of such use cases.

A. Security Properties of SpecMPK

SpecMPK has the following three properties that ensure that it mitigates memory vulnerabilities the same way as MPK and prevents potential side channels due to speculative permission upgrades.

1) **Ordering**: SpecMPK issues each memory instruction in order in relation to all WRPKRU instructions within the *WRPKRU-window* (see Section V-A for the definition). Therefore, it ensures all memory operations are under the control of the corresponding WRPKRU instruction.
2) **Conservative transient access control**: SpecMPK stalls transient memory instructions with potential protection violations by capturing all WRPKRU updates within the *WRPKRU-window*. With this conservative access control, SpecMPK ensures that no transient memory accesses can violate its access capability.
3) **Precise non-speculative access control**: SpecMPK ensures precise protection for stalled memory instructions with potential protection violation by using $\mathrm{ARF}_{\mathrm{pkru}}$ for protection check at the commit stage.

B. Security Comparison: MPK and SpecMPK

MPK has been used for improving memory safety by mitigating memory corruption, buffer overread, and *control-steering* attacks. These protections remain effective with our proposed SpecMPK, as showcased in Figure 12. Additionally, we demonstrate that SpecMPK prevents speculative permission upgrades.

For this analysis, we assume that WRPKRU instructions have their values to be written to PKRU independent of the control flow, i.e., the value is independent of speculation. This

is achieved through compiler support by using load-immediate for the EAX register, which is the implicit source operand of the WRPKRU instruction, and eliminating branch instructions between load-immediate and the subsequent WRPKRU instruction.

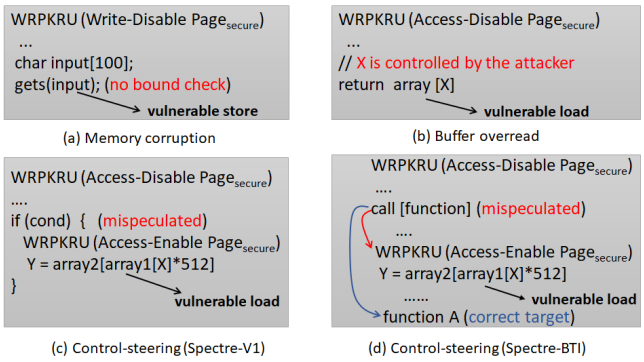

Fig. 12: Examples of various vulnerability mitigations utilizing MPK. $\mathrm{Page}_{\mathrm{secure}}$ stores array1 in example (c) and (d).

*1) **Vulnerability mitigations similar to MPK**:* We demonstrate SpecMPK's mitigation capability to prevent memory corruption and overread using examples shown in Figure 12(a) and 12(b), respectively.

Protection against unauthorized Stores: Figure 12(a) demonstrates the use of MPK to mitigate memory corruption vulnerability. In this example, $\mathrm{Page}_{\mathrm{secure}}$ is protected with Write-Disable permission within unsafe code sections to prevent corruption. WRPKRU serialization ensures that WRPKRU commits before the execution of a younger store instruction in *gets*, as shown in Figure 12(a), causing the vulnerable store to $\mathrm{Page}_{\mathrm{secure}}$ to raise a protection fault, effectively mitigating the memory corruption vulnerability.

SpecMPK successfully mitigates this attack. SpecMPK's ordering property ensures that it captures all WRPKRU updates within the *WRPKRU-window* before the vulnerable store is issued. Since *WRPKRU-window* includes at least one update with Write-Disable permission for $\mathrm{Page}_{\mathrm{secure}}$, either $\mathrm{ARF}_{\mathrm{pkru}}$ or *WriteDisableCounter* indicates a potential protection fault for the corresponding `pKey`. Using conservative transient access control property, SpecMPK identifies potential protection violation as *PKRU Store Check* fails if a store instruction attempts to speculatively write to $\mathrm{Page}_{\mathrm{secure}}$. Finally, the precise non-speculative access control property enforces protection fault check at commit to effectively mitigate this memory corruption attack, as it correctly raises a protection fault since the most recent WRPKRU with Write-Disable permission must have been committed to $\mathrm{ARF}_{\mathrm{pkru}}$.

Protection against unauthorized loads: MPK mitigates buffer overread attack, as shown in Figure 12(b), where $\mathrm{Page}_{\mathrm{secure}}$ has Access-Disable permission within unsafe code sections. WRPKRU serialization mitigates overread by raising a protection fault for the vulnerable load instruction accessing $\mathrm{Page}_{\mathrm{secure}}$, as this load can only execute after the most recent WRPKRU has been committed.

SpecMPK successfully mitigates this vulnerability. Using the same principle as discussed for the memory corruption mitigation but applied to loads, SpecMPK ensures that *PKRU Load Check* would fail for the vulnerable load instruction in Figure 12(b) if it speculatively accesses $\mathrm{Page}_{\mathrm{secure}}$. Therefore, the load instruction is stalled and marked to be issued at retirement. Precise non-speculative access control property ensures that re-issue of such a vulnerable load at retirement leads to a protection violation since the $\mathrm{ARF}_{\mathrm{pkru}}$ has the most recent update, which includes the Access-Disable permission for $\mathrm{Page}_{\mathrm{secure}}$ — effectively mitigating the overread attack.

*2) **Transient permission upgrade vulnerability mitigation**:* Figure 12(c) and 12(d) present two scenarios of transient permission upgrades which are similar to Spectre-V1 and Spectre-BTI (branch target injection) vulnerabilities, respectively. The former *control-steering* attack example exploits branch misprediction as taken, while the latter exploits indirect branch target misprediction to divert control flow to a code section that upgrades permission. WRPKRU serialization prevents speculative permission upgrades, blocking both attacks. Conversely, NonSecure SpecMPK architecture allows transient permission upgrades in both scenarios, subsequently enabling younger vulnerable loads to establish a side channel.

SpecMPK successfully mitigates these vulnerabilities. The ordering property ensures that the WRPKRU with the Access-Disable permission within the *WRPKRU-window* is correctly captured in either $\mathrm{ARF}_{\mathrm{pkru}}$ or *AccessDisableCounter* when the vulnerable load issues. Subsequently, SpecMPK's conservative transient access control property ensures that the vulnerable load stalls until retirement since *PKRU Load Check* fails. Since SpecMPK ensures such access instructions update the microarchitectural state non-speculatively, i.e., there is no protection fault, it effectively mitigates the *control-steering* attacks presented in Figure 12(c) and 12(d). Enforcing execution at retirement for load instructions with a potential protection fault mitigates all variants of *control-steering* and *chosen-code* attacks on a disabled page. Precise non-speculative access control property remains unused in mitigating these vulnerabilities as the vulnerable load instruction never reaches the head of the Active List.

*3) **Other mitigations**:* SpecMPK blocks store-to-load forwarding to prevent speculative buffer overflow (see Section III-C for the attack description), which exploits a transient Write-Enable permission upgrade. It also eliminates the speculative side channel through TLB by conservatively differing the TLB update until retirement time.

C. Proof-of-Concept Attack on NonSecure SpecMPK and Mitigation by SpecMPK

In this subsection, We analyze NonSecure SpecMPK vulnerability with the example shown in Figure 12(c) and demonstrate SpecMPK's mitigation capability using gem5. This code is part of a victim function in which the branch is trained to be taken initially and later mispredicted during an attack that uses

Flush+Reload side channel. The value of X differs between the training and attack phases.

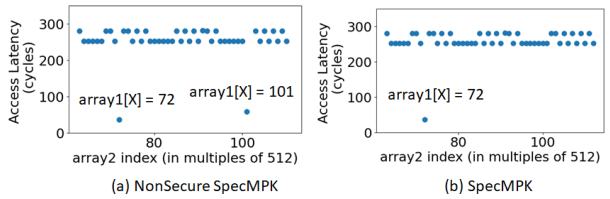

Fig. 13: Access latency for the array2 indices during the reload phase of the flush+reload attack.

Figure 13 provides the access latency observed for array2 indices in the reload step for NonSecure SpecMPK and SpecMPK microarchitectures. NonSecure SpecMPK shows low access latency for two indices of array2 (in multiples of 512), indicating cache hits: 72, the value of array1[X] during training, and 101, the value when the branch mispredicts. On the contrary, with the SpecMPK microarchitecture, the cache hit only occurs when array1[X] is 72, successfully preventing speculative access to array1 when the branch is mispredicted.

D. *Non-security Use Cases of MPK*

It is essential that SpecMPK does not result in unintended side effects due to speculation, so that it can replace MPK across all scenarios. In this subsection, we utilize a use-case of MPK, dynamic data race detection in multithreaded programs, to show how SpecMPK avoids any side effects.

Besides providing in-process isolation to mitigate various vulnerabilities, MPK is also efficient in identifying data race in multi-threaded applications dynamically [8], [66]. A potential data race occurs due to inconsistent lock usage [2]. One example of inconsistent lock usage is when concurrent threads write to a shared memory object using different locks. These proposals assign a `pKey` to each shared object and map access policies to the per-thread PKRU register. Additionally, these proposals rely on protection faults to identify data races. For instance, Kard [8] identifies shared objects accessed in each critical section dynamically by assigning these objects a `pKey` with Access-Disable permission. Therefore, access to these objects would result in a protection fault, which Kard traps, identifying the shared objects for the corresponding lock. Additionally, Kard allows only one thread to acquire write permission for a shared object. Accessing the same shared object in another thread using a different lock would result in a protection fault as the permission for the associated `pKey` for this thread specifies Access-Disable permission. Kard traps such faults and detects potential data races.

SpecMPK is capable of successfully replacing MPK in this usage scenario. Since disabling and enabling permission must occur before access to the shared object (e.g., Kard updates PKRU when a thread enters a critical section), the *WRPKRU-window* captures the most recent disabling permission update either in $\mathrm{ARF}_{\mathrm{pkru}}$ or in the *Disabling Counters*. SpecMPK's ordering property ensures that memory instructions accessing a shared object are always issued after all preceding WRPKRU instructions. As a result, conservative transient access control property identifies accesses with potential protection violation if the preceding WRPKRU has disabling permission for the associated `pKey`. Finally, the precise non-speculative access control raises protection violation correctly, as the most recent update in relation to the shared objects must have been committed to the $\mathrm{ARF}_{\mathrm{pkru}}$ – correctly identifying potential data races for the shared object.

X. Related Works

A. *Memory Domains*

Intel MPK permits a maximum of sixteen keys, which may prove inadequate for certain applications. Servers handling hundreds of clients simultaneously and requiring key isolation for each client demand more than sixteen keys. To facilitate the use of more than sixteen keys, both libmpk [40] and VDom [64] suggest domain virtualization. This approach effectively maps a large number of virtual domains to the limited number of physical domains by either disabling pages or leveraging Address Space ID, thus creating an extensive array of virtual domains.

[58] proposes hardware-based domain virtualization for Persistent Memory Objects (PMO) using Domain Translation Table. On the contrary, [17], [44] propose architecture changes to enable 1024 physical domains stored in the unused bits in the Page Table Entry.

Other hardware-based in-process isolation proposes ISA extension [11], [39] to avail scalable isolation. In both works, the instruction that switches between domains are serialized. However, the latter work allows the user to choose whether a domain switch requires to be serialized.

XI. Conclusion

In this study, we analyze MPK to gain a deeper understanding of its performance and security implications across various protection scenarios. Our focus lies in examining the efficiency of MPK in safeguarding against different types of threats. Specifically, as MPK is employed to defend against a broad spectrum of attacks, we identify situations where this protection strategy proves to be costly in terms of performance.

One notable instance is the application of MPK to block memory corruption, a critical measure to counteract control and data-flow hijack. Unfortunately, this approach introduces significant performance overhead, primarily due to the serialization of the WRPKRU instruction. This serialization strategy is crucial for preventing speculative attacks.

In this work, we propose SpecMPK, a novel microarchitecture enhancement that identifies and mitigates potential side channels while allowing speculative execution of the WRPKRU instructions. We show that with this microarchitecture solution, we achieve significant performance gains while safely stalling instructions that are susceptible to speculative attacks, and allowing other instructions to execute speculatively.

REFERENCES

[1] "Armv8.5-a memory tagging extension," file:///home/dadak/Downloads/Arm_Memory_Tagging_Extension_Whitepaper-2.pdf.

[2] "Inconsistent lock usage," https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/inconsistent-lock-use.html.

[3] "Intel® 64 and ia-32 architectures software developer's manual," https://cdrdv2.intel.com/v1/dl/getContent/671200.

[4] "Memory access permissions and domains," https://developer.arm.com/documentation/dui0056/d/caches-and-tightly-coupled-memories/memory-management-units/memory-access-permissions-and-domains.

[5] "Memory access permissions and domains," https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf.

[6] "Msrc-security-research," https://github.com/microsoft/MSRC-Security-Research.

[7] "Trends and challenges in the vulnerability mitigation landscape." Santa Clara, CA: USENIX Association, Aug. 2019.

[8] A. Ahmad, S. Lee, P. Fonseca, and B. Lee, "Kard: Lightweight data race detection with per-thread memory protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 647–660.

[9] A. Akram and L. Sawalha, "Validation of the gem5 simulator for x86 architectures," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 53–58.

[10] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 151–164.

[11] A. Bhattacharyya, F. Hofhammer, Y. Li, S. Gupta, A. Sanchez, B. Falsafi, and M. Payer, "Securecells: A secure compartmentalized architecture," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2921–2939.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[13] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM symposium on information, computer and communications security*, 2011, pp. 30–40.

[14] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.

[15] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 481–493.

[16] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[17] L. Delshadtehrani, S. Canakci, M. Egele, and A. Joshi, "Sealpk: Sealable protection keys for risc-v," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1278–1281.

[18] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[19] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[20] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "{IMIX}:{In-Process} memory isolation {EXtension}," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 83–97.

[21] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: language-based restriction of untrusted libraries," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 255–267.

[22] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1871–1885.

[23] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[24] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 368–379.

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[26] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "The taming of the stack: Isolating stack data from memory errors," in *NDSS*, 2022.

[27] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith, "Ghostbusting: Mitigating spectre with intraprocess memory isolation," in *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, 2020, pp. 1–11.

[28] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[29] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Pkru-safe: automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 132–148.

[30] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[31] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 437–452.

[32] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[33] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018, pp. 81–116.

[34] A. B. Kvalsvik, P. Aimoniotis, S. Kaxiras, and M. Själander, "Doppelganger loads: A safe, complexity-effective optimization for secure speculation schemes," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[36] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[37] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 747–761.

[38] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture." in *USENIX Security Symposium*, vol. 10, 2006, pp. 209–224.

[39] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita *et al.*, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 266–281.

[40] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel {MPK})," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.

[41] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed," in *Annual Computer Security Applications Conference*, 2021, pp. 824–836.

[42] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[43] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Understanding selective delay as a method for efficient secure speculative execution," *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1584–1595, 2020.

[44] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys–efficient {In-Process} isolation for {RISC-V} and x86," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1677–1694.

[45] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[46] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary {CPU} architectures," in *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[47] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 762–775.

[48] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002.

[49] K. Sinha and S. Sethumadhavan, "Practical memory safety with rest," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 600–611.

[50] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[51] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "{ERIM}: Secure, efficient in-process isolation with protection keys ({{{{{MPK}}}}}," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[52] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[53] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[54] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang, "Seimi: Efficient and secure smap-enabled intra-process memory isolation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 592–607.

[55] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[56] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

[57] J. Wikner and K. Razavi, "{RETBLEED}: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3825–3842.

[58] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 680–692.

[59] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[60] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE micro*, vol. 16, no. 2, pp. 28–41, 1996.

[61] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 429–438.

[62] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 707–720.

[63] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[64] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and unlimited virtual domains on multiple architectures," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 905–919.

[65] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "{BunnyHop}: Exploiting the instruction prefetcher," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7321–7337.

[66] D. Zhou and Y. Tamir, "Push: Data race detection based on hardware-supported prevention of unintended sharing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 886–898.

[67] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-fat: Architectural support for low overhead memory safety checks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 916–929.

[68] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "Zerø: Zero-overhead resilient operation under pointer integrity attacks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 999–1012.