PTStore: Lightweight Architectural Support for Page Table Isolation

Wende Tan[1], Yangyu Chen[2], Yuan Li[1], Ying Liu[1], Jianping Wu[1], Yu Ding[3], and Chao Zhang[1,✉]
[1]*Tsinghua University, Beijing, China* [2]*Chongqing University, Chongqing, China* [3]*Google, San Jose, CA, USA*
twd2.me@gmail.com, chaoz@tsinghua.edu.cn

***Abstract*—Page tables are critical data structures in kernels, serving as the trust base of most mitigation solutions. Their integrity is thus crucial but is often taken for granted. Existing page table protection solutions usually provide insufficient security guarantees, require heavy hardware, or introduce high overheads. In this paper, we present a novel lightweight hardware-software co-design solution, PTStore, consisting of *a secure region* storing page tables and *tokens* verifying page table pointers. Evaluation results on FPGA-based prototypes show that PTStore only introduces <0.92% hardware overheads and <0.86% performance overheads, but provides strong security guarantees, showing that PTStore is efficient and effective.**

I. Introduction

Page tables (PTs) are critical data structures in OS kernels. They store virtual-to-physical address mappings and access permissions of each virtual memory (VM) page. The integrity of page tables affects the security of kernels and user-space applications, since corruptions to page tables will change the locations of code and data as well as their access permissions. More importantly, their integrity also affects the effectiveness of most existing mitigation solutions deployed in either kernels or user space. For instance, the well-known Data Execution Prevention (DEP) follows the access permission settings in page tables to stop code injection and corruption attacks. In addition, *all defense solutions* that rely on code instrumentation to deploy security checks, e.g., Control-Flow Integrity (CFI) [1], all depend on VM to protect their instrumented checks or metadata from being corrupted [2], [3].

The integrity of page tables has been taken for granted for a long time. However, it can be corrupted in practice. A prior research PT-Rand [4] demonstrates that a sophisticated attacker can exploit memory corruption vulnerabilities to locate page tables, change page permissions to disable DEP, replace kernel code with malicious code, and finally jump to injected code with legal kernel interfaces (e.g., system calls). This attack would not be caught by defense solutions, including CFI, since the attacker does not trigger any illegal control-flow transfer. Nonetheless, the instrumented CFI checks can be disabled by this attack, too. In other words, the integrity of page tables can be corrupted, which renders existing defenses useless and enable attackers to launch attacks, including kernel hijacking. Thus, a solution to protect the integrity of page tables is highly demanded.

Several kinds of solutions have been proposed to protect page tables. The first kind of solution *randomizes the locations of page tables* in the VM space [4], [5]. Similar to Address Space Layout Randomization (ASLR) [6], this kind of solution may also suffer from attacks like information disclosure [4]. The second kind of solution *physically isolates page tables from other memory regions* [2], [7], [8], [9]. These mechanisms usually rely on hypervisors and introduce non-negligible extra performance overheads (>5%), even though accelerated by heavy hardware [7], [10], [11]. Thus, they are impractical and unaffordable for resource-limited systems (e.g., IoT devices). The third kind of solution *virtually isolates page tables from other memory* [12], [13], [14]. Some of these solutions have been deployed by the industry, e.g., Apple's iPhone [15]. However, such solutions still rely on permission bits in page tables, and suffer from a chicken-and-egg problem. Thus, only a few of them can mitigate page table injection or reuse attacks, and many of them may be bypassed due to TLB inconsistency [16].

Overall, existing solutions usually provide insufficient security guarantees, rely on heavy hardware, or introduce high overheads. Thus, a lightweight and effective defense solution is desirable.

In this paper, we present a novel lightweight hardware-software co-design solution, namely PTStore, to protect page tables against memory corruption vulnerabilities. Our solution mainly leverages hardware features to construct a secure region and store page tables, and relies on a novel token mechanism to further verify the integrity of page table pointers, which can stop all page table corruption, injection, and reuse attacks. First, the secure region can only be accessed via special instructions and thus protects page tables from being *corrupted* by regular instructions. Together with CFI, the page tables cannot be corrupted by the new special instructions as well. Second, the MMU enforces that page tables are retrieved from the secure region, which stops *injected* page tables from being used. Third, the token mechanism further guarantees the integrity of page table pointers in process control blocks, which further stops existing but out of context page tables from being *reused* by adversaries. As a result, PTStore can guarantee the integrity of legitimate page tables and enforce that the system uses the right page tables, thereby defeats page table attacks.

We have implemented an FPGA-based prototype of this solution on RISC-V. Evaluation results show that PTStore only costs <0.92% extra hardware resources to RISC-V BOOM cores, and slows down the whole system by only <0.86%, but provides much stronger security guarantees than existing solutions under various attacks, showing that PTStore is practical, efficient, and effective. Furthermore, PTStore is applicable to protect other critical data beyond page tables.

In summary, we make the following contributions:

- We propose a novel lightweight hardware-software co-design solution PTStore, consisting of a secure region and a novel token mechanism, to protect page tables.
- We build an FPGA-based prototype of PTStore on RISC-V.
- We conduct a thorough performance and security evaluation, showing that PTStore is lightweight, practical, and efficient, as well as effective against various page table attacks.

II. Background

A. Physical Memory Protection (PMP)

RISC-V and many other ISAs provide PMP support, which enables M-mode code to configure permissions of *physical* memory regions for S-mode code and data [17]. For example, M-mode code can mask some physical memory ranges so that kernels in S-mode are not permitted to access them.

B. Page Table Attacks

Page tables are critical data structures in kernels as they store address mappings and access permissions of VM pages. A system can get compromised if even only one bit of its page tables gets flipped [18]. However, page tables are vulnerable to both hardware [18] and software [4] vulnerabilities. Once memory-corruption vulnerabilities exist, attackers can utilize the following techniques to compromise the system (including kernels and even deployed mitigations), which is similar to code corruption, injection, and reuse attacks.

This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000) and NSFC #61972224.

Page Table Tampering (PT-Tampering): Attackers can exploit vulnerabilities to tamper with page tables [4], e.g., flipping critical permission bits or changing address mappings. For example, attackers can flip U-bit to (1) make kernel code and data pages accessible to user mode for tampering, or (2) make user pages accessible to the kernel and bypass defenses like Intel Supervisor-Mode Execution/Access Prevention (SMEP/SMAP). Attackers can also flip W-bit to disable the read-only restriction of code or critical data, and then overwrite code and critical data to bypass other deployed defense solutions.

Page Table Injection (PT-Injection): Attackers can also corrupt page table pointers to launch attacks. For instance, attackers can inject maliciously-crafted page tables into known locations [16], and hijack page table pointers stored in process control blocks (PCBs) to point to the injected page tables. Later, when the system accesses page tables via these pointers, the injected page tables will take place. This allows attackers to launch the same attacks as PT-Tampering allows.

Page Table Reuse (PT-Reuse): Attackers can also launch page table reuse attacks without injecting fake page tables. Instead, attackers can hijack page table pointers to point to existing data or page tables in the kernel [7]. For instance, attackers can first replace the page table pointer of a victim `root`-privileged process with that of attackers' normal privileged process. Then, the attackers can utilize the victim process to execute arbitrary (malicious) code under `root` privileges.

III. Design

A. Threat Model

As a kernel defense solution, our threat model is consistent with prior work [2], [4]. Specifically, we assume that:

- Attackers have full control of non-`root` user-mode processes and can interact with kernels using kernel interfaces like system calls.
- The kernels, are benign but have powerful memory-corruption vulnerabilities [4], which allow attackers to repeatedly use regular instructions to read from or write to arbitrary memory locations.
- DEP is deployed for the kernels so the kernel code is immutable.
- A fine-grained CFI [1] is also deployed for the kernels, so the integrity of the control flows of the kernels is enforced. Thus, attackers cannot reuse existing code to manipulate page tables.
- The boot procedures are secure and the attackers can only interact with the kernel after boot.
- The architectural behavior of the hardware is trusted, e.g., vulnerabilities like Rowhammer do not exist. However, the processor cores may have side-channel bugs [19] for attackers to gain information.

Besides, other kernel defense solutions may be deployed. Under this threat model, the attackers aim to tamper with the page tables to modify the mappings or permissions of the kernel address space and further bypass the deployed kernel defense solutions.

B. Intuition

Our goal is to protect the integrity of page tables. To do so, as illustrated in Figure 1, we put page tables into a hardware-enforced contiguous secure region in grey, distinguish page-table manipulation code from normal code, and only allow the former code to access the secure region (i.e., page tables), stopping PT-Tampering attacks. Also, we enforce that the PTW (page table walker) in the MMU only fetches page tables from the secure region to guarantee the integrity of the used page tables and stop PT-Injection attacks. Further, a special token in the secure region is associated with each process to ensure the integrity of the page table pointer and stop PT-Reuse attacks.

C. Design Choices

Our design slightly augments the processor core and the compiler. The OS kernel needs to be extended to utilize PTStore as well. Figure 2 shows the overall design (compilers are omitted). In this section, we first explore the design space and finally present our design choices.

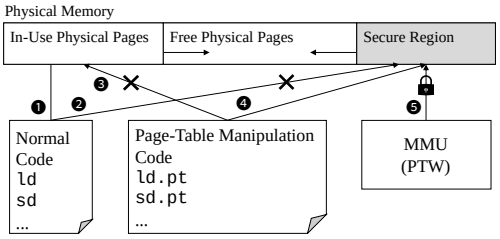

Fig. 1: Intuition of PTStore's Design.

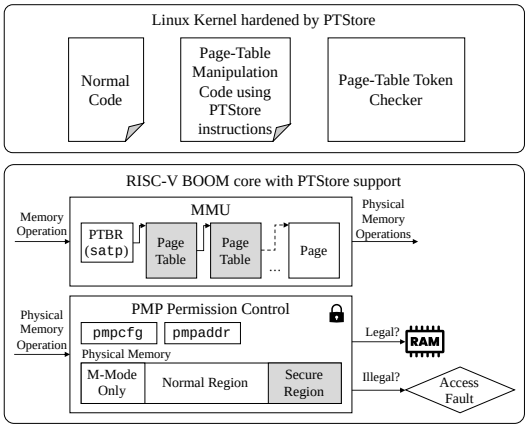

Fig. 2: Overview of PTStore's Design. The data in grey are in the secure region. PTBR refers to the page table base register.

1) A secure region to mitigate PT-Tampering: We put page tables into a hardware-enforced secure region to protect them from being tampered. Then, we need to allow legitimate code, i.e., the page-table manipulation code, to access the secure region. Meanwhile, all other code should be forbidden to access the secure region.

Many existing isolation solutions [7], [11], [12], [13], [15] choose to leverage control registers or other interfaces to switch the access permissions of the secure region and grant temporary access for their legitimate users when needed. However, these solutions, either implemented in pure software or assisted by control registers, need to execute many extra instructions and thus introduce many extra clock cycles. Besides, the system states (e.g., the control registers) have to be taken care of during context switching. As pointed in [4], some solutions may leave a small time window for attackers in other threads to tamper with the secure regions, too. Even in the same thread, many unrelated load or store instructions can now access the secure regions as well, which breaks the security principle of least privilege.

PTStore chooses to utilize a pair of newly introduced load and store instructions dedicated to accessing the secure region. These instructions provide users with the support of fine-grained data-flow isolation. We also extend the processor core to recognize the new instructions, make the secure region accessible only to the new instructions (❹ in Figure 1), and throw exceptions when the secure region is accessed by regular instructions (❷ in Figure 1).

At the same time, we will find out all page-table manipulation code in kernels and augment them to use the new instructions to access page tables, so as the processor core will grant the access. Note that we do not instrument any extra instructions. Thus, we can access the secure region efficiently while eliminating any potential permission-switching time window.

In this way, the secure region access policy can be enforced effectively and efficiently by the hardware. Attackers can neither tamper with page tables via regular instructions nor via the new instructions, as long as the CFI is guaranteed.

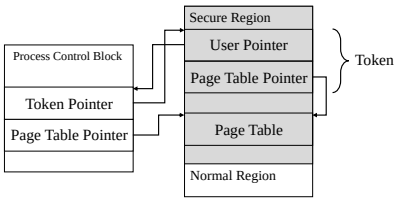

Fig. 3: Design of the Token Mechanism.

2) A physical-memory-based approach to mitigate PT-Injection: Attackers can also launch PT-Injection attacks. To address this, we enforce all page table users only accept page tables from the secure region. Each time the MMU fetches page tables (❺ in Figure 1), it checks whether the page table being fetched is from the secure region. We also enforce that the new instructions used by page-table manipulation code can only access the secure region (❸ in Figure 1). Thus, we can ensure that all users of page tables refuse injected fake page tables, no matter how page table pointers are corrupted.

To make this work, we need an efficient way to determine whether a page table is in the secure region. Different from the isolation solutions based on VM [14], we cannot use page permissions or any property of the virtual pages to differentiate the secure region from normal pages, since these properties are stored in the need-to-protect page tables. Otherwise, the PTW would have to retrieve the page table entry of the page table being fetched, to check whether it is in the secure region, forming a chicken-and-egg problem.

Further, we cannot use virtualization-based approaches as well. Although hypervisors can configure the permissions of each physical page of their guest OSes by utilizing nested page tables (NPTs) [7], [8], [20], they are usually heavy. Specifically, to utilize hypervisors, complex hardware components, including NPTs, need to be activated. This will increase the area and the power consumption of the processor and is unsuitable for resource-limited systems like embedded systems. Also, the performance of the whole system will be degraded by >5% [7], [10], [11], primarily due to the NPTs.

Therefore, we propose to develop *physical-memory-based approaches*. We extend the PMP to add a new permission bit to each physical memory region to identify the secure regions, so that the processor core can perform secure region checks efficiently following the existing PMP logic and deny any illegal access issued by the new or regular instructions, or the PTW. Note that PMP requires each memory region to have contiguous physical addresses. This is compatible with other mitigations, e.g., [18], which can be deployed together to further mitigate hardware attacks like Rowhammer.

3) Tokens to mitigate PT-Reuse: The last possible kind of attack till now is PT-Reuse. Pointer signatures, e.g., SipHash [7] or ARM Pointer Authentication (PA), can be used to protect page table pointers and mitigate these attacks. However, pure software mechanisms do not have the best performance, while hardware mechanisms may be absent due to hardware budgets. Besides, they are not resilient to side-channel attacks [19]. Alternatively, utilizing the secure region and the new instructions, we present a novel token-based mechanism to protect the page table pointers, even though they are stored in a non-secure region (e.g., PCBs), without further hardware modifications.

As shown in Figure 3, the token mechanism stores several *tokens* in the secure region. Each token consists of a page table pointer and a user pointer, indicating the page table pointer to be protected and its *unique* legitimate user. Correspondingly, each PCB is augmented to store a pointer to the token of its page table pointer, so that the process can use this token as a credential to update the PTBR.

We also need to augment OS kernels to maintain and verify the tokens. The augmented kernels will issue a token in the secure region whenever a process is created, copy the token whenever the page table pointer of a process is (legitimately) copied, and clear the token when a process is destroyed. Further, the kernels will check the integrity of each page table pointer when it is used, e.g., when the PTBR is updated to the page table pointer of a new process, by validating the token associated with this page table pointer. The token is valid if the user pointer in the token points back to the token pointer in the PCB and the two page table pointers in the token and the PCB match.

In this way, a legitimate page table pointer in a PCB is bound only to this PCB, since attackers cannot tamper with the tokens in the secure region. The page table pointer is thus cannot be badly reused.

TABLE I: #Lines of code of each PTStore component.

Components	Language	#Lines of Code		
		Added	Changed	Total
RISC-V Processor	Chisel	24	34	58
LLVM Back-end	C++ and TableGen	15	0	15
Linux Kernel	C	767	638	1,405
Total	–	806	672	1,478

IV. Implementation

We have implemented a prototype of PTStore based on a RISC-V BOOM (Berkeley Out-of-Order Machine) core, the LLVM compiler, and the Linux kernel, on an FPGA. As Table I shows, PTStore has a small number of lines of code and is very easy to deploy in practice.

A. ISA Extensions

1) Processor core modifications: First, we extend the processor core to support the ISA extensions of PTStore, including the secure region and the new instructions. We choose to extend the RISC-V ISA and the RISC-V BOOM core for simplicity.

To mark a secure region, we add a new S-bit into the `pmpcfg` CSR of each memory region, where S means secure. We also augment the core to trigger an *access fault* to deny any illegal access.

Then, we add two new secure region access instructions `ld.pt` and `sd.pt`. They are similar to the existing load and store instructions, except they have different opcodes and only access the secure region.

Further, the PTW should only access the secure region. However, before the secure region is set up at boot, this PTW check should be disabled. Thus, we add a new S-bit in the `satp` CSR to inform the PTW whether the secure region check is enabled, and augment the PTW to perform the check. When the S-bit is set, the PTW will only access the secure region and refuse injected fake page tables.

2) Compiler suite modifications: Besides, following the LLVM framework, we add the new instructions, i.e., `ld.pt` and `sd.pt`, to the RISC-V ISA description files to generate proper machine code.

B. Supervisor Binary Interface (SBI) Extensions

In the RISC-V ISA, only M-mode code can access the `pmpcfg` CSRs and set up the secure region. To allow S-mode kernels to manage the secure region, we provide new SBI functions to initialize, get, and set the secure region boundary.

C. Kernel Extensions

To protect the page tables, we first allocate physical pages from the secure region, store all page tables into them, and enable the secure region check of the PTW. To further protect the page table pointers, we allocate tokens from the secure region, maintain the tokens, and check the tokens each time the `satp` CSR is to be updated.

1) Secure Region Management: The Linux kernel uses a buddy system allocator to manage all physical pages by zones and serve allocation requests of different requirements described by different GFP (i.e., get free pages) flags. To allocate pages from the secure region, we add a `PTStore` zone at the high physical addresses, and introduce a `GFP_PTSTORE` flag to request pages from only this zone.

Note that PMP requires each region to have contiguous physical addresses, so we design a mechanism to dynamically adjust the secure region on demand. Specifically, the `PTStore` zone is initialized to 64 MiB when the kernel boots. To adjust the secure region, we first allocate several contiguous physical pages adjacent to the secure region boundary from the normal zone by leveraging the `alloc_contig_range()` function. Secondly, we release the allocated contiguous pages to the `PTStore` zone. Thus, the secure region gets more free pages and still has contiguous physical addresses. Then, we update the secure region boundary. Finally, we try to allocate pages from the secure region again and it should succeed this time.

2) Page Table Manipulation: After the page tables are stored in the secure region, regular instructions cannot access them, so all legitimate page table manipulation code, e.g., the `set_pXd()` macros, should be augmented to use the new instructions, i.e., `ld.pt` and `sd.pt`.

3) Slab Allocator for Token Manipulation: The kernel uses slab allocators to allocate (usually small) kernel objects. Different slab allocators can have different GFP flags to allocate from different underlying physical pages and have different constructors. To allocate the tokens, we add a new `PTStore` slab allocator. This allocator has the `GFP_PTSTORE` flag set to allocate the underlying pages from the secure region so that the tokens can also be stored in the secure region. The constructor will then initialize all new tokens to zeros.

4) Process Management: Finally, we maintain the token for each process during its whole lifetime to protect its page table pointer, e.g., in `copy_mm()` and `switch_mm()` functions.

V. Evaluation

We evaluate our prototype PTStore system by addressing four important questions: hardware resource cost, functionality correctness, runtime performance, and security guarantees.

A. The Prototype System

To evaluate our PTStore solution, we build an FPGA-based prototype system with the modified RISC-V BOOM core mentioned in Section IV and necessary peripherals to boot the Linux kernel, as listed in Table II. The floating-point unit (FPU) of the core is disabled since the FPU will take a large number of hardware resources and hide the hardware overheads we introduced.

We use the Linux kernel 5.14 without and with PTStore. We enable the Clang CFI mitigation provided by LLVM for the kernels, since our threat model requires that CFI is deployed. We then compile these kernels using LLVM 13 with the new instruction support. The used kernel configuration is `defconfig` with the drivers of the aforementioned Xilinx IP cores.

B. Hardware Resource Cost

The prototype system is then synthesized and mapped to a Xilinx Kintex 7 FPGA (`XC7K420T`) by using Xilinx Vivado 2021.2. Our target frequency is $F_{target} = 90.000\text{MHz}$, which is almost the maximum frequency of RISC-V BOOM cores on our FPGA. In Table III, we collect the usages of lookup tables (LUTs) and flip-flops (FFs), and the worst setup slack (WSS) of the whole prototype systems without and with PTStore. To highlight the hardware resource overheads introduced to processor cores by PTStore, we also synthesize the RISC-V BOOM cores without and with PTStore out of context (i.e., without any peripheral) and collect the usages of LUTs and FFs, and the WSS. Besides, the maximum frequency is estimated by using $F_{max} = \frac{1}{11.111\text{ns} - WSS}$ for reference purposes.

TABLE II: Configurations of our prototype system.

Components	Configurations
ISA Extensions	RV64IMAC with M, S, and U modes
BOOM Config	`SmallBooms`
Caches	16KiB 4-way L1I$, 16KiB 4-way L1D$
TLBs	32-entry I-TLB, 8-entry D-TLB
Peripherals	Xilinx MIG for a 4GiB DDR3 SO-DIMM Xilinx AXI Ethernet, 64KiB Boot ROM

The results show that PTStore takes only <0.92% extra hardware resources and does not affect the maximum frequency. If other ISA extensions (e.g., FPU) are included or the processor core uses a more complex microarchitecture, the hardware cost will become negligible. ***Thus, PTStore is lightweight to implement on hardware.***

C. Regression Tests for Correctness Evaluation

To confirm that our modifications to the kernel do not introduce any new bug, we run the Linux Test Project (LTP) (commit `b26d1317`) on both the modified and original version of the kernel. As we have disabled the FPU of the processor core, we skip the tests using FPU.

Finally, we compare the outputs of the two runs and do not find any deviation, showing that ***PTStore does not introduce any new bug, and the modified kernel behaves well.***

D. Performance Evaluation

We first conduct microbenchmarks to study how PTStore influences the performance of syscalls and trap handlers. Then, we perform macro benchmarks to evaluate the performance of the whole system. These benchmarks are run on the original kernel without the Clang CFI to provide a baseline. Also, we run the benchmarks on the original and modified kernel, both with the Clang CFI, and mark them as `CFI` and `CFI+PTStore`, respectively.

1) Microbenchmarks: We use LMBench 3.0-a9 as our microbenchmark suite. Each benchmark is executed 1,000 times and the average relative performance overheads are reported in Figure 4. Note that the negative overheads should be due to the changed code layout and cache hit rate, or random errors since their execution time is short.

Since LMBench only repeatedly creates and waits for one process every iteration in the `fork()` tests, LMBench can only occupy a fixed number of page tables and cannot trigger the secure region adjustment. Also, as we will show in the macro benchmarks, no adjustments were triggered in practice. To evaluate the performance of the secure region adjustment under extreme stress, we further create 30,000 processes at the same time (larger will make the original kernel unstable), and measure the total time. These microbenchmarks are run not only on the `CFI` and `CFI+PTStore`, but also on `CFI+PTStore` without the secure region adjustment (`CFI+PTStore-Adj`), on which we initialize the secure region to 1GiB to provide enough memory for page tables to avoid the adjustments. We have used a debug version of PTStore to confirm that `CFI+PTStore` triggers adjustments and `CFI+PTStore-Adj` does not. The results in terms of relative overheads are 2.84%, 6.83% (+4.00%), and 3.77%

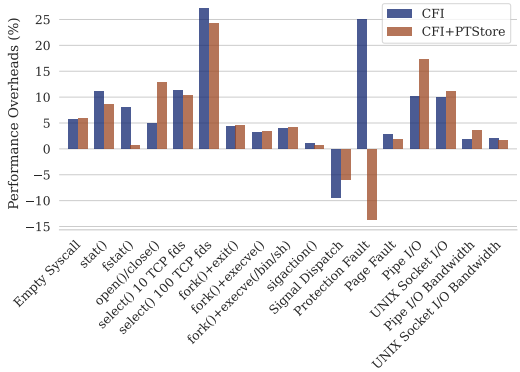

Fig. 4: Performance overheads of LMBench microbenchmarks.

TABLE III: Hardware resource cost of our prototype system PTStore when synthesised and mapped to an FPGA. $F_{target} = 90.000$MHz.

	RISC-V BOOM cores				Whole Systems					
	#LUT	%	#FF	%	#LUT	%	#FF	%	WSS (ns)	F_{max} (MHz)
without PTStore	55,367	–	37,327	–	71,633	–	57,151	–	0.033	90.269
with PTStore	55,875	+0.918	37,423	+0.258	72,081	+0.626	57,307	+0.273	0.136	91.116

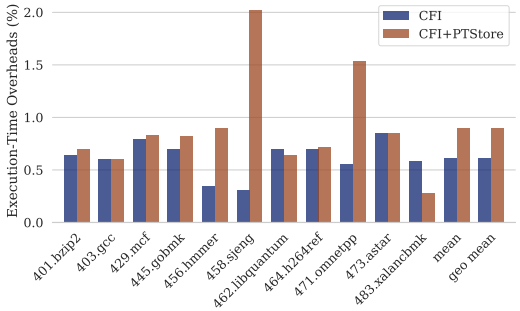

Fig. 5: Execution-time overheads of SPEC CINT2006 benchmarks.

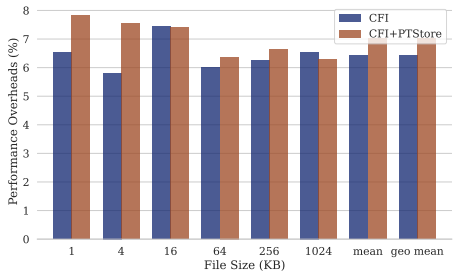

Fig. 6: Performance overheads of NGINX. 10,000 requests in total, 100 concurrent requests.

(+0.94%), respectively. The results of `CFI` and `CFI+PTStore-Adj` are similar to those of the `fork()` tests in LMBench, while that of `CFI+PTStore` is slightly higher.

Overall, the results show that, on average, PTStore does not introduce significant performance overheads for various system calls.

2) Macro Benchmarks: In line with prior work [2], [4], [12], [14], we use the SPEC CPU2006 benchmark suite, which is CPU intensive, as well as NGINX 1.20.1 benchmark and Redis 6.2.6 benchmark, which are kernel intensive, to evaluate the performance of the whole system. For the SPEC CPU2006, since the FPU is disabled, we only run the integral subset (i.e., SPEC CINT2006). Besides, *400.perlbench* is excluded due to compilation failure for the RISC-V ISA. All inputs are the *reference* workloads. The results of the relative performance overheads are presented in Figures 5, 6, and 7.

Since all the benchmarks complete successfully, we can confirm again that our modifications to the kernel are correct. In addition, we did not encounter any secure region adjustment, showing that the default 64 MiB secure region is sufficient in practice, which is consistent with prior work [18], [21].

The results show that the average performance overheads of PTStore together with the Clang CFI are <0.91% for CPU-bound SPEC CINT2006 benchmarks and <8.18% for kernel-bound NGINX and Redis benchmarks. Excluding the overheads of the Clang CFI, on average, PTStore only introduces <0.29% performance overheads for CPU-bound benchmarks and <0.86% for kernel-bound benchmarks. Besides, since the Clang CFI is a software solution, we expect much lower overall performance overheads when the CFI is assisted by hardware like ARM PA [22] or `ROLoad` [23]. ***To conclude, PTStore does not introduce significant overheads, and the whole system under protection can run almost as fast as the original one.***

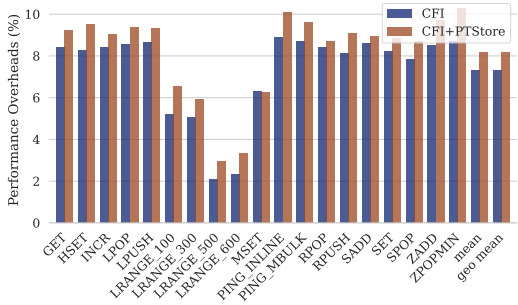

Fig. 7: Performance overheads of Redis. 100,000 requests for each test, 50 parallel connections.

E. Security Analysis

1) Basic Security Guarantees: The basic concepts of PTStore are similar to related work [4], [14] that differentiates all page-table manipulation code from normal code and permits only the former to access the page tables. Thus, the security guarantees of PTStore are at least the same as those of the prior work, which have been demonstrated thoroughly. Furthermore, PTStore can defeat PT-Injection attacks and the following four types of attacks, while prior solutions usually cannot.

2) Reuse Attacks: We also require the PTW to fetch page tables only from the secure region. This provides stronger security guarantees and prevents injected or reused data in normal memory from being fetched as page tables.

At the same time, out of context data in the secure region cannot be reused by the PTW as well. Actually, the data in the secure region are either page tables or tokens, and all fields in the tokens are pointers to the PCBs or the page tables. Since the addresses of the PCBs and the page tables are aligned to the size of pointers, i.e., 8B on the 64-bit ISA, all the 3 low bits of all fields in the tokens are zero. These fields are thus invalid page table entries since their present bits are unset. Besides, the tokens further ensure that a process can only use its root page table. Thus, attackers cannot reuse any data in the secure region.

3) Attacks on Allocator Metadata: A sophisticated attacker can tamper with the metadata of the allocators and force them to allocate new page tables from in-use pages in the secure region, overlapping with existing page tables. Although PTStore does not protect the allocators, it can defeat this attack by simply checking whether the newly allocated pages are all zeros, which ensures that the pages are actually free before allocating them to new page tables.

4) Attacks on VM Metadata: Attackers can also tamper with the metadata of VM in the kernel. For example, the attacker can tamper with the permission bits of VM areas, and then, the kernel will use the tampered data to compose malicious page table entries. Fortunately, we find that VM areas only hold the user-space VM information for processes. Thus, these structures do not affect the kernel address space, and they do not lower the protection of PTStore.

5) Attacks on TLB: TLB inconsistency caused by bugs may allow an attacker to exploit the stale (writable) permissions to tamper with previously mapped physical pages [16]. These vulnerable physical pages may further be allocated as page tables. In this case, all VM-based solutions may be bypassed. However, based on PMP, PTStore performs the secure region checks on physical addresses.

Thus, wherever VAs are mapped to, the secure region can only be accessed via PTStore-family instructions.

Overall, PTStore is secure against various attacks on page tables.

F. The Generality of PTStore

Although we only demonstrated that PTStore can effectively protect page tables, we believe that PTStore is general to isolate and protect other critical data, especially those in bare-metal applications. Examples include but are not limited to code pointers and critical MMIO registers like the control registers of watchdog timers.

VI. Related Work

There are several kinds of solutions proposed to protect page tables.

1) Randomization: The first kind of solution [4], [5] randomizes the locations of page tables in the VM space of kernels, and some of them also ensure that attackers cannot leak page table pointers. Similar to ASLR [6], these solutions also suffer from information-disclosure [4] or side-channel attacks [19], and have low entropy on 32-bit systems. Besides, the PTW does not limit the origin of the page tables, and attackers can hijack page table pointers to launch PT-Injection and PT-Reuse attacks.

2) Physical Isolation: The second kind of solution physically isolates page tables from other memory regions [2], [7], [8], [9]. They usually rely on hypervisors and introduce non-negligible performance overheads >5% [7], [10], [11], even without other needed mitigations (e.g., CFI) and accelerated by heavy hardware. Thus, they are unaffordable for resource-limited systems (e.g., IoT devices). Furthermore, the page tables of hypervisors need to be protected, too.

Some of them use HMACs, e.g., SipHash [24], to protect the page table pointers. These HMACs act the same role as our tokens. However, the HMACs require strong cryptographic algorithms, implemented in either software or hardware introducing performance or area overheads. Besides, they are not resilient to side-channel attacks [19].

3) Virtual Isolation: Other solutions also isolate page tables, but virtually [12], [13], [14], [15]. Some of them are hardware-assisted like PTStore, while they isolate page tables' every *virtual* page. Based on VM, they suffer from the chicken-and-egg problem (Section III-C2) and can hardly mitigate PT-Injection or PT-Reuse attacks.

Some others are implemented mainly in software [12], [13], [15]. They all leverage VM and provide a secure execution environment for page-table manipulation code on the same privilege level as the kernel. Among them, Apple's PPL uses extra *proprietary* hardware. However, they need trampolines to enter the secure execution environment and may flush the pipeline, and the security checks are done carefully in pure software. Thus, they will introduce non-negligible performance overheads and the security is hard to guarantee. Furthermore, these solutions require that the VM works properly, and thus are not resilient to the TLB inconsistency attack [16] (Section V-E5).

4) Secure Enclaves: Secure enclaves also isolate sensitive data from normal memory [9], [21], [25], [26]. For example, Penglai [21] proposes a similar PMP-based technique. However, Penglai checks mappings in an M-mode secure monitor every time the kernel modifies a page table and will introduce much more performance overheads. Furthermore, Penglai cannot dynamically adjust the secure region.

The main difference between secure enclaves and PTStore is the threat model: secure enclaves do not trust the kernel, while PTStore assumes the kernel is benign and helps the kernel to defend against vulnerabilities. This yields a simpler and more efficient design.

Overall, these existing solutions usually provide insufficient security guarantees, rely on heavy hardware, or introduce high overheads.

VII. Conclusion

OS kernels and many defense solutions all rely on the integrity of page tables. In this paper, we have proposed a lightweight solution, PTStore, to protect page tables against memory-corruption attacks. FPGA-based prototypes of PTStore reveal that PTStore is lightweight, which costs <0.92% hardware overheads and <0.86% performance overheads. Furthermore, we have explored the security guarantees of PTStore and showed that PTStore outperforms prior solutions.

References

[1] M. Abadi *et al.*, "Control-flow integrity," in *Proc. ACM CCS '05*, 2005.

[2] J. Criswell *et al.*, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *Proc. IEEE S&P '14*, 2014, pp. 292–307.

[3] X. Ge *et al.*, "Fine-grained control-flow integrity for kernel software," in *Proc. IEEE EuroS&P '16*, 2016, pp. 179–194.

[4] L. Davi *et al.*, "Pt-rand: Practical mitigation of data-only attacks against page tables," in *Proc. NDSS '17*, 2017.

[5] D. Weston and M. Miller. (2016) Windows 10 mitigation improvements. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf

[6] H. Shacham *et al.*, "On the effectiveness of address-space randomization," in *Proc. ACM CCS '04*, 2004, pp. 298–307.

[7] S. Proskurin *et al.*, "xmp: Selective memory protection for kernel and user space," in *Proc. IEEE S&P '20*, 2020, pp. 563–577.

[8] Intel. (2022) Hypervisor-managed linear address translation.

[9] ——. (2022) Intel(r) trust domain extensions. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf

[10] C. Dall *et al.*, "Arm virtualization: Performance and architectural implications," in *Proc. ISCA '16*, 2016, pp. 304–316.

[11] Z. Wang *et al.*, "Seimi: Efficient and secure smap-enabled intra-process memory isolation," in *Proc. IEEE S&P '20*, 2020, pp. 592–607.

[12] N. Dautenhahn *et al.*, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proc. ASPLOS '15*, pp. 191–206.

[13] A. M. Azab *et al.*, "Skee: A lightweight secure kernel-level execution environment for arm," in *Proc. NDSS '16*, 2016.

[14] T. Frassetto *et al.*, "IMIX: In-process memory isolation extension," in *Proc. USENIX Security '18*, Aug. 2018, pp. 83–97.

[15] A. Inc. (2021) Operating system integrity. [Online]. Available: https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/1/web/1

[16] B. Azad. (2020) The core of apple is ppl: Breaking the xnu kernel's kernel. [Online]. Available: https://googleprojectzero.blogspot.com/2020/07/the-core-of-apple-is-ppl-breaking-xnu.html

[17] A. Waterman *et al.*, "The risc-v instruction set manual, volume i: User-level isa, document version 20191213," *RISC-V Foundation*, Dec. 2019.

[18] X.-C. Wu *et al.*, "Protecting page tables from rowhammer attacks using monotonic pointers in dram true-cells," in *Proc. ASPLOS '19*, 2019.

[19] J. Ravichandran *et al.*, "Pacman: Attacking arm pointer authentication with speculative execution," in *Proc. ISCA '22*, 2022, pp. 685–698.

[20] Y. Liu *et al.*, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proc. ACM CCS '15*, 2015.

[21] E. Feng *et al.*, "Scalable memory protection in the PENGLAI enclave," in *Proc. USENIX OSDI '21*, 2021, pp. 275–294.

[22] H. Liljestrand *et al.*, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proc. USENIX Security '19*, pp. 177–194.

[23] W. Tan *et al.*, "Roload: Securing sensitive operations with pointee integrity," in *Proc. DAC '21*, 2021, pp. 307–312.

[24] J.-P. Aumasson *et al.*, "Siphash: A fast short-input prf," in *Proc. INDOCRYPT 2012*, 2012, pp. 489–508.

[25] Intel. (2022) Intel(r) software guard extensions (intel(r) sgx). [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

[26] D. Lee *et al.*, "Keystone: An open framework for architecting trusted execution environments," in *Proc. EuroSys '20*, 2020.