PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux

Juhee Kim
Department of Electrical and Computer Engineering
Seoul National University
kimjuhi96@snu.ac.kr

Jinbum Park
Samsung Research
jinb.park@samsung.com

Yoochan Lee
Department of Electrical and Computer Engineering
Seoul National University
yoochan10@snu.ac.kr

Chengyu Song
University of California, Riverside
csong@cs.ucr.edu

Taesoo Kim
Samsung Research and Georgia Institute of Technology
taesoo@gatech.edu

Byoungyoung Lee*
Department of Electrical and Computer Engineering
Seoul National University
byoungyoung@snu.ac.kr

ABSTRACT

Data-only attacks are emerging as a new threat to the security of modern operating systems. As a typical data-only attack, memory corruption attacks can compromise the integrity of kernel data, which effectively breaks the premises of access control systems. Unfortunately, the prevalence of memory corruption vulnerabilities allows attackers to exploit them and bypass access control mechanisms. Given the arbitrary memory access capability, attackers can overwrite access control policies or illegally access the kernel resources protected by the access control systems.

This paper presents PeTAL, a practical access control integrity solution against data-only attacks on the ARM-based Linux kernel. PeTAL is designed to ensure access control integrity by providing policy integrity and complete enforcement of access control systems. PeTAL first identifies kernel data used as access control policies and kernel data protected by access control policies, based on the user interfaces of the Linux kernel. Then, PeTAL leverages the ARM Pointer Authentication Code (PAC) and Memory Tagging Extension (MTE) to comprehensively protect the integrity of the identified kernel data and pointers. We implemented the prototype of PeTAL and evaluated the performance and the security impact of PeTAL on real AArch64 hardware with PAC and MTE support. Our evaluation results show that PeTAL can effectively thwart memory-corruption-based attacks on access control systems with reasonable performance overheads at most 4% on average in user applications, demonstrating its efficient prospects for kernel security.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**;

*Corresponding author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.

ACM ISBN 979-8-4007-0636-3/24/10…$15.00
https://doi.org/10.1145/3658644.3690184

ACM Reference Format:
Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. 2024. PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3658644.3690184

1 INTRODUCTION

Access control is a cornerstone of modern operating systems. Following the principle of least privilege, the policy of access control is designed to ensure that user processes are granted minimal access permissions to perform a certain task. With the access control enforcements, user processes can read or write kernel resources only if the policy permits it. Modern operating systems implement access control systems to practice the best least privilege while accommodating unique engineering challenges per operating system. Particularly in the Linux kernel, which is the major focus of this paper, several access control systems are employed, including discretionary access control (DAC), Capabilities, and Linux Security Modules (LSMs), each of which has a unique protection and isolation scope.

A secure and robust access control systems should ensure the following two properties: i) policy integrity and ii) complete enforcement. Policy integrity requires that access control policy is not corrupted, and complete enforcement requires that every user access to kernel resource is protected by access control enforcement. We refer to these two properties together as *access control integrity*.

Unfortunately, attackers often compromise access control integrity, particularly through memory corruption attacks [36]. First, policy can be illegally updated, allowing attackers to escalate their permissions or set the access permissions of kernel resources to their favor. Second, complete enforcement can fail when attackers access kernel resources without undergoing necessary access control enforcement. Both violations could result in unauthorized access to privileged kernel resources, including tasks, files, and security-critical configurations.

Several mitigations have been proposed and integrated into the Linux kernel to thwart memory corruption attacks, but those provide limited protection against access control attacks. Control-flow

integrity (CFI) ensures that runtime control-flows adhere to static control-flow graphs [20, 24, 33, 37, 56, 81]. By enforcing legitimate control-flow, CFI thwarts attacks that illegally jump to policy update code or skip access control enforcements. Nevertheless, CFI does not guarantee legitimate data-flow, leaving open doors for data-only attacks that illegally access access control policies or kernel resources. Recent kernel attacks (e.g., `Bad Binder` [57] and `DirtyCred` [59]) have demonstrated the potentials of data-only attacks, even in the presence of CFI. These findings underscore that CFI, while crucial, is not a silver bullet for kernel security.

Data-flow integrity (DFI) is another mitigation technique that enforces legitimate data-flows, e.g., a memory write within a device driver should not overwrite user credentials [4, 17, 55, 63, 77]. Kenali [77] is a DFI solution that protects data used in permission checks, thus ensuring policy integrity. However, Kenali does not guarantee complete enforcement; Attackers can still exploit memory corruptions to illegally read or write kernel resources, bypassing access control enforcement. To the best of our knowledge, no solution fully guarantees complete access control integrity in the Linux kernel against data-only attacks.

This paper proposes PeTAL[1], a practical DFI solution to provide access control integrity in the Linux kernel. PeTAL exhibits a reasonable performance overhead, and provide good compatibility with the Linux kernel. Notably, PeTAL runs with the state-of-the-art CFI techniques (e.g., PAL [89] and Clang/LLVM CFI [20]), ensuring access control integrity across both control- and data-flows. PeTAL can be summarized with the following key design features. First, comprehensively analyzing control-flow and data-only attacks against access control integrity, PeTAL identifies the protection scope of DFI leveraging well-defined user interfaces. Second, PeTAL designs a robust and effective framework to enforce DFI rules on the identified kernel metadata, orchestrating two recent ARM hardware extensions: ARMv8.3 Pointer Authentication Code (PAC) [67] and ARMv8.5 Memory Tag Extension (MTE) [10].

More specifically, the novelty of PeTAL lies in its ability to offer access control integrity. PeTAL thwart real-world access control attacks performing privilege escalation, such as `DirtyCred` [59] and `Bad Binder` [57]. PeTAL further prevent attacks illegally accessing critical kernel resources, such as `modprobe_path` and `randomize_va_space`, which were not protected by any existing solutions.

The prototype of PeTAL is implemented on Android Linux kernel version 5.10.136, and evaluated on a real MTE/PAC-supported device, Samsung Galaxy S22 [71]. [2] Our evaluation results on the real device show that PeTAL provides comprehensive protection against both control-flow attack and data-only attacks with a reasonable performance overhead: 32% for kernel workloads and 4% for user space workloads. When compared to Kenali [77], a state-of-the-art DFI on the Linux kernel, PeTAL shows a 35%-51% performance gain on kernel benchmark, highlighting the efficiency and practical effectiveness of PeTAL. According to our security analysis, PeTAL can prevent majority of the memory corruption attack vectors, preventing attacks on kernel access control integrity.

[1] PeTAL is open-sourced at https://github.com/compsec-snu/petal

[2] This device is commercially available with PAC support, but MTE is disabled. Our evaluations were conducted on an MTE-enabled device provided by Samsung Electronics.

2 BACKGROUND

PeTAL leverages two ARM hardware features, Memory Tagging Extension (MTE) and Pointer Authentication Code (PAC). In this section, we provide a brief overview of these features.

Memory Tagging Extension. MTE is a hardware-accelerated memory tagging feature introduced in ARMv8.5. MTE assigns a 4-bit tag per 16-byte memory region and stores the tag in the top byte of a pointer. On memory access, MTE checks if the pointer's tag matches the memory region's tag and raises an exception when they do not match. In synchronous mode, the exception is raised immediately at the memory access; in asynchronous mode, the exception is logged and raised later. MTE also provides a match-all tag enabled by setting the `TCMA` bits in `TCR_EL1` register [48]. If `TCMA1` is set, memory access in the kernel space is not tag-checked when the pointer contains the match-all tag (i.e., `0xf`).

Pointer Authentication Code. PAC is a hardware-based pointer protection introduced in ARMv8.3-A. It reserves the top bits of a pointer for a pointer signature called `PAC`. `PAC` is a cryptographic signature to detect pointer corruption. ARM provides two sets of instructions, `PAC*` and `AUT*`, to sign and authenticate pointers. `PAC*` instructions create `PAC` for a pointer by using the original pointer value, a 64-bit modifier (i.e., a *context*), and a 128-bit key, and store the `PAC` at the top bits of the pointer. `AUT*` instructions authenticate the pointer by computing the `PAC` using the pointer value, the context, and the key. The computed `PAC` is compared with the `PAC` stored at the top bits of the pointer ([54:39]). If the `PAC` is valid, `AUT*` instructions return the pointer in which the `PAC` bits are stripped out. Otherwise, an exception is raised, detecting pointer corruption. Before ARMv8.6-A FPAC extension, an error bit is set on the pointer top bit on authentication failure, which triggers a translation failure when the pointer is dereferenced. With FPAC extension, `AUT*` instructions immediately raise an authentication failure exception.

3 ACCESS CONTROL SYSTEMS AND ATTACKS

Following the principle of least privilege, Linux kernel implements access control systems that restrict unprivileged user's access to privileged kernel resources. Therefore, a primary goal of unprivileged attackers is bypassing the access control system so as to access resources that were previously not accessible. This section first introduces the Linux kernel access control systems (§3.1) and then describes the real-world access control bypassing attacks even when the state-of-the-art mitigation techniques are in place (§3.2).

3.1 Access Control Systems

An access control system is responsible for protecting kernel resources when serving user requests. In general, it is structured with three components: policy, mechanism, and resources [74].

Policy is access control rules that determine who can access which resources and how. For instance, policies are often defined by *credentials*, which are kernel data specifying the privileges associated with a user process or a kernel resource. Other examples of policies include resource's runtime states (e.g., bpf register state).

```
1  long do_sys_open(const char *filename, int flags) {
2   int error, fd;
3   struct file *file;
4   struct inode *inode;
5   fd = get_unused_fd(flags);
6
7   // look up the inode from inode_hashtable
8   inode = lookup_open(filename, file, flags);
9
10  // enforce DAC mechanisms -> acl_permission_check()
11  if (!may_open(inode, flags))
12    goto out;
13
14  if (!vfs_open(inode, file))
15   goto out;
16
17  // update file descriptor with the user input
18  file->f_mode = flags | FMODE_OPENED;
19
20  // install the file descriptor
21  current->files->fdt->fd[fd] = file;
22  return fd;
23
24 out:
25   return error;
26 }
27 int acl_permission_check(struct inode *inode, int mask) {
28  unsigned int mode = inode->i_mode;
29  if (current->cred->uid == inode->i_uid)
30    mode >>= 6;
31  if ((mask & ~mode &
32        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
33    return 0;      // success
34  return -EACCES;  // fail
35 }
```

user input | policy | enforcement | resource

Figure 1: A simplified kernel implementation of system call open.

Mechanism is the enforcement of access control policies through *permission checks*. Permission checks are kernel code that determines whether to allow user requests based on access control policies (i.e., credentials). If the permission check is successful, the kernel accesses the requested kernel resource on behalf of the user. Otherwise, the kernel declines the request and returns an error code (e.g., `-EPERM` or `-EACCES`).

Resources are kernel data protected by access control systems from unprivileged users. Resources include system configurations (e.g., executable paths), credentials (e.g., `UID`), and files (e.g., `/etc/shadow`). Any kernel code that reads or writes these resources on behalf of a user must be preceded by access control mechanisms. This ensures that the resources can be accessed by the user request approved by the policy.

It is worth noting that the kernel manages a subset of policies as resources, allowing privileged users to configure the policies. For instance, while a user's `UID` and the file owner's `UID` determine the policy of Discretionary Access Control (DAC), they are also kernel resources that can be updated by privileged users through system calls (e.g., `setuid` and `chown`, respectively). To ensure that only authorized users can update these policy data, the kernel introduces yet another layer of access control; Capability [50] of the current user is checked before updating the policies. [3]

[3] User's `UID` and the file's owner can be updated if the current user's permitted capability set includes `CAP_SETUID` and `CAP_CHOWN`, respectively. Capabilities ensure that its policy (i.e., the user's permitted capability set) cannot be escalated once the policy is set.

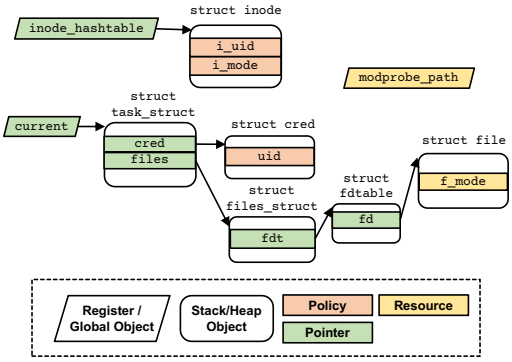


Figure 2: Policies and Resources in open.

Example: open(). Figure 1 is a simplified implementation of the open syscall that creates a file descriptor from a file name and access mode. Figure 2 illustrates the policies, resources, and pointers in this example. A file descriptor is a kernel resource that represents an opened file with an access mode, which can be accordingly used by other syscalls (e.g., `read`, `write`). To prevent an unauthorized file access, the kernel enforces DAC on file descriptors.

At line 1 of Figure 2, open takes a file name (i.e., `filename`) and access mode (i.e., `flags`) as arguments (Figure 1). At line 8, kernel retrieves the file's inode (i.e., `struct inode`) located by the file name from `inode_hashtable`. At line 11, the kernel enforces DAC checks in `may_open()`, where the actual checks are done in `acl_permission_check()`. DAC utilizes the following data as policies: (i) the user process's `UID` (i.e., `current->cred->uid`), (ii) the file owner's `UID` (i.e., `inode->i_uid`), and (iii) the access mode of the file owner and others (i.e., `inode->i_mode`). At line 29, `acl_permission_check` compares the user process's `UID` with the file owner's `UID`. If the UIDs match, the access mode of the file owner is used (`mode >>= 6`); otherwise, the access mode of others is used. At line 31, the file access mode (i.e., `mode`) is checked to see if it includes the access mode requested by the user (i.e., `mask`). If the check is successful, the kernel allows the user request and update the file descriptor accordingly. At line 18, the access mode of the file descriptor (i.e., `file->f_mode`) is initialized with the user-provided data (i.e., `flags`). At line 21, the file descriptor is installed at the user's file descriptor table (i.e., `current->files->fdt`). If the check is unsuccessful, `acl_permission_check` returns an error code `-EACCES` (i.e., Permission Declined), and the kernel declines the request.

3.2 Attacks on Access Control Systems

Access Control Integrity. Before we discuss the attacks on access control systems, we first define *access control integrity*, a property that access control systems require to be robust against memory corruption. Access control integrity is defined as the following two properties.

Policy integrity. The access control policies should only be updated through a legitimate control and data-flow. Ensuring policy integrity, the attacker cannot corrupt the policy of the user process or the kernel resource.

Complete enforcement. The access control mechanism should always be enforced before a user accesses kernel resources. With complete enforcement, every access to the kernel resources is preceded by permission checks.

Unfortunately, attackers often find and exploit kernel vulnerabilities and violates access control integrity. In the following, we explore two attack types, control-flow attacks and data-only attacks.

3.2.1 Control Flow Attacks. Control flow attacks divert the control flow by corrupting the return address or function pointers. Due to the adoption of Control-Flow Integrity (CFI) [2, 20, 33, 65, 81, 89] across major operating systems (including Linux [28], Android [6], Microsoft Windows [56], Apple XNU [8], etc), the effectiveness of control-flow attacks is severely limited [18, 41, 49, 57, 88].

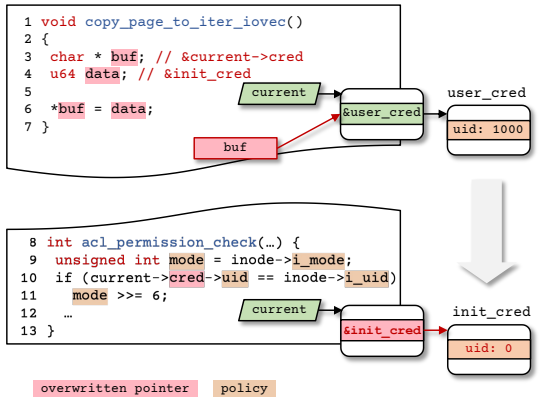
Breaking Policy Integrity. Policy integrity can be violated by illegally jumping to a write instruction. With Return-Oriented Programming (ROP) gadgets, the attacker can manipulate the source and destination of the write instruction, thereby illegally updating policies. For instance, the attacker can modify a user credential object (i.e., `struct cred`) with the highest privilege (i.e., `root`).

Breaking Complete Enforcement. Control flow attacks can break complete enforcement by illegally jumping to the kernel code that accesses kernel resources, thus bypassing the necessary enforcement mechanisms. For instance, attackers can skip the DAC enforcement by directly jumping to the code that initializes the file descriptor with the desired file and access modes.

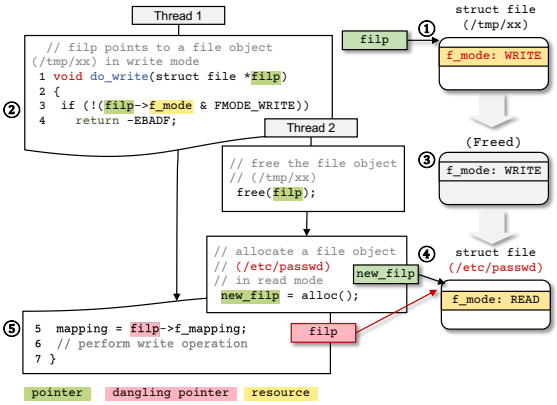
3.2.2 Data-Only Attacks. Data-only attacks [41, 42] pose a significant threat to the Linux kernel security. Unlike control flow attacks, data-only attacks compromise non-control data (e.g., numerical data or data pointers) to achieve their malicious goal, bypassing CFI mitigation techniques. The most alarming aspect of data-only attacks is that there is currently no effective protection against them in the Linux kernel. Recent incidents of data-only attacks [49, 57–59] have emphasized the severity of this problem. In the following, we take a closer look into the data-only attacks that break access control integrity.

Breaking Policy Integrity. Policy integrity is compromised when attackers manipulate policies with arbitrary write vulnerabilities. In particular, attackers can forge user credentials to escalate their privilege, or corrupt the credentials of kernel resource to downgrade the necessary privilege to access the resource. For instance, simply corrupting `UID` to 0 (i.e., `root`) would allow illegally bypassing DAC on root-only files. Similarly, attackers can modify the file credentials; change the owner `UID` of a file to illegally match with the attacker's `UID`, or change the access mode of a file to `0x666` (i.e., readable and writable by everyone). This would allow bypassing DAC when opening the file with the corrupted policies.

To clearly illustrate, we show a real-world example of policy integrity violation using `Bad Binder` [57], which illegally updates user credentials (Figure 3a). `Bad Binder` offers an arbitrary write primitive (line 6), where both destination and source operands can be crafted by the attacker via vulnerable binder `ioctl` calls: i) destination (i.e., `buf`) is manipulated to be a pointer of the current user credential pointer (i.e., `&current->cred`), where the current user credential object is unprivileged. and ii) source (i.e., `data`) is



(a) Breaking policy integrity with `Bad Binder` (CVE-2019-2215).



(b) Breaking complete enforcement with `DirtyCred` (CVE-2021-4154).

Figure 3: Examples of data-only privileged escalation attacks

manipulated to be a pointer of the init credential object pointer (i.e., `&init_cred`), where the init credential object has the highest privilege. When the write instruction is executed at line 6, the current user credential will be corrupted and point to the init credential object, effectively replacing the unprivileged credential to the highest privileged credential (i.e., updating `UID` to 0). Therefore, this attack breaks policy integrity (i.e., user credential), allowing attackers to bypass DAC and overwrite root-only files.

Breaking Complete Enforcement. Complete enforcement is violated when kernel resources are accessed via data-only attacks without undergoing the necessary access control mechanisms. An example of this attack is `DirtyCred` [49], which is a recent data-only attack on file descriptors, the kernel resources protected by DAC (Figure 3b). In thread 1, the attacker opens a writable victim file (e.g., `/tmp/xx`) in a write mode, successfully passing the DAC enforcement and creating a file descriptor with `WRITE` mode (①). As the file descriptor is opened with `WRITE` mode, the kernel allows write (②). Before the kernel proceeds with the write operation, in thread 2, the attacker frees the file descriptor (③) and allocates a new one for a read-only target file (e.g., `/etc/passwd`) with read mode, triggering a race condition (④). When thread 1 is resumed, use-after-free is triggered with the dangling file descriptor pointer (i.e., `filp`), processing the write operation with the read-only target

file (⑤). As a result, the attacker corrupts the file descriptor without going through DAC enforcement, breaking complete enforcement and gaining write access to the read-only target file. Similarly, attackers with arbitrary write capability can also corrupt the current file descriptor table pointer (i.e., `current->files->fdt`) or file descriptor pointers registered in the table (i.e., `fdt->fd[fd]`) to obtain arbitrary file descriptors, allowing them to bypass DAC.

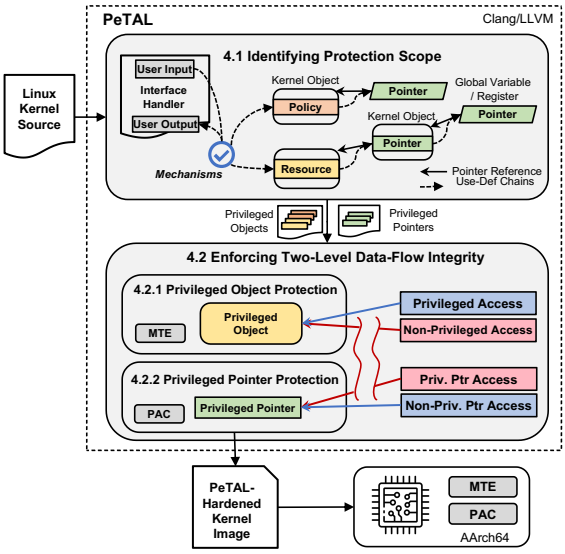
Another example of complete enforcement violation is corrupting or leaking kernel resources that are accessible via pseudo file systems, such as `procfs` and `sysfs`. Pseudo file systems allow users to read or write kernel resources leveraging existing the Linux file system and its access control (i.e., DAC). For instance, `modprobe_path` [69] is a kernel resource writable through a `sysctl` file (i.e., `/proc/sys/kernel/modprobe_path`). This variable contains the path to an executable binary file that runs with the `root` privilege. Consequently, the DAC policies associated with the `sysctl` file are set up to permit only the `root` to write. However, leveraging data-only attacks, a non-root attacker can directly overwrite `modprobe_path` variable with the path of an attacker-controlled binary. This would allow attackers to bypass DAC and control `modprobe_path`, allowing them arbitrary code execution capability with `root` privilege. `modprobe_path` exploit is widely used in real-world privilege escalation attacks [69]. However, to the best of our knowledge, there is no general protection solution that provides complete enforcement for access control systems.

4 THREAT MODEL

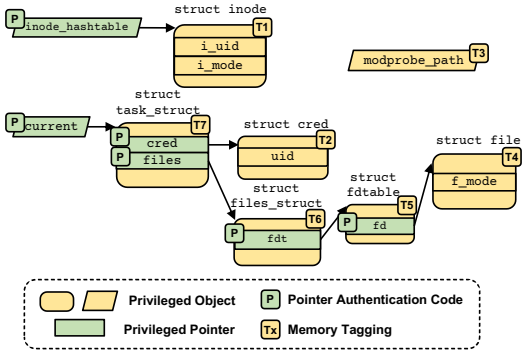
PeTAL is designed for a kernel developer who desires to protect the kernel against attacks violating access control integrity. Our threat model is similar to those used in privilege escalation mitigation [66, 77], considering an attacker with the lowest user privilege in the system who attempts to illegally gain higher user privilege (e.g., `root`). Our focus is more specific to access control, considering an attacker to exploit memory corruption vulnerabilities, obtain arbitrary read and write primitives, and violate access control integrity, either policy integrity or complete enforcement.

We assume that the Linux kernel runs with state-of-the-art self-protection techniques, including Control-Flow Integrity (CFI) [2, 6, 20, 33, 65, 81, 89], Kernel Address Space Layout Randomization (KASLR) [27], and NX/SMAP/SMEP [1, 22]. We assume CFI would prevent all control-flow attacks, so we only focus on preventing data-only attacks. We assume the hardware is the trusted computing base thus exploits against hardware vulnerabilities such as micro-architectural attacks [45, 51] are out-of-scope.

PeTAL operates on recent ARM processors with MTE and PAC support. PeTAL orchestrates MTE and PAC to protect kernel data and pointers. Both extensions have known security issues that may allow bypass attacks. We assume the kernel source code does not reveal the kernel MTE random tags, thus the attacker does not have any MTE oracle. PAC key leakage [90] is mitigated when used with PAL [89], a PAC-based CFI solution that ensures that PAC key is not leaked to the attacker. PAC replay/reuse attacks [47] are mitigated by binding PAC to the containing object (§5.2.2). We do not explicitly mitigate brute-force attacks [13], since the probability of the success is low, and the detection directly panics the system, preventing further exploitation (§7.3). PeTAL does not aim to prevent all privilege escalation attacks, which is infeasible considering the various attack vectors (e.g, malicious GPU page [60] and BPF/eBPF program [43]). However, PeTAL can be integrated with specialized defenses, such as sensitive page protections [9, 72, 80] and BPF-hardening techniques [43, 79].



(a) PeTAL workflow overview



(b) Example of privileged objects and privileged pointers in DAC

Figure 4: Design of PeTAL

5 ENFORCING ACCESS CONTROL INTEGRITY

The design goal of PeTAL is to guarantee *access control integrity* to the Linux kernel, which ensures policy integrity and complete enforcement (§3.2). PeTAL accomplishes this by enforcing data-flow integrity (DFI) on access control policies and resources. PeTAL focuses on addressing two main challenges: (i) how to find the optimal protection scope, and (ii) how to provide effective and efficient data flow enforcement on the identified kernel data. PeTAL addresses these challenges by (i) Linux kernel's user interface analysis to find access control policies and resources, and (ii) enforcing two-level DFI on the protection scope.

Figure 4a illustrates the overall design of PeTAL, where it takes Linux kernel source code as input and produces a PeTAL-hardened kernel binary. In the first phase, PeTAL performs an offline static

analysis to identify the protection scope (i.e., policy and resources) (§5.1). Once identified the protection scope, the second phase enforces DFI on the protection scope, leveraging ARM's MTE and PAC extensions (§5.2). Figure 4b shows how the policies and resources in the open example are protected with MTE and PAC (§3.1).

5.1 Identifying Protection Scope

To ensure access control integrity, PETAL must establish the protection scope. In the following, we outline the definition of the protection scope for access control integrity (§5.1.1) and introduce a user interface analysis for identifying the protection scope (§5.1.2).

5.1.1 Protection Scope for Access Control Integrity. Access control integrity requires two security properties: (i) policy integrity and (ii) complete enforcement. Data-only attacks can break both properties by corrupting either access control policies or resources, even if the CFI technique is deployed (§3.2). To guarantee access control integrity, PETAL enforces data-flow integrity (DFI) on two types of metadata: access control policies and kernel resources, ensuring policy integrity and complete enforcement, respectively.

Policy integrity requires that every write to policy should only be performed by legitimate write instructions (i.e., instructions that are intended to update the policy). It is worth noting that CFI alone does not ensure policy integrity. Even if CFI is employed, data-only attacks can still illegally overwrite the policy data by manipulating operands of a vulnerable write instruction [18, 41, 49]. Thus, enforcing DFI on policy with CFI, PETAL guarantees that policies are only updated through a legitimate write instruction.

Complete enforcement requires that every read and write operation to kernel resources should be preceded by access control enforcement. Similar to the case of ensuring policy integrity, CFI alone does not guarantee complete enforcement. Data-only attacks can directly read or write kernel resources through non-legitimate instructions (i.e., instructions that are not supposed to read or write kernel resources), thereby bypassing the code enforcing the access control (e.g., bypassing the permission checks). By enforcing DFI on resources in combination with CFI, PETAL guarantees that resources can only be accessed after the access control enforcement.

Limitations of Previous Works. Determining the attack surface of data-only attacks is a non-trivial task, as discussed in previous works [46, 77, 87]. Under-approximation could miss critical data and allow other attack vectors. Over-approximation, on the other hand, could introduce runtime performance overheads protecting unnecessary data. Approaches based on manual specification [66, 75, 78, 82] suffer from false negatives and scalability, largely due to the huge complexity of the kernel. Relying on heuristic rules to identify vulnerable data in the context of memory corruption [3] would not thwart all memory corruptions, failing to offer robust protection. Approaches based on permission check branches, such as Kenali [77] find access control policies, thereby ensuring policy integrity. However, they fall short in finding kernel resources protected by access control enforcement, such as `modprobe_path`, and thus cannot provide complete enforcement (Figure 3b).

5.1.2 User Interface Analysis. To identify access control policies and resources, PETAL leverages the Linux kernel's well-defined user interfaces, namely system calls and pseudo file systems. The key idea behind this approach is that the Linux kernel provides hundreds of user interfaces by system calls and pseudo file systems, each of which allows users to access kernel resources. To provide correct and complete protection, the kernel enforces access controls to these interfaces, allowing privileged users to access resources and denying unprivileged users. Following describes how PETAL identifies access control resources and then extends to finding access control policies.

Resources. According to the definition (§3.1), resources have the following two properties: (i) Resources can either be written or read by users; and (ii) Before writing or reading resources, an access control mechanism must be enforced. To find resources meeting these two properties, PETAL performs an analysis in two phases.

The first phase identifies kernel resources by a static taint analysis that tracks the input and output of the user interfaces based on use-def chains [64]. Then, it returns kernel metadata (i.e., struct fields and global variables) that users can either read or write with the user interface. The metadata *written* by users are collected by tracking the use-def chains in forward, starting from the user interface input, until identifying all kernel metadata where the tracked data is stored. The metadata *read* by users are collected by tracking the use-def chains in backward from the user interface output, until finding all kernel metadata from which the tracked data is loaded.

The second phase applies a constraint ensuring at least one access control enforcement is applied when accessing kernel metadata through the user interfaces. As suggested in previous works [77, 91], PETAL finds permission check branches that returns permission-related error codes (i.e., `-EPERM`, `-EACCES`, and `EROFS`). These checks are considered as the access control enforcement, since they allow or deny the user's access to the resources. Therefore, PETAL identifies the kernel metadata that is updated or retrieved after at least one successful permission check. If a successful permission check is found in the control flow from the interface entry to the kernel metadata access, PETAL regards the identified kernel metadata as resource. Since the error codes indicate the failure of permission checks across the Linux kernel (§3.1), this analysis is agnostic to specific access control mechanisms. Therefore, PETAL can effectively identify the resources protected by various access control mechanisms, including Capabilities, DAC, and LSMs. We provide concrete examples of the identified resources in the evaluation (§7.1).

In addition to the two-phased analysis, PETAL further ensures complete enforcement against pointer corruption. Thus, PETAL collects the chain of pointers given the identified resources by tracking the use-def chains in backward, starting from the input data, until finding all the pointers from which the tracked data is loaded. The analysis is recursively performed to collect the chain of pointers and terminates when all the tracked pointers reach global variables or special registers (e.g., `SP_EL0`).

Access Control Policies. According to the definition (§3.1), access control policy determines the permission check results, deciding if a user's request to access resources should be granted or denied. To identify the access control policies, PETAL first collects the permission check branches that guard resources, and extracts the kernel metadata that determines the branch results. These permission check branches are drawn from the second phase of the analysis for finding resources. From the collected branches, PETAL

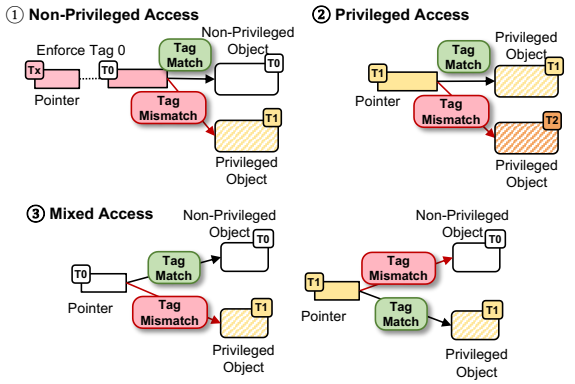


Figure 5: Privileged object protection

performs a static taint analysis to find the kernel metadata used as the branch predicate. This analysis tracks the permission check branch predicates by following the use-def chains in backwards, to find the kernel metadata from which the tracked data is loaded. The resulting kernel metadata are identified as access control policies. PeTAL further ensures policy integrity against pointer corruption by recursively tracking the pointers to the policies, in the same way the pointers to resources are collected.

5.2 Enforcing Two-Level Data-Flow Integrity

Based on the identified protection scope, PeTAL enforces selective data flow integrity (DFI) to protect access control integrity. Defining a safe data-flow for DFI is challenging due to the dynamic creation and destruction of kernel objects at runtime, as well as their interconnections through pointers. Moreover, it is crucial to maintain low performance overhead and high compatibility with the existing kernel code. To address these challenges, PeTAL designs a two-level DFI enforcement: *privileged object protection* and *privileged pointer protection*. PeTAL orchestrates two different hardware-based security features, ARM's Memory Tagging Extension (MTE) [10] and Pointer Authentication Code (PAC) [67] to provide robust security guarantees while optimizing the performance overhead.

The first layer is MTE-based protection on *privileged objects* (i.e., kernel objects containing access control policy, resource data or pointers to such data) (§5.2.1), and the second layer is PAC-based protection on *privileged pointers* (i.e., pointers to policy or resource) (§5.2.2). Each layer distinguishes privileged and unprivileged access and enforces data-flow integrity accordingly. Both layers are fully compatible with existing control-flow integrity (CFI) techniques, including Clang/LLVM CFI [20] and PAC-based CFI [52, 89]. Figure 4b provides a high-level overview of PeTAL, protecting the privileged objects and pointers in the `open` example.

5.2.1 Privileged Object Protection. The first layer aims to protect the integrity of access control policy and resource data. To achieve this, PeTAL isolates kernel objects holding policy and resource data with ARM's Memory Tagging Extension (MTE) [10].

Privileged Object. PeTAL enforces MTE-based DFI on *privileged objects*, the kernel objects storing access control policy, resource, and the pointers to them. All objects appeared in the `open` example are considered as privileged objects (e.g., `struct file`, `struct cred`, and `struct task_struct`). *Non-privileged objects* are the rest of the kernel objects.

Enforcing Tag Match. PeTAL assigns a unique random MTE tag to privileged objects and the default tag 0 to non-privileged objects. When a privileged object is allocated, a *non-zero* random tag (ranging from 0x1 to 0xe) is assigned to the object and this tag is stored at the top bits of the pointer. The tag is propagated with the pointer, and checked at the time of memory access to ensure that the pointer is pointing to a valid object. When a privileged object is freed, its tag is cleared by assigning tag 0 to the freed memory. When a non-privileged object is allocated, the object is not tagged, having the default tag 0, and the tag 0 is stored at the pointer top bits. When freed, the object is not tagged as well, and the freed memory has the default tag 0. By assigning a random tag only to privileged objects, PeTAL isolates them from non-privileged objects and further separates privileged objects from each other.

To further isolate privileged objects from non-privileged objects, PeTAL classifies memory access instructions into three types: *Non-privileged access*, *Privileged access*, and *Mixed access*, and enforces tag checks accordingly (Figure 5). Non-privileged access is the memory access instruction accessing non-privileged objects. For non-privileged access, PeTAL set the pointer tag to 0, restricting the access to non-privileged objects. This causes a tag mismatch if the instruction attempts to access privileged objects (Figure 5-①).

Privileged access is the memory access instruction accessing privileged objects (§5.2.2). In this case, PeTAL checks if the pointer tag matches with the object tag (Figure 5-②).

Mixed access is the memory access instruction accessing both privileged and non-privileged objects. For mixed access, enforcing tag 0 to the pointer can cause compatibility issues, as the pointer may point to privileged objects tagged with a non-zero tag. Therefore, PeTAL does not enforce tag 0 to the pointer, and checks if the pointer tag matches with the object tag, as in the case of privileged access (Figure 5-③). Mixed access can potentially introduce security risks, such that a corrupted non-privileged pointer accesses privileged objects. However, as we discuss in §7.2, the probability of this risk is significantly low when combined with the privileged pointer protection in §5.2.2.

Example of Privileged Object Protection. In the `open` example (Figure 4b), `struct cred` and `struct inode` are privileged objects because they contain access control policies (e.g., user and file credentials). Similarly, `struct file` and `modprobe_path` are privileged objects because they contain resources (e.g., access modes of a file descriptor and root-only writable variable). `struct task_struct`, `struct files_struct`, and `struct fdtable` are privileged objects since they contain pointers to privileged objects. PeTAL assigns a random non-zero tag to each privileged object and corresponding pointer. When the pointer is dereferenced, the tag is matched with the object tag, mitigating data-only attacks on policy and resources. In `DirtyCred` attack (Figure 3b), use-after-free access on `struct file` object would cause a tag mismatch when the pointer is dereferenced. The tag mismatch occurs when accessing i) freed area tagged with 0, ii) a newly allocated non-privileged object tagged with 0, or iii) a newly allocated privileged object (e.g., `struct file`) tagged with a different random tag.

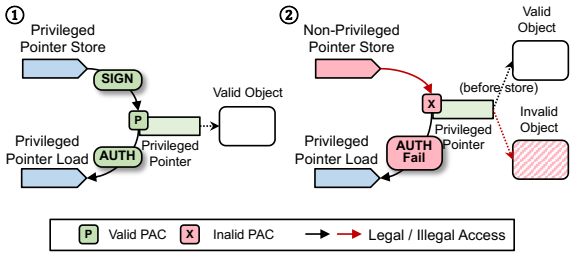


Figure 6: Privileged pointer protection

5.2.2 Privileged Pointer Protection. As the second layer, PETAL enforces data-flow integrity regarding the pointers in the protection scope, including the integrity of the MTE tag in the pointers. To this end, PETAL leverages ARM's Pointer Authentication Code (PAC) to sign and authenticate the pointers (Figure 6). This layer is fully compatible with CFI techniques that apply PAC on control pointers (i.e., function pointers and return addresses) [89], since the protection of PETAL does not overlap with that of CFI (i.e., PETAL protects data pointers with PAC).

Privileged Pointer. PETAL enforces data-flow integrity on *privileged pointers*, which are the pointers pointing to privileged objects. As described in §5.2.1, privileged objects contain access control policy and resource, or pointers to other privileged pointers (e.g., `struct task_struct*`, `struct cred*`, `struct file*`). We call the remaining pointers *non-privileged pointers*.

Enforcing Pointer Integrity. PETAL uses Pointer Authentication Code (PAC) to preserve the integrity of privileged pointers (Figure 6). All privileged pointers are PAC-signed at store and authenticated at load, preventing attackers from injecting pointers or corrupting existing ones. PETAL additionally uses the 64-bit tagged address of the containing object as the PAC context. This context scheme offers two security features. First, the context is spatially unique—i.e., two objects placed at different memory addresses have different PAC contexts, because the context includes the object address. Second, the context is temporally unique—i.e., two objects placed at the same memory address but allocated at the different time have different PAC contexts, because the context includes the MTE tag embedded in the 64-bit address. Thus, the PAC context of PETAL binds the privileged pointer to the containing object at the specific memory address and allocation time, which offers unique security advantages to PETAL to prevent PAC reuse attacks.

Memory access instructions that store or load privileged pointers (i.e., *privileged pointer access*) are modified to enforce pointer integrity. When storing a privileged pointer, PETAL signs the pointer with the stored address. When loading a privileged pointer, the `PAC` in the pointer is authenticated with the load address. If a privileged pointer contains a correct `PAC` (i.e., the `PAC` is generated with the correct context), the authentication succeeds and the `PAC` is removed from the pointer (Figure 6-①). If a privileged pointer contains a corrupted `PAC`, either due to an invalid target address (e.g., has been corrupted by non-privileged pointer access), or due to context mismatch (e.g., not written with the valid context), the authentication fails, and PETAL detects the pointer corruption (Figure 6-②). Other memory instructions (i.e., *non-privileged pointer access*) do not enforce PAC checks, due to performance overheads. Potential corruption of a non-privileged pointer is mitigated by enforcing MTE tag 0 to the pointer when dereferenced (Figure 5).

Attackers cannot corrupt privileged pointers through PAC reuse attacks [12, 67]. Spatial reuse of PAC, such that an existing `PAC`-signed pointer is copied to a different memory location, illegally constructing a new `PAC`-signed pointer, is prevented. Since `PAC` is bound to the original memory location, authentication fails when the new `PAC`-signed pointer is loaded from the different memory location. Temporal reuse of PAC, signing and authenticating the pointer in temporally distinct contexts, is also prevented. For instance, a privileged pointer is signed by a valid context (i.e., a valid pointer) and stored before, but a use-after-free vulnerability can illegally load the pointer and authenticate it with a wrong context (i.e., a dangling pointer). Since the objects containing privileged pointers randomly tagged privileged objects, the context would contain a random non-zero tag. If the MTE tag of the dangling pointer mismatches with the tag of the original pointer, the PAC context differs, triggering PAC authentication failure.

Pointers that are difficult to determine whether they are privileged or not, such as `void *` pointers or list type pointers, are excluded from pointer integrity enforcement. Protecting all of these pointers with PAC are likely to cause compatibility issues. Therefore, PETAL focuses on protecting pointers explicitly typed as privileged pointers. This design may leave some void pointers unprotected, allowing pointer corruption attacks on them. We discuss the potential security risks of not protecting void pointers in §7.2.

Example of Privileged Pointer Protection. As shown in Figure 4b, `current` and `cred` are privileged pointers since they point to the access control policies (i.e., user credentials). `inode_hashtable` is also a privileged pointer pointing to the access control policies (i.e., file credentials). `files`, `fdt`, and `fd` are privileged pointers pointing to the resources (i.e., access modes of the file descriptor). PETAL enforces PAC sign and authentication on privileged pointer access to prevent attacks on privileged pointers. As a result, PETAL can prevent pointer corruption attacks on both policy and resources. `Bad Binder` attack (Figure 3a) is detected when the corrupted `cred` pointer is loaded and authenticated, because its `PAC` is not signed with the correct context.

5.2.3 Spilled Register Protection. Data in registers can be compromised when they are spilled to memory, triggering a Time-of-Check to Time-of-Use (TOCTOU) attack [14, 16, 89]. If a register contains data protected with MTE or PAC, the protection can be bypassed by corrupting the register data spilled to the memory. To prevent this, PETAL extends its protection scope to spilled registers.

First, on interrupt, registers are spilled to the interrupted task's stack and restored after the interrupt is handled. PETAL preserves the spilled registers by PAC-signing the registers and authenticating them when they are restored, as done in PAL [89] and the XNU kernel [7]. As in PAL, registers are signed in a chained manner, where a previously PAC-signed register is used as the PAC context for the next register. To prevent replay attacks, a CPU cycle counter (i.e., `cntpct_el0`) is used together as the PAC context. Thus, corrupting spilled registers is detected by PAC authentication failure. However, the confidentiality of spilled registers is not ensured, as spilled registers are not encrypted. To further provide the confidentiality, an isolated interrupt stack [61] can be used.

Second, on the context switch in AArch64, registers are spilled to the process descriptor object (i.e., `struct task_struct`). As `struct task_struct` is a privileged object, the spilled registers are already protected by the random MTE tag (§5.2.1).

Third, registers can be spilled to the stack by the compiler. Compiler-induced spills are not explicitly protected by PETAL, but exploiting such spills is challenging. The attack window is short, and the attacker needs to leak the stack pointer, which is stored in the MTE-protected process descriptor. To further reduce the attack surface, potential register spills can be identified and patched by static binary analysis [89] or compiler backend analysis [30].

6 IMPLEMENTATION

PETAL is implemented based on Android Linux kernel v5.10.136. Three LLVM passes are developed with LLVM/Clang-14.0.0 for all static analyses and instrumentation. All analyses are path-insensitive and field-sensitive, covering the whole kernel heap, stack, global memory, and assembly code. We leveraged `wllvm` [68] to extract a single LLVM IR bytecode from the kernel source code and used `PeX` [91] for indirect call resolution. We provide implementation details on source code instrumentation at §A.2.

User Interface Analysis. User interface analysis collects access control policies, resources, and their pointers from user interfaces (§5.1). The input is the kernel LLVM IR bytecode, and the output is a list of privileged objects and pointers in a struct type or a global variable name format. It first collects interface handlers, the kernel functions implementing specific system calls and file operations, which are collected from global interface handler tables: `tid_base_stuff` for `proc` file system, `sysctl_base_table` for `sysfs` file system, and `sys_call_table` for system calls. For each interface handler, static taint analysis and constraint analysis are performed to find the access control policies and the protected resources.

The static taint analysis is inter-procedural, context-sensitive, and field-sensitive, propagating the taint through LLVM's use-def chain [64]. Taint is propagated forward or backward depending on the taint source (§5.1.2). Indirect calls are resolved using a indirect call resolution analysis [91]. For context-sensitivity, the analysis maintains a trace of use-def chains from the source to the current taint. To avoid indefinitely propagating the taint in recursions or loops, a use-def edge is ignored if it is revisited in the same context.

For field-sensitivity, the analysis maintains an *access path* [19, 25], a sequence of fields to access the tracked data from a given pointer. Leveraging the access path, the analysis tracks data-flow through a chain of memory objects (e.g., Figure 2). For instance, suppose a user input variable, `x`, is initially tainted. Since the variable itself is tracked, `x` is tainted with an empty access path `{}`. If `x` is stored to `ptr1->field1`, `ptr1` is tainted with the access path `{->field1}`. If `ptr1` is stored to `ptr2->field2`, `ptr2` is tainted with the access path `{->field2->field1}`. When `ptr2->field2->field1` is later loaded into `y`, `y` is tainted with the empty access path `{}`. Therefore, the analysis can track the data-flow from `x` to `y`, mediated by a chain of memory objects. To resolve pointer aliases in the chain of memory objects, the analysis performs an on-demand backward analysis as suggested by `FlowDroid` [11]. When tainting an object, the analysis recursively tracks its pointer backward to find the root object of the chain. The root object, identified by memory allocation or a global variable, is then tracked with the computed access path.

To apply the static taint analysis to the Linux kernel, we made several simplifications that may introduce false positives and negatives. First, we assumed a linear chain of memory objects in the on-demand backward analysis. Therefore, in a complex object relationship, where a single object is linked to multiple root objects, the analysis only identifies one root object. According to our evaluation, we observed one such false negative case in `dentry` related code (§7.1). This can be addressed by improving the backward analysis to handle multiple root objects. Second, for variable-offset pointer arithmetic, the analysis treats the offset as `0`, which may introduce false positives and negatives. However, in the Linux kernel, many pointer arithmetic operations use constant offsets for struct fields, while variable offsets are mostly array indexes. By treating variable offsets as `0`, the analysis over-approximates all array elements as a single object, which has a minimal impact on the result, as it only requires the object's type or name information.

Constraint analysis determines whether access to kernel metadata occurs after a successful permission check. For kernel metadata accessed through system calls, the analysis returns true i) if a permission check branch is reachable from the system call entry point, and ii) if the access is reachable from the permission check's successful branch. For kernel metadata accessed through pseudo file systems, which by default enforce DAC checks, the analysis returns true when the file's access mode is owner-only read or write. The returned information can be used as a constraint to find the kernel metadata only accessible by privileged users (e.g., file owner). For instance, `sysctl_base_table` stores the access mode of `/proc/sys/kernel/modprobe` as `644` (i.e., owner-only write). The constraint analysis would confirm that the write is only allowed for the file owner, and thus the kernel metadata updated by user through the interface (i.e., `modprobe_path`) is identified as resource.

Privileged Object Protection. Privileged object protection (§5.2.1) assigns a non-zero random MTE tag to privileged objects and tag 0 to non-privileged objects. Kernel allocators are modified to assign tag 0 at the allocated object's pointer by default, and reset the tag to 0 when a tagged object is deallocated. Privileged objects in the global, heap, and stack are tagged at allocation by instrumentation. Global objects are tagged once at the kernel boot time. Heap objects are tagged at allocation and the tag is reset with 0 at deallocation. Stack objects are tagged and reset at the object's first and last usage identified by `llvm.lifetime` intrinsics.

PETAL enforces tag 0 on non-privileged object access to isolate non-privileged and privileged objects. To this end, we classified memory access instructions into *non-privileged access*, *privileged access*, and *mixed access* based on whether the memory access uses non-privileged pointers, privileged pointers, or both. We first collected privileged and non-privileged pointers from (1) function argument, (2) return value, (3) local variable, and (4) global variable. We classified the pointers based on their type information and global variable names. For generic pointers, such as `void*` and list-type struct pointers (e.g., `struct list_struct`), we classified them based on their cast types. If the pointer is cast to a privileged pointer type, it is considered as a privileged pointer, and the same applies to the non-privileged pointer types. If not cast, the pointer is

conservatively considered as a privileged pointer to avoid enforcing tag 0 on benign privileged access.

Then, we conducted a comprehensive points-to analysis on the kernel code to collect memory access instructions using either non-privileged pointers, privileged pointers, or both. Based on the result, we classified memory instructions into three categories: Non-privileged access exclusively using non-privileged pointers, privileged access exclusively using privileged pointers, and mixed access using both privileged and non-privileged pointers. We enforced tag 0 on non-privileged access by performing a bitwise AND operation on the pointer with instrumentation. Privileged and mixed access expect the pointer to contain the matching tag of the accessed object, so they are not modified.

To preserve compatibility with the kernel memory management internals, we globally set `TCMA1` to allow the match-all tag (i.e., `0xf`) to skip the MTE tag check. Although match-all tag may introduce a security risk, pointers contain the tag ranging from 0x0 to 0xe, so another memory corruption to forge the match-all tag is required. We evaluate the risk of allowing match-all tag in §7.2.2.

Privileged Pointer Protection. To implement privileged pointer protection (§5.2.2), we classified memory access instructions into *privileged* and *non-privileged pointer access*. Similar to privileged object protection, we first identified privileged pointers based on their type and global variable name. Then, we collected memory instructions accessing privileged pointers from points-to analysis. Privileged pointer load/store instructions are modified to authenticate/sign the pointer using the 64-bit memory address as a PAC modifier. As the `PAC` is bound to the memory address, copy instructions (e.g., `memcpy` and `memmove`) should preserve the pointer integrity while copying a privileged pointer to a different address. We selectively re-signed pointers for legitimate pointer copy instructions. With the points-to analysis, we collected the instructions that potentially copy privileged pointers, along with the offsets of the privileged pointers in the copied memory. These copy instructions are modified to load and authenticate privileged pointers in the given offsets, re-sign the pointer, and store the new pointer in the destination memory. This does not introduce arbitrary PAC signing gadgets, as authentication failure during copy is detected.

In-pointer Metadata. Android Linux kernel uses 39-bit virtual address space, and PETAL uses the top 8-bits (Bit[63:56]) and 16-bits (Bit[54:39]) for the MTE tag and PAC, respectively.

Linux Kernel and LLVM Modifications. Several minor modifications were made in LLVM and the Linux kernel to ensure the 16-byte alignment of kernel objects, handle `container_of` macro, and preserve type information in LLVM IR. We further removed known MTE tag software side-channels in the `kcmp` system call [53].

7 EVALUATION

This section evaluates PETAL, focusing on its security and performance. First, we evaluate the protection scope of the static analysis (§7.1). Next, we evaluate the security of PETAL theoretically and empirically (§7.2). Then, we demonstrate the effectiveness of PETAL with real-world exploits (§7.3). Finally, we evaluate the performance of PETAL for kernel and user-space workloads (§7.4). We show the instrumentation overheads in the appendix (§A.1).

7.1 Effectiveness of Protection Scope

Analysis Results. PETAL utilizes the Linux kernel's user interfaces to identify access control policies and resources (§5.1). From Android Linux kernel v5.10.136, we analyzed 35 files from `proc`, 322 files from `sysfs`, and 416 system calls. Our analysis identified 507 kernel struct types, 449 stack objects, and 350 global objects and and 141 global pointers as privileged, out of 9,371 struct types and 104,820 global variables. Our results include well-known kernel objects and pointers that are known to be privileged (Figure 4b).

We manually inspected the results to identify false negatives and false positives. To identify false negatives, we referenced the Linux `man page` [44] to understand the semantics of each user interface, such as which argument should update what kernel metadata, and whether any privilege check is performed. Based on this domain knowledge, we cross-checked the identified privileged objects and pointers with the user interface source code. To identify false positives, we checked whether the user interface code can indeed read or write the identified privileged objects and pointers. As a result, we found three false negatives and no false positives. For three false positives, `dac_mmap_min_addr` and `struct dentry_hashtable` were missed due to complex data flows [4] , which can be handled by improving the static taint analysis. `struct cgroup_root` was missed since directory creation (i.e., `mkdir`) operation was not analyzed, which can be handled by adding support in the future. The analysis confirmed no false positives, indicating that all identified privileged objects and pointers are indeed used as access control policies, resources, or the pointers.

Compared to the previous approach focused on access control *policies* [77], PETAL newly uncovered *resources* protected by the policies (314 struct types and 220 global variables). From `proc`, we identified process-specific resources whose access is managed by DAC. From `/proc/[pid]/syscall`, we identified `struct syscall_info`, which contains system call arguments of a process that might contain sensitive information. From `sysfs`, we identified critical root-only system configurations such as `modprobe_path` and `randomize_va_space` from `/proc/sys/kernel/modprobe` and `/proc/sys/kernel/randomize_va_space`, respectively. Corrupting `modprobe_path` would allow arbitrary execution in root-privilege [69], while corrupting `randomize_va_space` would disable ASLR [27].

From system calls, we identified resources managed by various access control systems. From `setsockopt`, whose permission is managed by SELinux, we identified socket-related resources. One example is `struct sock_npa_vendor_data`, which contains network traffic information used by Samsung Knox Network Platform Analysis (NPA) [70]. Illegal memory access on this object can leak or manipulate sensitive network traffics monitored by NPA. From `prctl`, we identified `struct mm_struct` protected by Capabilities (i.e., `CAP_SYS_RESOURCE`). `mm_struct` contains sensitive memory mapping information, which can be used to bypass KASLR or ASLR [21].

7.2 Security Analysis

We evaluate the security effectiveness of PETAL by enumerating all theoretical attack vectors (§7.2.1). Next, based on the theoretical

[4] `dac_mmap_min_addr` was missed because the use-def analysis stops when it reaches a global variable. `struct dentry_hashtable` was missed because the on-demand backward analysis does not handle multiple root objects in the object chain.

Table 1: Success rate of PeTAL against attack vectors

(a) Object Corruption Attack

Pointer Type	Access Type: Non-priv.	Priv.	Mixed
Priv.	N/A	Match tag (7.14%)	Match tag (7.14%)
Non-priv.	Enforce tag 0 (0.00%)	N/A	Match tag (0.00%)
Void	Enforce tag 0 (0.00%)	Match tag (0.00% or 7.14%)	Match tag (0.00% or 7.14%)

(b) Pointer Corruption Attack

Pointer Type	Access Type: Non-priv.	Priv.	Mixed
Priv.	N/A	PAC Auth (0.002% or 7.14%)	PAC Auth (0.002% or 7.14%)
Non-priv.	Enforce tag 0 (0.00%)	N/A	**No protection** (100.00%)
Void	Enforce tag 0 (0.00%)	**No protection** (100.00%)	**No protection** (100.00%)

security analysis, we further evaluate the empirical security impacts exploiting vulnerable cases of PeTAL (§7.2.2).

7.2.1 Theoretical Security Analysis. In this security analysis, we enumerate all theoretically possible attack vectors to bypass the protection of PeTAL. Specifically, the attacker's goal is to illegally access privileged objects leveraging a vulnerable memory access, such as out-of-bound access or use-after-free. Further, following the threat model (§4), it is assumed that the attacker does not know the tag of the privileged object.

To achieve the attacker's goal, there can be two general attack methods. The first attack is object corruption, where the vulnerable memory access directly accesses the privileged object (e.g., use-after-free access to the privileged object). The second attack is pointer corruption, where the attacker first corrupts a pointer and uses the corrupted pointer to access the privileged object (e.g., corrupting a neighboring pointer with out-of-bounds access).

Object Corruption Attacks. Object corruption attacks illegally access a privileged object with a vulnerable memory access, without corrupting any pointers. As shown in Table 1, this attack can be further categorized depending on (i) the type of pointer to access the privileged object (i.e., privileged, non-privileged, and void pointers) and (ii) the type of the vulnerable memory access (i.e., non-privileged, privileged, and mixed access).

If a privileged pointer is used, the memory access is either privileged or mixed (there are no non-privileged access using a privileged pointer in PeTAL). In both cases, the pointer contains a non-zero tag, and PeTAL matches the tag in the pointer with the one in the pointed privileged object. As the tag is randomly assigned among 14 tags, the attack is successful when two tags match, with a rate of 1/14 (7.14%).

When a non-privileged pointer is used, the memory access is either non-privileged or mixed. In the non-privileged access, PeTAL enforces tag 0 to the pointer, thus this attack cannot access the privileged object with non-zero tag (0.00%). In the mixed access, PeTAL does not enforce tag 0, but the non-privileged pointer would contain tag 0, thus the attack fails (0.00%).

When a void pointer is used, the memory access can be either non-privileged, privileged, or mixed. In the non-privileged access, PeTAL enforces tag 0 to the void pointer, thus it cannot access the privileged object (0.00%). In the privileged or mixed accesses, PeTAL matches the void pointer's tag with the one in the privileged object. If the void pointer has a tag 0, the attack cannot succeed (0.00%). If the void pointer has a non-zero tag, the attack success rate is 1/14 (7.14%).

Pointer Corruption Attacks. Pointer corruption attacks first corrupt a pointer with vulnerable memory access, and later use the corrupted pointer to illegally access a privileged object. We assume that the corrupted pointer is in a non-privileged object. We do not consider the case that the corrupted pointer is in a privileged object, because such an attack is an object corruption attack that we analyzed above. Similar to the categorization of the object corruption attack, we categorize the pointer corruption attacks based on (i) the type of the corrupted pointer and (ii) the type of the illegal memory access using the corrupted pointer.

If a privileged pointer is corrupted, the pointer corruption is detected by PAC authentication failure by PeTAL at the time of pointer load. If the attacker has to guess the correct 16-bit PAC to bypass the PAC authentication, the success rate is 1/65,536 (0.002%). If the attacker spatially reuses the previously signed PAC by copying a privileged pointer from one address to another, the attack is detected by PAC context mismatch, unless PAC collision occurs (0.002%). If the attacker temporally reuses the previously signed PAC from the previous allocation, such as use-after-free, the attack is detected by PAC context mismatch when the tag of two allocations are different. Thus, the success rate is 1/14 (7.14%).

If a non-privileged pointer is corrupted, the pointer corruption is not detected because PeTAL does not authenticate non-privileged pointers. This non-privileged pointer can be used by either non-privileged or mixed access. In case of non-privileged access, the attack always fails to access the privileged object since the pointer is enforced with tag 0 (0%). However, if the corrupted pointer is used in mixed access, the attack succeeds if the pointer tag is corrupted with match-all tag (0xf) to access the privileged object.

Similarly, when a void pointer is corrupted, the pointer corruption is not detected by PeTAL, as void pointers are not authenticated by PAC. The void pointer can be used in either non-privileged, privileged, or mixed access. If the corrupted pointer is used in non-privileged access, the attack always fails since the pointer is enforced with tag 0 (0.00%). However, if the corrupted pointer is used in privileged or mixed access, the attack succeeds due to the same problem of the non-privileged pointer—i.e., the attacker can embed the match-all tag (0xf). §7.2.2 further empirically evaluates unprotected memory accesses, and shows their security impact is fairly limited in practice.

7.2.2 Empirical Security Analysis. To understand the unprotected attack vectors (§7.2.1), we further show their empirical security impacts. We particularly focused on whether a successfully corrupted pointer would be used in memory access, allowing to access privileged objects. Specifically, PeTAL has three types of unprotected attack vectors: i) non-privileged pointer corruption with mixed access, ii) void pointer corruption with privileged access, and iii) void pointer corruption with mixed access. We assume the attacker corrupts a pointer to contain a match-all tag (0xf) and the address of a privileged object, and uses the corrupted pointer to illegally access the privileged object.

Next, we analyzed the exploitability of each unprotected attack vector. An attacker vector is considered exploitable if the corrupted pointer does not raise any PAC authentication failure later. Specifically, if the corrupted pointer is used to load any privileged pointer, the pointer would be authenticated by PAC. In this case, if the corrupted pointer points to a different type of object not containing the privileged pointer, the PAC authentication would fail. If the corrupted pointer points to the same type of object that contains a PAC-signed privileged pointer, the mismatching PAC context (the corrupted MTE tag value 0xf) would fail the PAC authentication.

Among total memory access, 7.52% were mixed access using potentially corruptible non-privileged pointers, 2.96% were privileged access using potentially corruptible void pointers, and 9.43% were mixed access using potentially corruptible void pointers. However, the exploitable accesses that do not involve subsequent PAC authentication were around 3%. Specifically, 2.51% were exploitable for non-privileged pointer corruption with mixed access, 0.17% were exploitable for void pointer corruption with privileged access, and 0.77% were exploitable for void pointer corruption with mixed access. This result shows that the majority of data-flows are protected by PeTAL, significantly reducing the attack surface.

7.3 Concrete Attack Analysis

We show the effectiveness of PeTAL against data-only attacks with two real-world attacks: `Bad Binder` [57] and `DirtyCred` attacks [49]. The attacks were evaluated on QEMU [15] with 4 CPU cores.

Breaking Policy Integrity with Bad Binder. We tested the `Bad Binder` attack [57] for policy integrity violation as described in §3.2. The attack overwrites the `struct cred` pointer with the address of `init_cred`. While the attack is successful on the vanilla Linux kernel, it fails on PeTAL-hardened Linux kernel. PeTAL classifies `struct cred` pointer as a privileged pointer since it stores various credential data used in permission checks. Therefore, the corrupted pointer is authenticated when it is used, and the authentication fails since the pointer is not generated by the legitimate PAC signing instruction, with the probability of 99.998%.

Breaking Complete Enforcement with DirtyCred. We tested `DirtyCred` attack [49] that violates complete enforcement (§3.2). The attack leverages a use-after-free vulnerability on `struct file`, exploiting two different file descriptors pointing to the same `struct file` object. While successful on the vanilla Linux kernel, the attack was always detected on PeTAL-hardened Linux kernel on 100 runs. This is because PeTAL protects `struct file` as a privileged object, randomly tagging the object, so the use-after-free access on the file descriptor failed with 13/14 probability due to MTE tag mismatch. Even when the MTE tags collided, when the file descriptor was freed, the memory tag is reset to 0, which is detected when the dangling pointer is dereferenced. A successful attack would require additional techniques to avoid access on the freed memory.

7.4 Runtime Overhead

Evaluation Setup. We applied PeTAL on Android Linux v5.10.136 for Samsung Galaxy S22 Pamir (SM-S906B), an Exynos chip-based device supporting both ARMv8 MTE and PAC. Backward-edge CFI is applied on the baseline kernel [52]. Forward-edge CFI is implemented by integrating PAL [89], which leverages ARM Pointer Authentication (PAC). The default Android kernel's CFI [6], which is based on Clang/LLVM CFI [20], is also compatible with PeTAL, because PeTAL's MTE and PAC-based protection does not interfere with the function signature-based CFI checks.

Table 2: Performance overhead (x) on LMBench

Test	*Obj*		*Obj+Ptr*		*Obj+Ptr+PAL*		Kenali
	async	*sync*	*async*	*sync*	*async*	*sync*	
Null syscall	1.03	1.46	1.06	1.50	1.12	1.50	1.00
read	1.07	1.12	1.09	1.41	1.10	1.42	n/a
write	1.42	1.41	1.44	1.45	1.44	1.56	n/a
fstat	1.24	1.35	1.38	1.41	1.41	1.41	n/a
open_close	1.30	1.48	1.32	1.50	1.32	1.53	2.76
select_10	1.02	1.32	1.17	1.39	1.17	1.41	1.42
sig_install	1.08	1.32	1.25	1.43	1.25	1.43	1.3
sig_catch	1.02	1.05	1.13	1.16	1.13	1.24	2.23
unix_sock	1.03	1.01	1.01	1.01	1.01	1.01	n/a
fork_exit	1.00	1.03	1.03	1.04	1.03	1.06	2.18
fork_execve	1.10	1.10	1.10	1.10	1.10	1.10	2.26
Average	1.11	1.23	1.17	1.30	1.18	1.32	1.78

We measured runtime overhead of each benchmark by comparing the results from vanilla kernel [73] (baseline) and PeTAL-hardened kernel (PeTAL). Maximum six distinct settings varied in the protection (*Obj*, *Obj+Ptr*, *Obj+Ptr+PAL*), and the MTE mode (*async*, or *sync*). In *Obj*, only privileged object protection is applied, while in *Obj+Ptr*, full protection of PeTAL is applied. In *Obj+Ptr+PAL*, PeTAL is applied in combination with PAL, which provides forward-edge CFI protection. The backward-edge CFI [2] is applied on the baseline. *async*, and *sync* denote that MTE is enabled in asynchronous mode and synchronous mode, respectively. Asynchronous mode detects a tag mismatch at context switch, whereas synchronous mode detects the mismatch at memory access time. The user-application benchmarks were evaluated in *Obj+Ptr+PAL* and *sync*. All benchmark performance were averaged over 20 runs.

Memory Overhead. We measured runtime memory overhead of PeTAL caused by 16-byte alignments on kernel objects, leveraging `/proc/meminfo`'s `Slab` field. After booting, PeTAL adds 324 KB (1.42%) from the baseline kernel (22,836 KB). After LMBench, PeTAL adds 328 KB (1.43%) from the baseline kernel (22,924 KB). The results shows that PeTAL incurs a negligible memory overhead.

Micro-Benchmark: LMBench. Table 2 shows the performance overheads of kernel workloads in PeTAL-hardened kernel compared to the baseline. Each overhead was measured in both MTE sync and async modes. Several system calls that showed inconsistent results in baseline (e.g., `pipe`, `fork+shell`) were excluded.

The overhead of PeTAL with PAL was 1.32x on average, where most of the overhead was caused by MTE tag check. The highest overhead was observed in `open_close`, as file systems operate on many access control policies and resources protected by PeTAL. The lowest overhead was observed in `fork_exit`. We believe this is because the experiment was conducted on a real Android device, where various factors (e.g., scheduling) can affect the performance of such system calls. Furthermore, we compared the performance overhead of PeTAL with Kenali, a software DFI solution for Linux designed to protect sensitive control and non-control data [77]. The result showed that PeTAL, when combined with PAL, has a 35%-51% performance gain over Kenali, highlighting the efficiency of PeTAL's hardware-assisted DFI solution. In real-world settings, the

Table 3: Performance evaluation on user-application benchmarks (*Obj+Ptr+PAL* and *sync*)

(a) NBench

Test	Base (s)	PeTAL (s)
NumSort	20.0	20.0 (1.00x)
StrSort	87.7	87.3 (1.00x)
BitFld	25.5	25.4 (1.00x)
FPEmu	127.3	127.0 (1.00x)
Fourier	142.7	142.4 (1.00x)
Assign	58.7	59.0 (1.00x)
Idea	70.1	70.0 (1.00x)
Huff	92.4	91.1 (1.01x)
Nnet	163.5	161.4 (1.01x)
LUDec	187.5	186.8 (1.00x)
Average		1.00x

(b) LevelDB

Test	Base (us)	PeTAL (us)
fillseq	5.0	5.4 (1.06x)
fillsync	36.4	37.6 (1.03x)
fillrandom	6.0	6.3 (1.05x)
overwrite	6.3	6.6 (1.05x)
readrandom	2.4	2.4 (1.01x)
readseq	0.2	0.2 (1.00x)
readreverse	0.3	0.3 (1.00x)
Average		1.03x

(c) Apache httpd

# Req, Size	Base (ms)	PeTAL (ms)
500, 1KB	3.2	3.28 (1.03x)
500, 10KB	3.28	3.29 (1.00x)
500, 1000KB	8.41	8.98 (1.07x)
1000, 1KB	4.2	4.54 (1.08x)
1000, 10KB	4.23	4.29 (1.02x)
1000, 1000KB	8.53	9.02 (1.06x)
Average		1.04x

performance overhead of PeTAL can be further reduced with MTE Asynchronous mode (1.18x on kernel workloads). In asynchronous mode, the tag check is deferred to the context switch, where the kernel manually checks the tag fault status register (`TFSR_EL1`). Switching to asynchronous mode would not affect the security guarantees of PeTAL. The kernel only updates `TFSR_EL1` during boot and after handling the tag check fault, and CFI prevents invalid control flow to such kernel code, mitigating the risk of clearing the tag fault status register [5].

Application Benchmark: NBench. We evaluated the impact of PeTAL on CPU-centric user application using NBench [62]. The result, suggests that the performance overhead introduced by PeTAL was negligible, being less than 1% (Table 3a). This minimal overhead can be attributed to the nature of NBench, a CPU-centric benchmark, which does not frequently invoke system calls.

Application Benchmark: LevelDB. To measure the performance of PeTAL on I/O intensive applications, we used LevelDB [35], a key-value database Table 3b. The performance overhead introduced by PeTAL was 1.03x on average, 1.05x for write operations, and 1.02x for read operations.

Application Benchmark: Apache httpd. Performance on network-intensive applications, specifically on a widely used web server, Apache httpd [32] are also evaluated. With ApacheBench [31], we tested 500 and 1,000 requests on files with sizes ranging from 1KB to 1MB. The results are shown in Table 3c, where PeTAL showed an overall performance overhead of 1.04x.

8 DISCUSSION

This section discusses the current limitations of PeTAL and further outlines potential solutions to address them.

MTE Security. PeTAL utilizes 4-bit MTE tags to safeguard privileged objects. If attackers leak an MTE tag of the target privileged object, they could set the leaked MTE tag at an unprotected data pointers and access the target object. However, MTE tags stored in kernel pointers are difficult to leak as the Linux kernel does not expose kernel pointers to user-space, and memory corruption-based information leakage is limited due to MTE enforcement. Still, the attackers can guess the MTE tags to access privileged objects, with at most a 1/14 chance of correctly guessing the MTE tag. We believe this still makes real-world attacks significantly challenging, especially to bypass every MTE and PAC enforcement related to the exploitation, as shown in §7.2.2 and §7.3.

Limitation of Type-based Data-Flow Integrity. As a type-based DFI solution, PeTAL avoids compatibility issues with unsafe C codes, but does not strictly enforce DFI on generic pointers. However, as discussed in §7.2.2, attack surface leveraging generic pointers is small due to the combined enforcement of MTE and PAC. In addition, this limitation is similar to those of the de-facto fine-grained forward CFI techniques [20, 33, 81], which provide a conservative protection scheme based on function prototypes. Future work could explore a more fine-grained DFI enforcement by separating the generic pointers for the privileged and non-privileged usages, leveraging well-defined wrappers (e.g., `list_add`).

In-Kernel Virtual Execution. In-kernel virtual execution, with BPF/eBPF [23, 54] (referred to as BPF hereafter) as a representative example, is yet another attack vector of the Linux kernel [39, 43]. By exploiting BPF verification bugs or corrupting BPF programs, attackers can execute arbitrary code in the kernel. Although PeTAL does not directly protect BPF, as MTE and PAC are enforced throughout the kernel, memory corruption attacks through BPF still go through MTE checks and cannot forge a PAC-signed pointer. However, by attacking void pointers that are not protected by PeTAL, or setting the match-all MTE tag (`0xf`), attackers could bypass PeTAL's protection. While the kernel restricts unprivileged BPF usage [38], to completely mitigate such attacks, specialized defenses, such as enforcing CFI [43] and validating the verifier logic [79], can be employed in conjunction with PeTAL.

9 RELATED WORK

ARM Pointer Authentication and Memory Tagging. ARMv8.5-A and v9-A support pointer authentication code (PAC) and memory tagging extension (MTE). Previous works have proposed various security solutions based on ARM PAC and MTE. `PARTS` [47] introduced the concept of pointer integrity protection with PAC. `PAL` [89] is a PAC-based kernel control flow integrity. `HAKC` [55] confined untrusted kernel modules with PAC and MTE. `Capacity` [26] developed intra-process isolation based on MTE and PAC.

Data-only Attacks. With the common use of control-flow integrity (CFI) in modern systems, attackers have shifted their focus to data-only attacks. Chen et al. [18] first proposed the concept and impact of data-only attacks. Evans et al. [29] bypassed code pointer integrity (CPI) with data pointer manipulation. `FLOWSTITCH` [41] and `BOP` [42] proposed the concept of Data-Oriented Programming

(DOP) and constructed Turing-complete data-only exploits for arbitrary programs. `DirtyCred` [49] is a kernel privilege escalation attack that swaps file descriptors with data-only attacks. Some works focused on finding objects vulnerable to data-only attacks. `AlphaEXP` [83] identifies kernel objects that can be used for data-only attacks given a vulnerability. `Viper` [87] and `DPP` [3] find data vulnerable to data-oriented attacks in user space.

Data-Flow Integrity. Castro et al. [17] proposed data-flow Integrity (DFI) by enforcing the data-flow based on static analysis. WIT [4] enforced code-level write integrity. To reduce the performance overhead, `HDFI` [78] and `Kenali` [77] proposed hardware-assisted DFI solutions. Intra-process isolation have been studied and implemented based on various hardware features, such as Intel MPK [34, 40, 75, 76, 82], Intel CET [85], Intel Extended Page Tables [66], SMAP/PAN [61, 84, 86], and MTE+PAC [26, 55]. However, Intra-process isolation cannot prevent memory corruption within the same security domain. Guoren et al. [46] proposed an alias analysis to automatically identify global variables that can be set as read-only. PETAL focuses on providing access control integrity by selectively protecting kernel objects with ARM MTE and PAC.

10 CONCLUSION

PETAL is a practical data-flow integrity (DFI) technique to provide access control integrity–i.e., policy integrity and complete enforcement. PETAL orchestrates ARM Pointer Authentication Code (PAC) and Memory Tagging Extension (MTE) on ARM-based Linux kernel. The runtime overhead of PETAL is at most 4% on average in user applications, demonstrating its effectiveness for kernel security. We believe PETAL has strong potential to be employed in practice, which would significantly raise the security bar for kernel attacks.

ACKNOWLEDGMENT

We greatly appreciate the anonymous reviewers for their valuable and constructive feedback. We thank the anonymous shepherd for guiding us through the revision process. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University, Grant FA8750-24-2-0002 under DARPA I20, and Samsung Research and System Core Lab at Mobile eXperience (MX) Division, Samsung Electronics Co.,Ltd. The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

REFERENCES

[1] x86 nx support, 2004. https://lwn.net/Articles/87814/.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. In *Proceedings of the ACM Transactions on Information and System Security*, Nov. 2009.

[3] S. Ahmed, H. Liljestrand, H. Jamjoom, M. Hicks, N. Asokan, and D. D. Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against {Data-Oriented} attacks. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[5] S. Amar. Security analysis of mte through examples, 2022. https://msrndcdn360.blob.core.windows.net/bluehat/bluehatil/2022/assets/doc/Security%20Analysis%20of%20MTE%20Through%20Examples__Saar%20Amar.pdf.

[6] Android. Kernel control flow integrity, 2024. https://source.android.com/docs/security/test/kcfi.

[7] Apple. Armv8.3 pointer authentication in xnu, 2021. https://opensource.apple.com/source/xnu/xnu-7195.60.75/doc/pac.md.

[8] Apple. Operating system integrity, 2024. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web.

[9] Apple. Page protection layer, 2024. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/1/web/1#sec314c3af61.

[10] Arm. Memory tagging extension, 2019. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[12] B. Azad. Examining pointer authentication on the iphone xs, 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[13] B. Azad. A study in pac, 2019. https://bazad.github.io/presentations/MOSEC-2019-A-study-in-PAC.pdf.

[14] B. Azad. ios kernel pac, one year later, 2020. https://i.blackhat.com/USA-20/Wednesday/us-20-Azad-iOS-Kernel-PAC-One-Year-Later.pdf.

[15] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[16] Z. Cai, J. Zhu, W. Shen, Y. Yang, R. Chang, Y. Wang, J. Li, and K. Ren. Demystifying pointer authentication on apple m1. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[17] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[18] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2005.

[19] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

[20] Clang. Control flow integrity, 2024. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[21] K. Cook. proc: protect mm start_code/end_code in /proc/pid/stat, 2011. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5883f57ca0008ffc93e09cbb9847a1928e50c6f3.

[22] J. Corbet. Supervisor mode access prevention, 2012. https://lwn.net/Articles/517475/.

[23] J. Corbet. Bpf: the universal in-kernel virtual machine, 2014. https://lwn.net/Articles/599755/.

[24] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[25] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994.

[26] K. Dinh Duy, K. Cho, T. Noh, and H. Lee. Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenghagen, Denmark, Nov. 2023.

[27] J. Edge. Kernel address space layout randomization, 2013. https://lwn.net/Articles/569635/.

[28] J. Edge. Control-flow integrity for the kernel, 2020. https://lwn.net/Articles/810077/.

[29] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[30] A. Fanti, C. C. Perez, R. Denis-Courmont, G. Roascio, and J.-E. Ekberg. Toward register spilling security using llvm and arm pointer authentication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11): 3757–3766, 2022.

[31] T. A. S. Foundation. Apache http server benchmarking tool, 2022. https://httpd.apache.org/docs/2.4/programs/ab.html.

[32] T. A. S. Foundation. Apache http server project, 2022. https://httpd.apache.org.

[33] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis. Fineibt: Fine-grain control-flow enforcement with indirect branch tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Hong Kong, China, Oct. 2023.

[34] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, USA, Apr. 2021.

[35] google. leveldb, 2023. https://github.com/google/leveldb.

[36] Google. syzbot dashboard, 2024. https://syzkaller.appspot.com/upstream.

[37] Grsecurity. How does rap works, 2024. https://grsecurity.net/rap_faq.

[38] P. Gupta. bpf:disallow unprivileged bpf by default, 2021. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8a03e56b253e9691c90bc52ca199323d71b96204.

[39] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li. Cross container attacks: The bewildered {eBPF} on clouds. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[40] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[41] 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[42] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Tronto, Canada, Oct. 2018.

[43] D. Jin, V. Atlidakis, and V. P. Kemerlis. {EPF}: Evil packet filter. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2023.

[44] M. Kerrisk. Linux man pages online, 2024. https://man7.org/linux/man-pages/index.html.

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.

[46] G. Li, H. Zhang, J. Zhou, W. Shen, Y. Sui, and Z. Qian. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[47] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[48] A. Limited. Arm a64 instruction set for a-profile architecture, 2024. https://developer.arm.com/documentation/ddi0602/latest/.

[49] Z. Lin, Y. Wu, and X. Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

[50] Linux. capabilities(7), 2024. https://man7.org/linux/man-pages/man7/capabilities.7.html.

[51] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[52] LukeCheeseman. [aarch64] - return address signing, 2018. https://reviews.llvm.org/D49793.

[53] G. P. Z. Mark Brand. Mte as implemented, part 3: The kernel, 2023. https://googleprojectzero.blogspot.com/2023/08/mte-as-implemented-part-3-kernel.html.

[54] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[55] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow. Preventing kernel hacks with hakc. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Apr. 2022.

[56] Microsoft. Control flow guard for platform security, 2024. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[57] MITRE. CVE-2019-2215., 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2215.

[58] MITRE. CVE-2020-8835., 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835.

[59] MITRE. CVE-2022-4154., 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4154.

[60] M. Y. Mo. Gaining kernel code execution on an mte-enabled pixel 8, 2024. https://github.blog/security/vulnerability-research/gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/.

[61] M. Momeu, F. Kilger, C. Roemheld, S. Schnückel, S. Proskurin, M. Polychronakis, and V. P. Kemerlis. Islab: Immutable memory management metadata for commodity operating system kernels. In *Proceedings of the 19th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Singapore, July 2024.

[62] nbench. Linux/unix nbench, 1996. https://www.math.utah.edu/~mayer/linux/bmark.html.

[63] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, USA, May 2021.

[64] L. Project. Iterating over def-use & use-def chains, 2024. https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains.

[65] T. C. Projects. Control flow integrity, 2024. https://www.chromium.org/developers/testing/control-flow-integrity/.

[66] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xmp: selective memory protection for kernel and user space. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, USA, May 2020.

[67] Qualcomm. Pointer authentication on armv8.3, 2017. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[68] T. Ravitch. Whole program llvm, 2023. https://github.com/travitch/whole-program-llvm.

[69] sam4k. Kernel exploitation techniques: modprobe_path, 2022. https://sam4k.com/like-techniques-modprobe_path.

[70] Samsung. Network platform analytics (npa), 2023. https://docs.samsungknox.com/admin/fundamentals/whitepaper/network-security/network-platform-analytics/.

[71] Samsung. Exynos 2200, 2024. https://semiconductor.samsung.com/processor/mobile-processor/exynos-2200/.

[72] Samsung. Real-time kernel protection (rkp), 2024. https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-for-android/core-platform-security/real-time-kernel-protection/.

[73] Samsung. Mobile, 2024. https://opensource.samsung.com/uploadList?menuItem=mobile.

[74] R. S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[75] 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 *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, USA, Aug. 2020.

[76] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard. Jenny: Securing syscalls for {PKU-based} memory isolation systems. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[77] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[78] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[79] H. Sun and Z. Su. Validating the {eBPF} verifier via state embedding. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[80] S. Tanda. Intel vt-rp - part 1. remapping attack and hlat, 2023.

[81] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[82] 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 *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[83] R. Wang, K. Chen, C. Zhang, Z. Pan, Q. Li, S. Qin, S. Xu, M. Zhang, and Y. Li. AlphaEXP: An expert system for identifying Security-Sensitive kernel objects. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[84] 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 *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, USA, May 2020.

[85] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang. Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

[86] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei, et al. Panic: Pan-assisted intra-process memory isolation on arm. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenghagen, Denmark, Nov. 2023.

[87] H. Ye, S. Liu, Z. Zhang, and H. Hu. {VIPER}: Spotting {Syscall-Guard} variables for {Data-Only} attacks. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[88] X. W. Yong Liu, Jun Yao. Usma: Share kernel code with me. May 2022.

[89] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim. {In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[90] G. P. Zero. Examining pointer authentication on the iphone xs, 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[91] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

```
void *__petal_set_random_tag(void *ptr, size_t size)
{
  // Generate a random tag
  u8 tag = irg();
  // Set the tag to the object
  mte_set_mem_tag_range(ptr, size, tag);
  // Set the tag to the pointer
  return __tag_set(ptr, tag);
}

struct cred *prepare_creds(void)
{
  // Privileged object allocation
  struct cred *new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
  // Set random tag to the object and the pointer
+ new = __petal_set_random_tag(new, sizeof(struct cred));
}
```

(a) Instrumentation for privileged object allocation

```
inline void *__petal_enforce_zero(void *ptr)
{
    return __tag_set(ptr, 0xf0);
}
struct binder_node *binder_init_node_ilocked()
{
    struct binder_node *node;
    binder_uintptr_t cookie = fp ? fp->cookie : 0;
    ...
-    node->cookie = cookie;
+    __petal_enforce_zero(node)->cookie = cookie;
}
```

(b) Instrumentation for non-privileged access

```
// Sign the pointer with the address
void *__petal_sign_ptr(void *ptr, void *addr)
{
    return pacda(ptr, addr);
}

// Authenticate the pointer with the address
void *__petal_auth_ptr(void *ptr, void *addr)
{
    return autda(ptr, addr);
}

int copy_creds(...)
{
    ...
-   p->cred = get_cred(new);
+   p->cred = __petal_sign_ptr(get_cred(new), &p->cred);
}
void revert_creds(...)
{
    const struct cred *override;
-   override = current->cred;
+   override = __petal_auth_ptr(current->cred, &current->cred);
}
```

(c) Instrumentation for privileged pointer access

Figure 7: Instrumentation implementation for PETAL.

A APPENDIX

A.1 Instrumentation Overhead

Table 4: Instrumentation Overhead

		Priv. Obj Protection	Priv. Ptr Protection
Instance	**Total**	**Priv. Obj**	**Priv. Ptr**
struct type	9,371	507 (5.41%)	507 (5.41%)
global variable	104,820	350 (0.33%)	141 (0.13%)
Memory alloc/free	**Total**	**Priv. Obj**	
heap alloc	4,723	385 (8.15%)	
heap free	7,723	931 (12.05%)	
stack variable	22,346	453 (2.03%)	
Memory access	**Total**	**Non-priv**	**Priv. Ptr**
load	535,612	194,153 (36.25%)	41,238 (7.70%)
store	228,677	74,373 (32.52%)	8,952 (3.91%)
copy	6,758	2,020 (29.89%)	369 (5.46%)

Table 4 shows the instrumentation overhead of PETAL, which is measured from the android Linux kernel v5.10.136, with the whole-kernel LLVM IR bytecode. Total struct type number is measured by counting struct type objects that are allocated in the kernel, specifically from struct-typed heap allocations, stack variables, and global variables. The number of total global variables in the kernel is measured by counting the global variable definitions. From user interface analysis, 507 struct types were collected as privileged objects and pointers. 350 global variables were collected as privileged objects, and 141 global variables as privileged pointers. In privileged object protection, we instrumented 8.15% of heap allocations, 12.05% of heap frees, and 2.03% of stack variables to insert the random tag at allocation and clear the tag at free. Then, we instrumented 36.25% of loads, 32.52% of stores, and 29.89% of copies to enforce tag 0 for non-privileged access. In privileged pointer protection, we instrumented memory access to privileged pointers, which are 7.70% of loads, 3.19% of stores, and 5.46% of copy instructions.

A.2 Code Instrumentation

Figure 7 demonstrates the code changes with the instrumentation for PETAL. For privileged object allocation (Figure 7a), we inserted the `__petal_set_random_tag` function to assign a random tag to the allocated object. For non-privileged access (Figure 7b), we inserted the `__petal_enforce_zero` function to enforce tag 0 for non-privileged access. For privileged pointer access (Figure 7c), we inserted the `__petal_sign_ptr` and `__petal_auth_ptr` on pointer load and store instructions to preserve the integrity of privileged pointer.