DOPE: DOmain Protection Enforcement with PKS

Lukas Maar
Graz University of Technology
Graz, Austria
lukas.maar@iaik.tugraz.at

Martin Schwarzl
Graz University of Technology
Graz, Austria
schwarzl.marteun@gmail.com

Fabian Rauscher
Graz University of Technology
Graz, Austria
fabian.rauscher@iaik.tugraz.at

Daniel Gruss
Graz University of Technology
Graz, Austria
daniel.gruss@iaik.tugraz.at

Stefan Mangard
Graz University of Technology
Graz, Austria
stefan.mangard@iaik.tugraz.at

ABSTRACT

The number of Linux kernel vulnerabilities discovered has increased drastically over the past years. In the kernel, even simple memory safety vulnerabilities can have devastating consequences, e.g., compromising the entire system. Efforts to mitigate these vulnerabilities have so far focused mainly on control-flow hijacking attacks in the kernel. Yet, data-oriented attacks remain largely unmitigated in practice as existing mitigations are limited in providing robust security guarantees at reasonable performance overhead for multiple sensitive data objects.

In this paper, we present DOmain Protection Enforcement (DOPE), a novel kernel mitigation to protect against data-oriented attacks leveraging Intel's new hardware feature PKS. DOPE enforces domain protection, restricting memory access to sensitive data during kernel space execution based on the principle of least privilege. Hence, in case of an exploitable kernel bug, an attacker is prevented from using sensitive data for privilege escalation. We demonstrate DOPE's effectiveness and usefulness by implementing a proof-of-concept, protecting eight selected sensitive data objects. The proof-of-concept is realized as compiler-assisted and hardware-enforced kernel mitigation. It consists of less than 5000 lines of code on the Linux kernel 5.19 and LLVM clang 15.0.1. Our evaluation on real hardware shows an average runtime overhead of 2.3 % for real-world user applications. Lastly, we systematically analyze 11 state-of-the-art kernel mitigations against data-oriented attacks and illustrate that DOPE is a significant improvement in terms of security with respect to performance.

CCS CONCEPTS

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

ACM Reference Format:
Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. 2023. DOPE: DOmain Protection Enforcement with PKS. In *Annual Computer Security Applications Conference (ACSAC '23), December 4–8, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627106.3627113

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ACSAC '23, December 4–8, 2023, Austin, TX, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0886-2/23/12.
https://doi.org/10.1145/3627106.3627113

1 INTRODUCTION

While memory safety is a well-researched topic, the challenge of finding complete, high-performance mitigations remains unsolved. Memory safety is especially relevant to the kernel since it is a valuable target for memory-corruption attacks. Memory-safety vulnerabilities in the kernel enable privilege escalation, rootkits, and confidential data leakage. Historically, Linux kernel exploits injected instruction sequences into kernel memory and hijacked the control flow to these sequences [31]. In 2009, Linux introduced an in-kernel `W^X` policy enforcing that kernel memory is never writable and executable, preventing code injection attacks [29].

However, attackers could still hijack the kernel control flow to escalate privileges [7, 9, 11, 38, 41]. In an attempt to mitigate control-flow hijacking attacks in the kernel, processor vendors introduced hardware-enforced restrictions for both user-space code execution (Intel SMEP [18] and ARM PXN [59]) and user-space data access (Intel SMAP [18] and ARM PAN [59]) in kernel mode. Subsequently, Linux added support for these features, making exploitation substantially more difficult. To further complicate control-flow hijacking attacks, Control-Flow Integrity (CFI) [1, 2, 21, 31, 57, 58] has been established as the state-of-the-art mitigation. CFI restricts the control flow to a set of transfers to ensure correct program execution, reducing the exploitation surface.

Besides hijacking the control flow, data-oriented attacks are a common attack class [15, 37]. Xiao et al. [81] showed that data-oriented attacks are not only a security concern for user applications but also the kernel. In the kernel, data-oriented attacks corrupt kernel data to indirectly change the control flow and escalate the attacker's privileges. Several mitigations [13, 14, 28, 45, 46, 54, 62, 63, 70, 80, 82] have been proposed to protect against data-oriented attacks. However, their practical deployment is hindered as they are limited in providing robust security guarantees at reasonable performance overhead for multiple sensitive data objects. For example, xMP [62] provides strong security benefits but has a performance overhead of up to 20 % for macro-benchmarks, potentially prohibited for commodity use cases. On the other hand, KDPM [45] has a low performance overhead but offers inadequate security guarantees as it does not mitigate forgery attacks.

In this paper, we present DOmain Protection Enforcement (DOPE), a novel kernel mitigation protecting against data-oriented attacks. By following the principle of least privileges [65], DOPE restricts the memory accesses of threads during kernel space execution. To achieve this restriction, we move sensitive data objects into distinct security domains based on whether access to them could be used in

privilege escalation exploits. Access to sensitive data objects is only granted if the thread has the associated domain's access permission, which DOPE enforces with Intel PKS [22]. DOPE only grants temporary access permission to domains in predefined, trusted code locations. In addition to protecting sensitive data objects, DOPE ensures the integrity of data pointers pointing to these sensitive data objects through ownership at runtime.

To demonstrate DOPE's effectiveness and usefulness, we implement a proof-of-concept and perform a case study. In our case study, we select eight sensitive data objects (*i.e.*, credentials, virtual memory, virtual memory areas, inodes, page tables, filesystem mount, user-accessible pages, and stored registers) susceptible to being used for privilege escalation exploits and protect them with DOPE. The proof-of-concept implementation consists of less than 5000 lines of code on a Linux kernel 5.19 and LLVM clang 15.0.1 [53]. Our implemented LLVM pass analyzes code and inserts domain switches and validation checks into the kernel's binary to ensure DOPE's functionality. We run our DOPE proof-of-concept on Ubuntu 22.04.1 LTS with a recent Intel Alder Lake processor. We evaluate the performance overhead of our proof-of-concept with micro-benchmarks from LMbench [56], showing an overhead of 32 %. In macro-benchmarks from Phoronix Test Suite [61] and SPEC CPU 2017 [25], we observed 2.3 % and 0.4 % overheads.

Lastly, we systematically analyze 11 state-of-the-art kernel mitigations against data-oriented attacks and point out blank spots in the mitigation landscape. We classify all analyzed mitigations against data-oriented attacks based on four techniques: Object monitoring [14, 63, 82], randomization [13, 28], compartmentalization [54, 80], and isolation [45, 46, 62, 70]. We further classify these mitigations according to their overheads and security guarantees. In this systematic analysis, we show that DOPE is a significant improvement in terms of security with respect to performance.

Contributions. The main contributions of this work are:

(1) **DOPE:** We present DOPE, a novel principled kernel mitigation for data-oriented attacks using Intel's new hardware feature PKS to enforce domain protection.
(2) **Proof-of-concept:** We develop a proof-of-concept of DOPE[1] consisting of a Linux kernel extension and an LLVM pass illustrating the feasibility of our approach.
(3) **Case study:** We perform a case study to demonstrate the effectiveness of DOPE in providing robust protection for eight selected sensitive data objects.
(4) **Evaluation:** We evaluate DOPE's security and performance overhead, showing strong security guarantees with an overhead of 2.3 % for macro-benchmarks.
(5) **Systematic analysis:** We examine 11 existing kernel mitigations for data-oriented attacks, identifying gaps in the mitigation landscape. We then show that DOPE provides superior security with respect to performance.

Outline. In Section 2, we provide background and state-of-the-art countermeasures. In Section 3, we present our threat model. Section 4 presents our mitigation DOPE. While in Section 5, we describe our DOPE proof-of-concept, Section 6 performs a case study.

[1] Available https://extgit.iaik.tugraz.at/sesys/dope

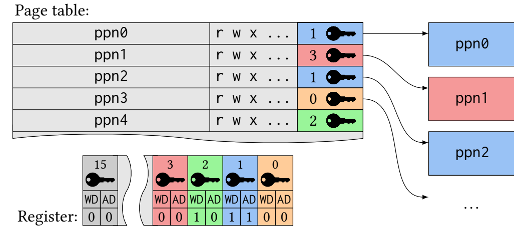

Figure 1: Working prinicple of MPK, where AD and WD stands for access and write disable, respectively. Pages tagged with key 0 are write- and access-permitted, while pages tagged with key 1 are write- and access-prohibited, and pages tagged with key 2 are only write-prohibited.

In Section 7, we discuss DOPE's security and evaluate the proof-of-concept's performance overhead. Section 8 presents a systematic analysis. Lastly, we conclude our work in Section 9.

2 BACKGROUND AND STATE-OF-THE-ART

In this section, we provide background on Memory Protection Keys (MPK), as MPK plays an essential role in the design and implementation process. We then discuss existing user and kernel countermeasures against data-oriented attacks.

2.1 Memory Protection Keys

MPK are a hardware feature to enforce page-level permissions without modifying page-table entries [77]. The page permissions are enforced by tagging pages with a key and changing the permissions of these keys, stored in a dedicated hardware register. Intel features two variants of MPK, one for user- and one for kernel-space pages [39], namely Protection Keys for Userspace (PKU) [77] and Protection Keys for Supervisor (PKS) [22]. Both variants support 16 distinct keys, where a tagged page stores its applied key in its associated page-table entry. Each key comprises two permission bits: Access and write disable. Figure 1 shows MPK's working principle.

The dedicated register used by PKS is MSR `0x6E1` [39], called Protection Key Register for Supervisor (PKRS). Changing a key's permission is done by writing to PKRS. Since for permission changes no page-table walk or TLB flush is required [23], PKS is faster than changing permission bits directly in the page-table entry.

2.2 User Space Mitigations

Previous works [15, 37, 81] showed that data-oriented attacks can actively change the program's control flow by modifying sensitive data objects. Castro et al. [12] proposed Data-Flow Integrity (DFI), which tracks both read and write instructions. DFI enforces that data was not tampered with at runtime. Instead of validating a sensitive data object on every access, Akritidis et al. [3] proposed Write Integrity Test (WIT) that only performs checks on write instructions. WIT employs static analysis to assign a color to each write instruction and their associated sensitive data. Only write instructions with the correct color are permitted, practically preventing illegal write operations. Another approach to protecting sensitive

data is data randomization [5, 6, 10], which encrypts sensitive data in memory with context-specific encryption keys.

To limit the exploitation surface of data-oriented attacks, previous works [36, 60, 68, 78] proposed various isolation mitigations. These mitigations use the Intel user-space implementation of MPK, PKU, to enforce isolation between different entities. Moreover, Intel's PKU is also commonly used in proposals to enforce isolation in unikernels [49, 71] and libOSes [48, 66].

2.3 Linux Kernel Mitigations

The explained user space techniques were adopted into the Linux kernel. Previous works provide DFI for sensitive data objects through object monitoring [14, 63, 82] or compartmentalization [54, 80]. Moreover, randomization-based approaches to the data location and layout were applied to the kernel [13, 28]. Other proposals [45, 46, 62, 70] isolate sensitive data objects from untrusted code.

As we later show in Section 8, existing mitigations have limitations in providing robust security guarantees, reasonable performance overhead, or protection for multiple sensitive data objects. To address these limitations, we propose our mitigation approach DOmain Enforcement Protection (DOPE). Our case study demonstrates that DOPE's proof-of-concept protects eight selected sensitive data objects with a reasonable runtime overhead. Compared to existing countermeasures, DOPE offers protection for multiple objects with strong security guarantees and lower performance overhead. This significant enhancement in security relative to performance underscores the superiority of our approach.

3 THREAT MODEL

We assume an attacker can execute code in user space and has an arbitrary read-and-write primitive in the kernel accessible through the syscall interface without violating control-flow integrity. This aligns with the threat model of existing kernel defenses [13, 14, 28, 45, 46, 54, 62, 63, 70, 80, 82]. We do not specify the underlying flaw that leads to this primitive. We also assume modern kernel defense mechanisms such as Secure Boot, the `W^X` policy, KASLR [30], SMEP, and SMAP [18] are enabled, along with a CFI scheme [26, 32, 33, 57].

Attack vector. Since we assume that the enabled CFI scheme prevents control-flow hijacking attacks [7, 9, 11, 38], attackers focus on manipulating non-control data to elevate privileges. Such non-control data may include objects that contain information about privilege levels, like credential or inode objects, or information obtained from page permissions, such as page tables.

Out of scope. Although we acknowledge the presence of various types of attacks, such as side-channel [34, 51], microarchitectural [35, 44], and software fault injection [19, 69], as well as malicious hypervisors and operating systems, these are out of scope.

4 DOMAIN PROTECTION ENFORCEMENT

DOmain Protection Enforcement (DOPE) is a principled mitigation for data-oriented attacks by protecting sensitive data objects from being exploited for privilege escalations. DOPE achieves this protection by adhering to the principle of least privilege [65]. During kernel space execution, DOPE restricts access to sensitive data for each thread based on their access permissions (cf. Section 4.1). To

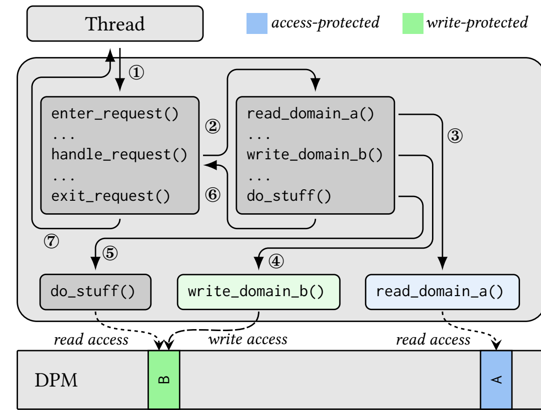

Figure 2: Access restriction of DOPE, where a thread invokes kernel space execution (①, ⑥), handles the request (②) and accesses sensitive data objects (③, ④, ⑤).

achieve this restriction, it places each sensitive data object into distinct security domains. Access to these objects is only granted if the thread has the associated domain permission (cf. Section 4.2), which DOPE enforces using Intel PKS (cf. Section 4.3). DOPE only grants temporary domain access permission in predefined, trusted code locations. We define code as trusted (cf. Section 4.5) if all accessed pointers are integrity ensured and, hence, trusted (cf. Section 4.4). Thus, DOPE effectively thwarts attackers from utilizing sensitive data for privilege escalation exploits in case of an exploitable bug.

4.1 Restricted Access Permissions

DOPE provides a fine-grained permission setting for the security domains by prohibiting access or writing to the domain's data. On each kernel execution request entry, the thread's access permissions are set to restricted. The restricted access permissions list what domain is prohibited from being accessed or written to. They are defined before compile time by the system developers and enforced at runtime by DOPE. For example, the restricted access permissions in Figure 2 are: Prohibit access to domain `A` and prohibit write to domain `B`.

4.2 Sensitive Data Access

When a thread handles an execution request in kernel space, it is restricted from accessing sensitive data objects in accordance with its access permissions. A thread is only permitted to access such objects if it has the corresponding domain's permission, which DOPE temporarily grants by performing domain switches.

For instance, Figure 2 shows DOPE's access restriction. After a thread invokes kernel space execution ①, `enter_request` is called, which sets the thread's access permissions to restricted. While having restricted access, the thread handles the invoked request ②. In `read_domain_a` ③, the thread switches to domain `A` for every read access and, hence, acquires temporary read permission. The `write_domain_b` function ④ performs domain switches to gain domain `B` temporarily write permission. Since we define domain `B` as readable, the function `do_stuff` ⑤ is legally permitted to do so. Finally, `exit_request` ⑥/⑦ finishes the kernel execution request.

DOPE interprets any access from a thread that does not have domain permission as an exploit and terminates its execution, e.g., when the thread accesses domain A or writes to B within do_stuff.

DOPE supports two kernel execution requests: The first originates on thread creation and ends on termination. In between, the thread executes code in the kernel and user space, where switching between kernel and user space does not alter its permissions. The second originates on an asynchronous interrupt when the disrupted thread is in kernel space. Considering this execution request is crucial, otherwise, an attacker could perform the **elevated permission interrupt** attack, as we later describe in Section 7.1.

4.3 Enforcing Domain Protection with PKS

DOPE utilizes one PKS key per security domain and tags the pages of each security domain with their respective key. This tagging enables Intel PKS to enforce domain access restrictions. Any attempt to violate the access permissions of the tagged pages results in a fault, which DOPE interprets as an exploit attempt. Consequently, DOPE thwarts the exploit attempt, effectively mitigating the attack.

DOPE configures each AD and WD bit in the per-thread PKRS according to the access permission restrictions of the security domain, where AD and WD stand for access and write disable. In the example of Figure 2, DOPE sets AD and WD in the PKRS for domain key A to prohibit access to domain A. For domain key B, DOPE sets WD and resets AD to prevent writing to domain B. In order to enforce these restricted access permissions, the thread acquires this PKRS on each kernel execution request entry.

To perform a domain switch, DOPE alters the permission bit of the target domain key for the current thread. As a result, the current thread gains read or write access to the desired domain. The function read_domain_a in Figure 2 switches domains each time it reads data from domain A. It does so by modifying the AD bit in the PKRS of domain key A, where resetting the bit grants access permission and setting the bit removes the permission. Similarly, in write_domain_b, DOPE grants write permissions temporarily by resetting and setting the WD bit in the PKRS for domain key B.

Maintaining high performance is essential since the kernel is the lowest software abstraction layer. Previous research [60, 68] has demonstrated a direct correlation between the number of domain switches and the performance overhead. Therefore, DOPE aims to minimize domain switches by providing three variants of domain protection enforcement using Intel PKS. These variants differ in their level of spatial granularity, offering flexibility for system developers for their specific use cases. By minimizing domain switches, DOPE helps optimize system performance while providing strong security guarantees against data-oriented attacks.

4.3.1 Entire data object protection. The first protection variant of DOPE involves protecting an entire data object. In this approach, the page containing the data object is protected with PKS by tagging it with the associated domain key. Any data access that violates the domain's permissions is prohibited. This applies to both sensitive and non-sensitive members of the object. Consequently, to access a protected data object, the current thread must have the appropriate domain permissions and switch domains as needed, regardless of whether the accessed member is sensitive or non-sensitive. This approach is best suited when the data object consists mainly or entirely of sensitive data members. Since switching domains incurs a performance overhead, it may not be optimal if the object contains a mix of sensitive and non-sensitive members.

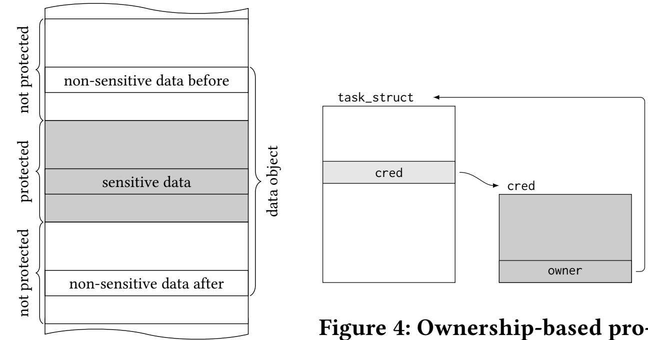

Figure 3: Data layout of sensitive data protection.

Figure 4: Ownership-based protection is employed to protect the sensitive pointer to cred within task_struct.

4.3.2 Shadow memory protection. DOPE introduces shadow memory protection for data objects containing a mix of sensitive and non-sensitive data members. In such cases, the sensitive data members are duplicated on allocation, and the duplicated data are protected by tagging its page with the associated domain key. Consequently, the data object stores a pointer to the duplicated data. DOPE synchronizes sensitive and duplicated data every time sensitive data is written. With shadow memory protection, DOPE checks for each sensitive data read to see if the sensitive data matches the duplicated data. If sensitive and duplicated data differ, DOPE detects this as an exploitation attempt and terminates the thread's execution. To prevent an attacker from overwriting the pointer to the duplicated object, DOPE ensures the pointer's integrity, as we later show in Section 4.4.

This approach is particularly suitable for data objects with a mix of sensitive and non-sensitive data members. However, the runtime overhead associated with ensuring pointer integrity during read access to sensitive data members may make it less suitable for scenarios involving frequent access to such data members.

4.3.3 Sensitive data protection. Our proposed third variant provides a more efficient way to protect data objects containing both many non-sensitive data members and frequent access to sensitive data members. This variant enforces a specific data object layout, where all sensitive data members are placed on a PKS-protected page. On the other hand, non-sensitive data members are placed on an adjacent, non-protected page. As a result, accessing sensitive data members is protected by DOPE, while non-sensitive data members can be accessed without restrictions. This approach ensures that sensitive data is protected while minimizing the performance overhead. The only downside is that adapting the Linux kernel to accommodate this specific data layout requires effort.

The object layout of this variant is depicted in Figure 3, where the data object spans three contiguous pages. The sensitive data is safeguarded by Intel PKS and is only present on the middle page (grey). To prevent sensitive and non-sensitive data from ever coexisting on the same page, a dummy page is inserted between

```
1  /* get ext4 inode */
2  struct inode *ext4_iget(){
3    struct ext4_inode *ei;
4    struct inode *inode;
5    ...
6    inode = dentry->inode;
7    inode->i_uid = i_uid;
8    inode->i_gid = i_gid;
9    ei->i_data[blk] = data;
10   ...
11   return inode;
12 }
```

Listing 1: `ext4_iget` reads `inode` from its owner `dentry`, and modifies its sensitive members `i_*id`.

```
1  /* get ext4 inode */
2  struct inode *ext4_iget(){
3    struct ext4_inode *ei;
4    struct inode *inode;
5    ...
6    inode = dentry->inode;
7  + owner_check(dentry, inode);
8  + enter_inode_wr();
9    inode->i_uid = i_uid;
10   inode->i_gid = i_gid;
11 + exit_inode_wr();
12   ei->i_data[blk] = data;
13   ...
14   return inode;
15 }
```

Listing 2: Modified and trusted `ext4_iget`.

the end of non-sensitive data and the beginning of sensitive data, as well as between the end of sensitive data and the beginning of non-sensitive data. As a result, this approach entails a memory overhead of two pages per protected data object.

It is feasible to reduce the memory overhead by grouping sensitive data members from distinct data objects on the PKS-protected page while storing non-sensitive members on the adjacent page. Although implementing this memory layout necessitates even more engineering efforts to modify the Linux kernel, it presents a possible direction for future work.

4.4 Pointer Integrity through Ownership

DOPE ensures the integrity of data pointers to sensitive data objects by enforcing ownership, where access to the sensitive data object is restricted to its owner. The sensitive data object comprises the address of its owner object, and DOPE modifies the kernel to perform an owner validation before accessing a sensitive pointer. This validation checks whether the correct owner object is accessing the sensitive data object, thereby preserving the pointer's integrity.

DOPE utilizes two checks for owner validation: The first verifies if the sensitive object is tagged with the correct domain key. In contrast, the second compares if the owner address stored in the sensitive data object matches the owner object address. DOPE interprets a failure of either check as an exploitation attempt. Figure 4 exemplifies the credential struct `cred` with its owner `task_struct`. The validation procedure confirms that `cred` is tagged with the appropriate domain key and verifies whether `owner` and `task_struct` are identical. If the credential is suitably tagged, it is impossible to manipulate the `owner` member, thereby ensuring ownership.

In our ownership approach, we devise a reliable solution to handle aliasing, where a sensitive object is shared among multiple owners. To achieve this, we store the address of both the sensitive object and its owner object in a hashtable. On validation, DOPE checks whether the sensitive object with its owner is stored in the hashtable. This ensures that each owner is verified and eliminates any chances of ownership forgery of sensitive objects with multiple owners. Additionally, we tag the hashtable with a write-protected domain key, preventing any potential tampering attempts.

4.5 Trusted Code

DOPE imposes three constraints on trusted code. Firstly, only pointers whose integrity is ensured (cf. Section 4.4) are dereferenced in the trusted code area. Secondly, the memory pointed to by these pointers must be tagged with the domain to which the trusted code is temporarily granted access permission. Thirdly, objects are only permitted to be dereferenced with a fixed offset.

For instance, Listing 1 shows a code snippet where the `inode` is read from its owner `dentry`, and its sensitive data members `i_*id` are written. DOPE performs an owner validation to ensure the first constraint, as seen in Line 7 of the modified code snippet in Listing 2. With the trusted pointer, the trusted code part is between Lines 9 and 10, fulfilling all three constraints previously described. DOPE enters the write domain for inodes in Line 8 and exits in Line 11 to legally write to these sensitive data members.

Consider granularity. The granularity with which DOPE handles domain switches can be adjusted, influencing the performance overhead directly. However, this adjustment represents a trade-off: Finer granularity increases security at the expense of performance, while coarser settings can improve performance with potential security degradation. Determining the appropriate granularity, therefore, requires a thorough assessment of the balance between security and performance.

5 IMPLEMENTATION

In this section, we highlight our DOPE proof-of-concept implementation in the Linux kernel and LLVM pass [53]. We employed Linux version 5.19, the latest stable version when we started this work. At the time of writing, the Linux kernel did not have support for Intel PKS. Therefore, we implement secure PKS for the Linux kernel regarding data-oriented attacks. We then implement an LLVM pass to perform code analysis and automate function insertion. Finally, we implement our DOPE proof-of-concept.

Direct physical mapping and SLUB. Since DOPE requires permission setting at the page level granularity, we first break down the Direct Physical Mapping (DPM) from huge pages into 4 kB pages [24]. We then extend the buddy allocator to allocate pages tagged with a desired PKS key that defines the domain of the associated page. Moreover, we extend the functionality of the SLUB allocator to obtain an allocator that returns only data objects tagged with the desired domain. Our implementation extends the functionality of `kmem_cache` to provide the `kmem_dope_cache` object. Hence, each domain allocates tagged objects via `kmem_dope_cache`.

Sensitive state data. Since each thread can be in a different domain at a time, the PKRS has to be stored and restored on every context switch. The PKRS value of a currently not scheduled thread is stored in memory. Storing the PKRS in an unprotected area poses the risk of an attack. With an arbitrary write primitive, an attacker could overwrite the stored PKRS and gain control over the hardware PKRS. To prevent this illegal control gain, we implement a secure way to store the PKRS. DOPE protects the stored PKRS with a write-prohibited security domain, where only limited and trusted locations are permitted to write to. We explain the sensitive state protection against attack scenarios in more detail in Section 7.1.

Thread creation. On thread creation, we allocate a sensitive state object and store a pointer to it in `thread_struct`. We then set

its stored PKRS to restricted. Hence, the thread starts after it is first scheduled with restricted access permissions. DOPE protects the sensitive state objects with our ownership protection to prevent potential corruption attacks of pointers to sensitive state objects.

Domain switch. In DOPE, whenever a domain switch happens, it changes the permission bit of the target domain key for the current thread. This allows the current thread to gain read or write access to the desired domain temporarily. The permission bit is changed by writing to the MSR `0x6E1` with the `wrmsr` instruction.

Asynchronous interrupt. On asynchronous interrupt entry, DOPE stores the current PKRS in a stack-like structure within the write-protected sensitive state object, where the PKRS is read with instruction `rdmsr` from MSR `0x6E1`. Subsequently, the access permissions of the thread are set to restricted. On interrupt exit, DOPE restores the stored PKRS to obtain the interrupted permissions.

Instrumentation. We implement an LLVM pass that performs two crucial tasks: Code analysis and function insertion.

To protect sensitive data objects with either the entire data (cf. Section 4.3.1) or sensitive data (cf. Section 4.3.3) variant, our LLVM pass analyzes the code and identifies all read and write locations of the sensitive data. We then manually verify the analysis output to ensure domain switches are inserted at the appropriate locations. This combined approach of automatic analysis and manual verification provides the benefits of both methods. While automatic analysis helps identify difficult-to-find domain switch locations, manual verification ensures efficient domain switch placements and upholds constraints of trusted code. Additionally, the LLVM pass automatically inserts owner validations on each sensitive data object's read access from its owner object.

To estimate the manual effort required by our proof-of-concept, Listing 1 shows the function `__ext4_iget`, where `inode` is write-protected and `dentry` is its owner. The code analyzer outputs that between Lines 7 and 8 all sensitive data members (*i.e.*, `i_*id`) are written. Hence, we manually insert an `enter_inode_wr` before Line 7 and `exit_inode_wr` after Line 8, where `*_inode_wr` enters and exits the inode write domain. Additionally, our LLVM pass automatically inserts an owner validation of the `inode` from its owner object `dentry`. Listing 2 shows the total instrumented code.

For sensitive data objects protected with shadow memory (cf. Section 4.3.2), our LLVM pass inserts synchronizations automatically for every write and validations for every read. The synchronization functions synchronize the sensitive and duplicated data. For validation, DOPE first performs an owner validation. DOPE then checks if the sensitive data has been modified illegally. If at least one of the two is true, DOPE detects the corruption attempt and terminates the thread's execution.

Trusted code. In Appendix 10, we provide measures to address any implementation issues while ensuring our code adheres to the trusted code constraints.

6 CASE STUDY

We demonstrate the effectiveness and usefulness of DOPE by protecting eight sensitive data objects (cf. Section 6.1) from malicious accesses that violate restricted access permissions (cf. Section 6.2). For each sensitive data object, DOPE enforces domain protection with one of its three protection variants (cf. Section 6.3). Additionally, DOPE ensures the integrity of pointers to sensitive data objects by enforcing ownership (cf. Section 6.4) at runtime. Lastly, we discuss the manual efforts of our case study (cf. Section 6.5).

Table 1: Applied protection variant for our sensitive data objects.

Variant	Sensitive data objects								
	User-accessible pages	Credentials	Inodes	Page tables	Virtual memory areas	Virtual memory	Filesystem mount	Stored registers	Sensitive state
4.3.1 Entire	●	●	○	●	○	○	○	●	●
4.3.2 Shadow	○	○	○	○	●	●	●	○	○
4.3.3 Sensitive	○	○	●	○	○	○	○	○	○

● Applied ○ Not applied

6.1 Sensitive Data Objects

All restriction-based mitigations against data-oriented attacks face a fundamental question of which data objects to protect. The more sensitive data objects a mitigation adequately protects, the higher the security and performance overhead. As the number of protected objects increases, the security benefit of additional objects decreases as exploiting the system becomes substantially more difficult. How to set a trade-off between security and performance depends on the use case. For our case study, we demonstrate that DOPE can be deployed to protect eight sensitive data objects with a reasonable performance overhead. More precisely, we protect user-accessible pages, stored registers, credentials, inodes, page tables, virtual memory areas, virtual memory, and filesystem mount, with the former two discussed in detail and the remaining six in Appendix 11. Crucially, our approach protects more objects with strong security guarantees while imposing a lower runtime overhead than existing countermeasures [13, 14, 28, 45, 46, 54, 62, 63, 82].

User-accessible pages. To our knowledge, we are the first to consider user-accessible pages via the DPM in their threat model for data-oriented attacks. User-accessible pages are either mapped in any user space or read from the disk and remain in kernel space. These pages may either be from the current or another process's user space, including high-privilege processes. If left unprotected, attackers can perform DPM-FPATCH (cf. Appendix 12).

Stored registers. An attacker can convert an arbitrary read-and-write to a register manipulation primitive. To achieve this, the attacker enforces a victim thread to preempt, causing the registers to be stored in memory [72]. Consequently, the attacker corrupts this memory location. When the thread resumes, the registers are restored from the corrupted memory, granting the attacker control over them. Suppose the preemption happens when the victim thread has access permission to a domain, the attacker can manipulate the victim's registers to perform an access, bypassing the applied mitigation if not protected. However, unlike existing schemes [13, 14, 28, 45, 46, 54, 62, 63, 82], DOPE provides protection for stored registers during preemption, effectively preventing such attacks.

6.2 Restricted access permissions

DOPE provides a fine-grained permission setting that applies to our sensitive data objects, which comprise nine objects, including sensitive state. Our case study works with three security domains:

- *Default:* Permits read and write to data[2].
- *Write-protected:* Permits read and prohibits write to data.
- *Access-protected:* Prohibits read and write to data.

We place the stored register to the *access-protected* domain because an attacker could otherwise access confidential data, potentially bypassing DOPE. Moreover, we set the user-accessible page to be *access-protected* because an attacker could otherwise leak confidential data from a high-privilege user process. The other sensitive data objects are set to *write-protected* because: First, the data can be legally read via syscalls, *i.e.*, credentials, inodes, and filesystem mount. Second, read access to these objects cannot be exploited for privilege escalation, *i.e.*, page tables, `vm_page_prot` (virtual memory areas), `pgd` (virtual memory), and sensitive state.

While it may seem that assigning each sensitive data object to an individual security domain would increase security, our goal is not to isolate domains from each other but to isolate them from attackers with strong capabilities. Hence, we group sensitive data objects with the same access permissions to one security domain. This approach ensures that highly capable attackers cannot gain access to sensitive data that violates our restricted access permissions.

6.3 Enforcement variant

In this section, we demonstrate the feasibility and usefulness of the protection variants provided by DOPE, each of which is well-suited for specific sensitive data objects. Table 1 presents the applied variant for each object. In our case study, we protect credentials, user-accessible pages, page tables, stored registers, and sensitive states with entire data object protection (cf. Section 4.3.1) as they comprise mostly or entirely of sensitive data members. For virtual memory areas, virtual memory, and filesystem mount, we utilize shadow memory protection (cf. Section 4.3.2) as these objects contain a combination of sensitive and non-sensitive data. In contrast, since inodes contain many non-sensitive data members, such as locks and modification time, and their sensitive data is accessed frequently, sensitive data protection (cf. Section 4.3.3) is more suitable.

6.4 Ownership

DOPE ensures ownership of sensitive data objects to prevent forgery attacks. When accessing a data pointer to a sensitive data object, DOPE performs an owner validation to determine if the correct owner is accessing the data object. DOPE stores the address of its owner object in the sensitive data object, as shown in Table 2.

We identify seven sensitive data objects susceptible to forgery attacks. For the shadow data (virtual memory areas, virtual memory, and filesystem mount), sensitive state objects, and stored registers, DOPE stores the owner's address to bind the object to the owner uniquely. Neither page tables nor user-accessible pages can be forged as the higher-level page table is protected with the *write-protected* domain. In case an attacker tries to manipulate page-table entries, DOPE detects and prevents the tampering attempt. The highest page-table level, `pgd` (*i.e.*, virtual memory), can also not be forged, as it is also protected with the *write-protected* domain.

[2] Crucially to note, the tagged PKS key does not override permission bits, such as writable bit.

Table 2: Owner of each sensitive data objects.

Owner	Sensitive data objects								
	User-accessible pages	Credentials	Inodes	Page tables	Virtual memory areas	Virtual memory	Filesystem mount	Stored registers	Sensitive state
`task_struct`	-	●	○	-	○	○	○	●	●
`dentry`	-	○	●	-	○	○	○	○	○
`vfsmount`	-	○	○	-	○	○	●	○	○
`vma_struct`	-	○	○	-	●	○	○	○	○
`mm_struct`	-	○	○	-	○	●	○	○	○

● Owner ○ Not owner

Both credentials and inodes may have multiple owners. In the case of credentials, they are shared among threads within a process. To ensure ownership, DOPE stores the `task_struct` address of the initial thread within the credential. For additional `task_structs`, DOPE stores their address combined with the credentials in a dedicated hashtable. In the case of inodes, the dentry is designated as the owner since it links the user accessibility file to its inode [73]. Although inodes are not typically shared between different dentries, hardlinks result in multiple dentries sharing the same inode. Hence, the inode stores the dentry's address as its owner, and in case of a hardlink, both the dentry and inode are stored in a hashtable. Both hashtables, for credentials and inodes, are *write-protected.*

6.5 Instrumentation of our Case Study

Manual effort. To address the manual efforts, we followed three steps. Firstly, we modified the sensitive data objects by adjusting their layout to match the protection variant and adding a member variable to store the owner object's address. We also separated the `rcu_head` member from the credential by dynamically allocating the `rcu_head` and storing a pointer within the credential. Secondly, we replaced allocation and freeing of sensitive data objects with `kmem_dope_cache`. Thirdly, as explained in Section 5, we inserted domain switches based on the LLVM pass's code analyzer output. To ensure optimal performance without undermining security, precisely during multiple sensitive data accesses, we grouped these accesses. We then inserted a single domain switch both at the start and end of these grouped accesses.

False negatives. With proper domain switches in place, access to sensitive data is granted in trusted code locations. In cases where we would have missed inserting a domain switch (false-negative), DOPE would mistakenly identify the access as an exploitation attempt, as the current thread does not have access permissions. However, we did not encounter any such occurrences during our evaluation (cf. Section 7.2) and testing with LTP [27].

7 EVALUATION

We assess DOPE's security before evaluating our proof-of-concept on real hardware with various benchmarks [25, 56, 61].

7.1 Security Discussion of DOPE

This section demonstrates the robustness of DOPE even in the presence of a powerful attacker, as described in Section 3.

Sensitive data objects. If access to sensitive data objects violates the restricted access permissions, Intel PKS triggers a fault, which DOPE interprets as an exploitation attempt and terminates the thread's execution. Hence, it is not possible to access sensitive data without proper access permissions, *i.e.*, in trusted code.

Ownership. An attacker may aim to manipulate pointers to sensitive data objects protected with ownership. If the attacker tampers with the sensitive data pointer and points it to memory tagged with the wrong or no domain, DOPE detects it on owner validation. If the memory is tagged with the correct domain, the attacker cannot manipulate the owner member because they do not have write permissions to the domain-protected data. Additionally, if the attacker overwrites the pointer with an existing high- or low-privilege object correctly tagged, DOPE detects the manipulation on owner comparison during validation. Therefore, it is not possible to forge a sensitive data object that passes owner validation.

Asynchronous domain-protected data access. Compared to previous works that protect memory by mapping it as read-only [14, 70], PKS sets permissions on a logical core granularity. Even if a thread currently has access permission to a domain, another thread from another logical core does not. Therefore, asynchronous access to sensitive data is not possible with Intel PKS by design.

Elevated permission interrupt. Due to the preemptive nature of the Linux kernel, an asynchronous interrupt may occur while a thread has access permission to a domain. During the interrupt, the disrupted thread accesses data via untrusted pointers. An attacker could carefully craft the untrusted pointers to force the thread to perform domain data access. However, DOPE protects against this attack scenario by storing the access permissions (PKRS) and setting it to restricted on interrupt entry. On interrupt exit, DOPE restores the access permissions and continues execution.

Sensitive state protection. DOPE protects all sensitive state data, comprising the stored PKRS values, by placing it in the *write-protected* domain. Pointers to the objects are integrity-ensured using our ownership approach. Only four routines are permitted to write to this data: Thread creation, context switch, and asynchronous interrupt entry and exit. During these operations, the thread validates ownership of the sensitive state object and temporarily grants write access for storing the PKRS to the object. This robust protection ensures that attackers cannot tamper with the stored PKRS.

`kmem_dope_cache` manipulation. An attacker may manipulate the state of the `kmem_dope_cache` object in order to return an attacker-controlled address. Therefore, the attacker can force the `kmem_dope_cache` object to return an object that is not tagged. DOPE protects against this attack by checking whether the returned address is tagged with the correct domain key after the allocation. DOPE interprets a domain key mismatch as an exploit attempt.

Arbitrary use-after-free. An arbitrary write primitive can be converted to an arbitrary use-after-free primitive by tampering with unprotected memory to obtain `kfree(sens_obj_in_use)`. On the next allocation, `sens_obj_in_use` may be returned, allowing it to be overwritten with either low- or high-privileged data. One attack scenario is overwriting credentials owned by a low-privilege thread with high-privilege credentials. Another scenario is to overwrite an inode owned by a high-privilege file with low-privilege metadata. If left unprotected, both scenarios would lead to privilege escalation. Notably, this attack closely resembles DirtyCred [50].

However, during allocation, our `kmem_dope_cache` overwrites the owner member of the sensitive data object with the new owner. When the actual owner first accesses the object, DOPE performs an owner validation, which fails since the owner was overwritten during allocation. DOPE interprets this attack scenario as an exploitation attempt and terminates the thread's execution.

Multi-ownership. DOPE employs a two-step validation before adding the sensitive data object and new owner to the hashtable. Firstly, it validates the old owner and sensitive data object; secondly, it validates that the new owner is not already present in the hashtable. If either of these validations fails, DOPE terminates execution. As a result, an attacker can neither forge ownership nor perform **arbitrary use-after-free** with the new owner.

Physical memory. Attackers may tamper with the DPM to potentially bypass DOPE's protection of sensitive data objects. However, since all sensitive data objects and their permissions are directly accessed and set on the DPM, it is not possible to use the DPM for bypassing. Additionally, it is not possible to corrupt the permissions of sensitive data objects as the page tables containing the tagged domain key are protected by the *write-protected* domain.

Pointer-to-pointer attack. DOPE provides a robust mechanism for ensuring the integrity of pointers to sensitive data objects through ownership. Specifically, DOPE restricts pointer access to sensitive data objects (e.g., `cred`) only to their respective owner objects (e.g., `task_struct`). Although an attacker can manipulate a pointer to the owner object in an attempt to bypass DOPE, it is important to note that the sensitive pointer, such as the `cred` pointer, must still pass owner validation from its forged owner pointer, such as `task_struct`. Additionally, the attacker must find a valid execution path that does not cause a kernel panic due to the corrupted owner pointer. In summary, while a pointer-to-pointer attack is technically feasible, executing it may not be practical.

Confused deputy attack. A confused deputy attack [47] aims to trick a high-privilege function into performing access, violating the restricted access permissions. DOPE's trusted code design drastically reduces the exploitation surface of confused deputy attacks. Other isolation-based schemes [14, 45, 46, 63, 82] are vulnerable if an attacker corrupts a non-protected pointer that is dereferenced within trusted code. This allows the attacker to convert a low-privilege arbitrary read-and-write primitive to a high-privilege one. However, this conversion is not possible with DOPE's trusted code constraints, significantly improving protection against confused deputy attacks compared to existing countermeasures.

Even though there are some scenarios where DOPE is vulnerable to a confused deputy attack, the system's trusted code constraints make it challenging for attackers to exploit any vulnerabilities. Two possible attack scenarios are identified, where an integrity-ensured pointer temporarily stored on the stack could be corrupted or where the kernel stores the argument that will be written to the sensitive data object on the stack, which a TOCTTOU attack could exploit.

Although DOPE may have limitations regarding stack tampering, we view it as an opportunity for future research to enhance

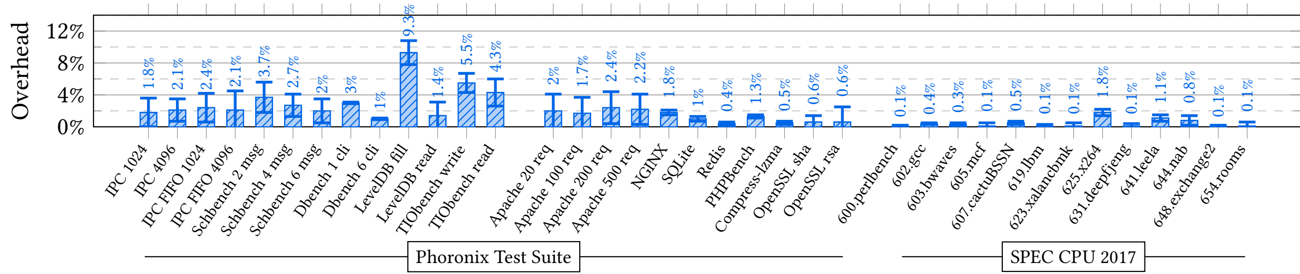

Figure 5: Overhead of macro-benchmarks.

the protection of isolation-based schemes against confused deputy attacks.

Scalability. In our case study, we demonstrate the effectiveness of DOPE, as it protects eight sensitive data objects from exploitation. We firmly believe that these eight objects form an appropriate set for protection. Moreover, the flexibility of DOPE allows for expansion to safeguard additional objects. While this presents a standalone research question [64, 70] that requires further investigation, we see it as a promising avenue for future work.

Manual effort. Any kind of manual effort by developers may unintentionally introduce implementation bugs or instabilities, making it susceptible to Denial-of-Service (DoS). However, this is a fundamental issue in software development, particularly kernel development, which is one of the primary motivations behind DOPE. DOPE cannot eliminate the possibility of a developer introducing a security bug, but DOPE alleviates the security risk posed by the bug. To address the concern of introducing bugs while implementing the DOPE policies, we follow the state-of-the-art kernel software testing with the Linux Test Project (LTP) (cf. Section 6.5), which helped us attain a high code coverage and ensure the robustness of our system against potential DoS.

Comparison to PKU-based approaches. In comparison to approaches [36, 60, 67, 68, 78] based on Intel's PKU, DOPE addresses and resolves several kernel challenges inherent in PKS: Firstly, the kernel handles system events (*i.e.*, exceptions, interrupts, and context switches), which attackers with memory can exploit write primitives to tamper with the PKS state. Our solution, detailed in Sections 5 and 6.1, introduces protections, preventing potential tampering attempts during these events. Secondly, the kernel manages low-level page permission handling (e.g., manipulating access permissions of pages, including MPK keys), posing a potential DOPE bypass. To counter this, Section 6 describes how we fortified page tables using DOPE, effectively eliminating the risk of page table tampering and subsequent DOPE bypasses. Thirdly, the kernel's memory allocator, the buddy allocator, recycles physical memory pages, requiring special handling. In Appendix 10, we describe how we adapted the allocator. Lastly, the Linux kernel combines sensitive and non-sensitive data within the same data structures. In Section 4.3, we propose three enforcement techniques, *i.e.*, entire data object protection, shadow memory protection, and sensitive data protection, to securely and efficiently protect sensitive data.

Call gates. A primary security concern with PKU-based systems arises from the `wrpkru` instruction responsible for altering access rights. A malicious thread could execute this instruction, thereby changing its access rights [17, 79], e.g., using the kernel as a confused deputy. In response, researchers have introduced various countermeasures [60, 67, 68, 78]. These include advanced techniques for code/binary analysis and the integration of a call gate. Similarly, the PKS system uses the `wrmsr` instruction for modifying access rights. Kernel threads, by default, have unrestricted access to execute `wrmsr`. To fortify against this potential threat, DOPE has been designed to leverage call gates.

7.2 Performance Evaluation

We evaluate our DOPE proof-of-concept implementation's binary size, compile time, and performance overhead, where Appendix 13 shows the detailed results. We perform micro-benchmarks with LMbench [56], and macro-benchmarks with Phoronix Test Suite [61] and SPEC CPU 2017 [25]. Our benchmark CPU is Intel i7-1260P. We run Ubuntu 22.04.1 (kernel 5.19) as the Linux distribution.

Micro-benchmarks. We use LMbench to evaluate the latency and bandwidth overhead of our proof-of-concept. We consider the baseline kernel version 5.19, DOPE-light, and DOPE, where DOPE-light protects the same data objects as our case study DOPE, except for user-accessible pages. We include DOPE-light to highlight the overhead caused by protecting user-accessible pages. To achieve stable results, we run each benchmark 80 times and compute the mean and standard deviation, with the results shown in Table 4. We compute the total overhead by averaging over all overheads, resulting in an overhead of 32 % for DOPE and 17 % for DOPE-light.

Phoronix Test Suite macro-benchmarks. Our benchmarks from Phoronix Test Suite split up into stress tests and real-world applications, as shown in Figure 5. Among the stress tests are one inter-process communication, one kernel scheduler, two filesystem, and one threaded I/O benchmarks, while among the real-world applications are two web-server, two database, and four user application benchmarks. The average performance overhead of the Phoronix Test Suite macro-benchmarks is 2.3 %.

SPEC CPU 2017. We perform speed benchmarks of SPEC CPU 2017, as shown in Figure 5. The measured overheads of the macro-benchmarks are all below 1.8 %, consistent with the user application benchmarks from the Phoronix Test Suite. The overall overhead is calculated to be 0.4 % when averaging all the results.

Table 3: Systematic overview of state-of-the-art mitigations against data-oriented attacks in the Linux kernel.

Mitigations	Technique	Protection Targets									Overhead
		Credentials	Virtual memory	Virtual memory areas	Inodes	Page tables	Filesystem mount	Other non-control data	User-accessible pages	Stored registers	
PrivGuard [63]	Monitoring	○	○	-	-	-	-	-	-	-	⧖
AKO [82]	Monitoring	○	-	-	-	-	-	-	-	-	⧖
PrivWatcher [14]	Monitoring	●	●	-	-	-	-	-	-	-	⧗ [1]
SALADS [13]	Randomization	●	-	-	●	-	-	● [2]	-	-	⧗
PT-Rand [28]	Randomization	-	-	-	-	●	-	-	-	-	⧖
Mondrix [80]	Compartmentalization	-	-	-	-	-	-	●	-	●	⧗ [1]
HAKC [54]	Compartmentalization	○	○	●	○	●	○	●	○	◐	⧗
KDPM [45]	Isolation	○	-	-	-	-	-	-	-	-	⧖ [1]
KPRM [46]	Isolation	○	-	-	-	-	-	○	-	-	⧗
KENALI [70]	Isolation	◐	●	◐	◐	●	◐	●	-	●	⧗
xMP [62]	Isolation	◐	●	-	-	●	-	● [3]	-	-	⧗
DOPE	Isolation	●	●	●	●	●	●	-	●	●	⌛

● Strong protection ◐ Partial protection ○ Insufficient protection - Not protected
⧖ Low overhead ⌛ Reasonable overhead ⧗ High overhead
[1] Not tested on hardware [2] Non-sensitive data [3] User space data

8 SYSTEMATIC ANALYSIS

In this section, we systematically analyze existing mitigations against data-oriented attacks with a threat model aligned with ours. Table 3 illustrates the analysis results.

We categorize these mitigations into four techniques: Object monitoring, randomization, compartmentalization, and isolation. Moreover, we classify them based on the performance overhead they introduce. However, directly comparing DOPE's overhead with existing countermeasures is not possible as we have no access to the source code, kernel versions, kernel configurations, and hardware setup from these countermeasures. All of these factors influence the benchmark outcomes. As a result, any performance data reported in works should be treated as an estimate when comparing the performance across mitigations. We strive to perform a fair comparison with the following classification scheme.

We first consider macro-benchmark results as the primary criterion. If no macro-benchmarks are available, we rely on micro-benchmark results. For low overhead ⧖, macro-benchmarks are below 1 % or the micro-benchmarks are below 5 %. For reasonable overhead ⌛, we set the boundaries between 1 % to 3 % for macro-benchmarks and 5 % to 25 % for micro-benchmarks. For high overhead ⧗ mitigations have an overhead above 3 % for macro-benchmarks or 25 % for micro-benchmarks.

Monitoring. PrivGuard [63] protects credentials and the pgd by monitoring their changes and only permits its modification for high-privilege syscalls, e.g., `sys_set*id`. This monitoring involves duplicating these objects at the beginning of the syscall and checking them at both the beginning and end. However, an attacker may perform two attack scenarios. Firstly, the attacker modifies the sensitive data between the duplication and changes it back before the check. Secondly, during high-privilege syscalls, the kernel dereferences numerous untrusted pointers. The attacker may overwrite these pointers, enforcing these syscalls to perform a high-privilege write operation. Consequently, a low-privilege arbitrary write is converted into a high-privilege one. AKO [82] is similar PrivGuard, but it only protects credentials. Moreover, the duplicated data is not on the stack but on a reserved unprotected area.

PrivWatcher [14] protects credentials and the pgd. Compared to PrivGuard, PrivWatcher stores sensitive data in read-only domains and monitors its access. Furthermore, it assumes these domains are only writable by PrivWatcher. With this assumption, an attacker cannot tamper with sensitive data, preventing their exploitation. As Quan et al. [14] discussed, this assumption was not supported by hardware. Hence, the actual performance overhead of PrivWatcher may be higher than the evaluated overhead.

Randomization. SALADS [13] protects sensitive (*i.e.*, `cred` and `inode`) and non-sensitive (e.g., `list_head`) members of data objects by randomizing their layout at runtime. An attacker may manipulate the wrong data members since the data layout may change between the leak and attack phases. Hence, SALADS mitigates its exploitation. However, the protection level of SALADS strongly depends on how often the data objects are re-randomized. Moreover, the re-randomization rate determines the performance overhead.

PT-Rand [28] randomizes the location of all page tables by mapping them with an offset to a random base instead of the DPM. This random base is stored in a dedicated inaccessible register. Furthermore, PT-Rand ensures no leakage of the random location by substituting page table references with an offset to this random base. Therefore, the location of page tables cannot be leaked, preventing the manipulation of page tables. Davi et al. [28] achieve this strong security claim with a low runtime overhead.

Compartmentalization. Mondrix [80] provides memory protection by proposing significant hardware changes to add multiple protection features, a concept of ownership, and protection domains. A separate permission table stored in the main memory provides more fine-grained control over the memory access rights. Stacks are only writable within a thread's current stack frame. Witchel et al. [80] introduce a dedicated stack permission table for access outside the current stack frame. Access to functions that run in a higher privilege domain uses a new form of lightweight call gates that push the return address to a shadow call stack. Furthermore, Mondrix adds new caches for the added protection checks and call gates to increase performance. However, these significant hardware changes prevent the use of Mondrix today.

HAKC [54] performs compartmentalization of Loadable Kernel Modules (LKMs). It moves all data objects accessible by the LKM into partitions, where each data object belongs to exactly one partition. Each partition and hence data object can only be accessed by its owner, defined on compile time. Since HAKC does not account for simultaneous ownership, data objects with simultaneous readers cannot be compartmentalized. In the Linux kernel, simultaneous readers are very common, e.g., RCU-locked credentials and inodes [55]. Hence, we mark all data objects with simultaneous readers as insufficiently protected. Credentials and virtual memory are shared between threads within one process. Inodes, filesystem mounts, and user-accessible pages can also be accessed simultaneously via the DPM [8, 75], e.g., during a pathname lookup. Furthermore, HAKC protects the stack only on compartment granularity and does not account for concurrent threads in these compartments. Therefore, in case of an exploitable bug, a thread can overwrite the stack (including stored registers on preemption) from another thread if they are in the same compartment. Although the HAKC proof-of-concept implementation only has two compartments, we classify the runtime overhead as high.

Isolation. KDPM [45] protects sensitive kernel data by only permitting certain syscalls (`sys_execve` and `sys_set*id`) to grant write permissions. However, since write permissions are granted to the entire syscalls, an attacker can tamper unprotected pointers which are dereferenced within these syscalls to obtain a high-privilege arbitrary write primitive. Moreover, KDPM is susceptible to forgery attacks. Lastly, they evaluated KDPM on an MPK emulator instead of real hardware.

KPRM [46] protects sensitive kernel data during syscalls by unmapping them from the threads' address space. To manage sensitive data access, KPRM hooks the page-fault handler. KPRM maps a restricted page if the access is allowed within the executed code or kills the process if the access is invalid. However, KPRM does not account for multiple threads with a shared kernel address space. A thread executing kernel code can access sensitive data currently mapped for a different thread within the same thread group. Furthermore, the high reliance on frequent page faults and unmapping restricted pages leads to high performance overhead.

Song et al. [70] proposed an automated tool for identifying sensitive kernel data objects. Moreover, they proposed mitigation KENALI protects these objects and the stack with shadow memory. KENALI also prevents various mitigation-bypass attacks. Unfortunately, they lack a hardware primitive to protect sensitive data efficiently, leading to high performance overhead. KENALI does not mitigate against the discussed **arbitrary use-after-free** attack from Section 7.1. Therefore, an attacker can convert an arbitrary write primitive to an arbitrary use-after-free primitive and perform DirtyCred [50], resulting in a privilege escalation KENALI cannot protect against. Since the principle of DirtyCred can be applied to `cred`, `vm_area`, `vfsmount`, and all other objects allocated with a `kmem_cache`, these data objects are only partially protected.

xMP [62] employs Extended Page Table (EPT) switching to enforce domain protection similarly to DOPE. It protects page tables, credentials, the `pgd`, and sensitive data mapped in user space. Unfortunately, xMP does not mitigate against the discussed **arbitrary use-after-free** attack because it only ensures pointer integrity instead of ownership like DOPE. Therefore, the credentials are only partially protected. Besides credentials, xMP protects the `pgd` and page tables sufficiently. Even though xMP only protects three kernel data objects, their performance overhead is high.

We deploy DOPE to protect credentials, virtual memory, virtual memory areas, inodes, page tables, filesystem mount, stored registers, and user-accessible pages with strong security guarantees while maintaining a reasonable performance overhead.

9 CONCLUSION

In this paper, we presented our principled mitigation DOPE to protect against data-oriented attacks. DOPE enforces domain protection by restricting memory accesses during kernel execution based on the principle of least privilege. We implemented a DOPE proof-of-concept and conducted a case study that protects eight sensitive data objects from being used for privilege escalation exploits. For our proof-of-concept, we observed a reasonable performance overhead of 2.3 % for real-world user applications, significantly improving in terms of security with respect to performance over existing mitigations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. Furthermore, we thank Gaëtan Cassiers, Martin Unterguggenberger, and Andreas Kogler for valuable discussions and feedback on this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Additionally, this work was supported by Red Hat. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *CCS*.
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *TISSEC* (2009).
[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *S&P*.
[4] Khalid Aziz. 2019. Add support for eXclusive Page Frame Ownership. https://lwn.net/Articles/779818/
[5] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. 2018. Hardware assisted randomization of data. In *RAID*.
[6] Sandeep Bhatkar and R Sekar. 2008. Data space randomization. In *DIMVA*.
[7] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*.

[8] Neil Brown. 2015. RCU-walk: faster pathname lookup in Linux. https://lwn.net/Articles/649729/
[9] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*.
[10] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. 2008. Data randomization. (2008).
[11] Nicholas Carlini and David A. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium*.
[12] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *OSDI*.
[13] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. 2015. A Practical Approach for Adaptive Data Structure Layout Randomization. In *European Symposium on Research in Computer Security*.
[14] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. 2017. PrivWatcher: Non-Bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *AsiaCCS*.
[15] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *USENIX Security Symposium*.
[16] Chromium. 2018. PartitionAlloc Design. https://chromium.googlesource.com/chromium/src/+/lkcr/base/allocator/partition_allocator/PartitionAlloc.md
[17] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.
[18] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/
[19] Jonathan Corbet. 2016. Defending against Rowhammer in the kernel. https://lwn.net/Articles/704920/
[20] Jonathan Corbet. 2016. Exclusive page-frame ownership. https://lwn.net/Articles/700647/
[21] Jonathan Corbet. 2018. Kernel support for control-flow enforcement.
[22] Jonathan Corbet. 2020. Memory protection keys for the kernel. https://lwn.net/Articles/826554/
[23] Jonathan Corbet. 2022. Seeking an API for protection keys supervisor. https://lwn.net/Articles/894531/
[24] Jonathan Corbet. 2022. Solutions for direct-map fragmentation. https://lwn.net/Articles/894557/
[25] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/
[26] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *S&P*.
[27] Cyril Hrubis. 2022. Linux Test Project. https://github.com/linux-test-project/ltp
[28] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *NDSS*.
[29] Jake Edge. 2011. Extending the use of RO and NX. https://lwn.net/Articles/422487/
[30] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/
[31] Jake Edge. 2020. Control-flow integrity for the kernel. https://lwn.net/Articles/810077/
[32] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. *arXiv:2303.16353* (2023).
[33] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *Euro S&P*.
[34] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[35] Daniel Gruss, Michael Schwarz, and Moritz Lipp. 2018. Meltdown: Basics, Details, Consequences. In *BlackHat USA*.
[36] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *USENIX Security Symposium*.
[37] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *S&P*.
[38] Ralf Hund, Thorsten Holz, and Felix C. Freiling. 2009. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium*.
[39] Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers.
[40] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*.
[41] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *USENIX Security Symposium*.
[42] kernel.org. 2009. Virtual memory map with 4 level page tables (x86_64). https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
[43] Michael Kerrisk. 2021. *capabilities(7) — Linux manual page*. https://man7.org/linux/man-pages/man7/capabilities.7.html.
[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.
[45] Hiroki Kuzuno and Toshihiro Yamauchi. 2022. KDPM: Kernel Data Protection Mechanism Using a Memory Protection Key. *International Workshop on Security* (2022), 66–85.
[46] Hiroki Kuzuno and Toshihiro Yamauchi. 2022. Prevention of Kernel Memory Corruption Using Kernel Page Restriction Mechanism. *Journal of Information Processing* 30 (2022), 563–576.
[47] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2022. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *NDSS*.
[48] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Architectural Support for Programming Languages and Operating Systems*.
[49] Guanyu Li, Dong Du, and Yubin Xia. 2020. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity* 3 (2020), 11.
[50] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *ACM*.
[51] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*.
[52] Yong Liu, Jun Yao, and Xiaodong Wang. 2022. USMA: Share Kernel Code with Me. *BlackHat Asia* (2022).
[53] LLVM. 2019. The LLVM Compiler Infrastructure. https://llvm.org
[54] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKC. In *NDSS*.
[55] Paul McKenney. 2007. What is RCU, Fundamentally? https://lwn.net/Articles/262464/
[56] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*.
[57] Joao Moreira. 2022. Kernel FineIBT Support. https://lwn.net/Articles/891976/
[58] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios Kemerlis. 2017. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel. In *Black Hat Asia*.
[59] James Morse. 2015. arm64: kernel: Add support for Privileged Access Never. https://lwn.net/Articles/651614/
[60] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX ATC*.
[61] Phoronix. 2022. OpenBenchmarking. https://openbenchmarking.org
[62] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *S&P*.
[63] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. 2018. PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks. *IEEE Access* 6 (2018), 46584–46594.
[64] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. 2021. uSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *RAID*.
[65] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.
[66] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Architectural Support for Programming Languages and Operating Systems*.
[67] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.
[68] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium*.
[69] Mark Seaborn. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html Retrieved on June 26, 2015.
[70] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *NDSS*.
[71] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *ACM*.
[72] Yoo Sungbae, Park Jinbum, Kim Seolheui, Kim Yeji, and Kim Taesoo. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *USENIX Security Symposium*.
[73] The Linux Kernel. 2021. File system drivers (Part 2). https://linux-kernel-labs.github.io/refs/heads/master/labs/filesystems_part2.html

[74] The Linux Kernel. 2022. Index Nodes. https://www.kernel.org/doc/html/latest/filesystems/ext4/inodes.html
[75] The Linux Kernel. 2022. Locking. https://www.kernel.org/doc/html/latest/filesystems/locking.html
[76] The Linux Kernel. 2022. Memory Allocation Guide. https://docs.kernel.org/core-api/memory-allocation.html?highlight=kmem_cache_alloc
[77] The Linux Kernel. 2022. Memory Protection Keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html
[78] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. In *USENIX Security Symposium*.
[79] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *EuroSys*.
[80] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *ACM SIGOPS Operating Systems Review*.
[81] Jidong Xiao, Hai Huang, and Haining Wang. 2015. Kernel data attack is a realistic security threat. In *International Conference on Security and Privacy in Communication Systems*.
[82] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. 2021. Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes. *International Journal of Information Security* 20 (2021).

APPENDIX

10 IMPLEMENTATION DETAILS OF TRUSTED CODE

In this section, we provide measures to address any implementation issues while ensuring that our code adheres to the trusted code constraints we defined in Section 4.5.

Memory management. The buddy allocator in the Linux kernel allocates contiguous physical memory in page order chunks, *i.e.*, $2^n \cdot PAGE_SIZE$. On top of the buddy allocator sits the slab allocator, and stores caches of available objects with a desired predefined size [76]. The kernel supports three slab allocators: SLAB, SLOB, and SLUB, all of which store metadata on the allocated page. Allocations via `kmem_cache` and our modified `kmem_dope_cache` deploy one of these slab allocators, *i.e.*, SLUB. If a domain protects a page allocated by the buddy allocator, then allocating or freeing an object via `kmem_dope_cache` would require write permission to the domain, as metadata may be written to the protected page. However, since during allocation and freeing, untrusted pointers are accessed, granting write permission would violate trusted code constraints.

To address this issue, we propose to extend the slab allocator by adopting a PartitionAlloc-based design similar to Chrome [16], which separates data and metadata into two distinct locations. This approach would eliminate the need for write permission to the domain when allocating or freeing an object via our `kmem_dope_cache`. While implementing this extension requires significant effort, we acknowledge that it is outside the scope of this work.

Outsourcing the metadata of the slab allocator to an unprotected object does not pose a security risk because DOPE does not rely on the allocator's trustworthiness. In Section 7.1, we illustrate various attacks on the allocator and show how DOPE mitigates them.

Read-Copy Update. The Linux kernel supports the efficient synchronization mechanism Read-Copy Update (RCU) that enables concurrent updates by readers [55]. Besides blocking the current thread for synchronization, RCU also supports non-blocking updates by invoking a callback function either during a software interrupt (*i.e.*, `softirq`) or by a dedicated RCU thread [62]. During

```
1  /**
2   * struct callback_head - callback structure for use with
3   * RCU and task_work
4   * @next: next update requests in a list
5   * @func: actual update function to call after the grace
6   *        period.
7   */
8  struct callback_head {
9    struct callback_head *next;
10   void (*func)(struct callback_head *head);
11 } __attribute__((aligned(sizeof(void * ))));
12
13 /* Types */
14 #define rcu_head callback_head
15 typedef void (*rcu_callback_t)(struct rcu_head *head);
16
17 /**
18  * call_rcu() - Queue an RCU callback for invocation
19  * after a grace period.
20  */
21 void call_rcu(struct rcu_head *head, rcu_callback_t func){
22   ...
23   head->func = func;
24   head->next = NULL;
25   ...
26 }
```

Listing 3: Callback function provided by Linux's RCU locking mechanism.

the callback, data (stored as generic data type, *i.e.*, `rcu_head`) is accessed that may or may not be protected by a domain. This is illustrated in Lines 23 and 24 from Listing 3, where the function `call_rcu` access the `head` pointer. If `head` is stored in a sensitive and protected data objects, e.g., `struct cred`, than this function would require write permission to the corresponding domain, e.g., credential domain. Otherwise, the hardware raises an fault, and DOPE would interpret this as an exploitation attempt. However, granting temporary access permission would violate trusted code constraints. To address this issue, we separated the `rcu_head` member from sensitive data objects and instead stored a pointer to a dynamically allocated `rcu_head`.

11 DETAILED SENSITIVE DATA OBJECTS

Credentials. The credential struct contains information on its thread's privilege level and is stored as a pointer in the `task_struct`. It is a popular attack target [13, 14, 62, 63, 70], cf. CVE-2021-26708, CVE-2017-16995, or CVE-2017-2636. Overwriting credentials immediately leads to privilege escalation as the kernel interprets the thread as high-privilege. In addition to traditional UNIX per-process credentials (*i.e.*, `*id`), Linux supports per-thread capabilities [43]. These capabilities partition the privileges associated with the superuser into a finer granularity. Hence, we also protect capabilities.

Inodes. The inode in a UNIX filesystem, e.g., `ext4`, stores all metadata associated with its file [74]. Among this metadata is non-sensitive data, e.g., last modified or last access time, and sensitive data, e.g., access rights and owner and group identifier. By modifying the sensitive data of an inode, an attacker can change the permission or owner of the associated file. Moreover, we define the flag variable as sensitive because it contains information on

whether the file is private or immutable. Another way to use inodes for privilege escalation is to forge their identifiers, uniquely identifying the inodes within the mounted file system [73]. In summary, we protect `i_ino`, `i_mode`, `i_uid`, `i_gid`, and `i_flags`.

Page tables. Page tables are valuable targets for attackers because they contain lower-level page permissions [28, 62, 70]. By overwriting those permission bits, an attacker controls the access permissions of the lower page levels, e.g., an attacker can modify permission bits in the page-table entry to change a kernel code page to writable. As a result, the attacker has a writable and executable kernel memory area.

Virtual memory areas. In Linux, there is another possibility to manipulate the permission bits of page tables by tampering with the `vm_page_prot` stored in `vm_area_struct`. Liu et al. [52] demonstrated a data-oriented attack called User Space Mapping Attack, in which an attacker overwrites `vm_page_prot` to modify the permissions of page-table entries. This attack results in a kernel page being mapped into the user space, which is then overwritten.

Virtual memory. In the Linux kernel, the thread's `pgd` is the top level of a page table [42] and is stored in the `mm_struct`. By overwriting the stored `pgd`, an attacker has control over the hardware `pgd` and may forge a virtual to physical page mapping [14, 62, 63, 70]. The attacker then may add virtual addresses that map to arbitrary physical addresses with arbitrary permissions.

Filesystem mount. Song et al. [70] showed a data-oriented attack in which an attacker tampers with mount flags. The attacker manipulates the flag to mark a mounted file system as no longer read-only. Hence, the attacker can write to files within the read-only file system. The target is the system partition, which is read-only mounted on most Android devices.

12 DPM-FPATCH ATTACK

Operating systems have the important task of managing privilege levels and ensuring the isolation of processes. It is crucial to prevent low-privilege processes from tampering with the data of other processes. However, since the entire physical memory is mapped via the Direct Physical Mapping (DPM), an attacker can use an arbitrary write primitive in the kernel to manipulate data on the DPM, in particluar with including user-accessible data. This introduces a new variant of data-oriented attacks called DPM-FPATCH, which current state-of-the-art mitigations are unable to protect against.

12.1 Attack

The DPM-FPATCH data-oriented attack variant overwrites data of any user-accessible file. With DPM-FPATCH, we demonstrate an attack on the `/etc/passwd` file, as illustrated in Figure 6.

In step ①, an attacker opens and reads the entire content of the high-privilege but user-accessible file. Hence, the kernel loads the content from the disk into the DPM. In step ②, the attacker scans the entire DPM with the arbitrary read and obtains the address where the file content is stored. In step ③, the attacker can use the arbitrary write to overwrite the content via the DPM. One possible modification of a high-privilege file is to change the first line of `/etc/passwd` from `root:x:0:0:root:/root:/usr/bin/zsh` to `root::00:0:root:/root:/usr/bin/zsh`. This change indicates that a root login does not require authentication ④.

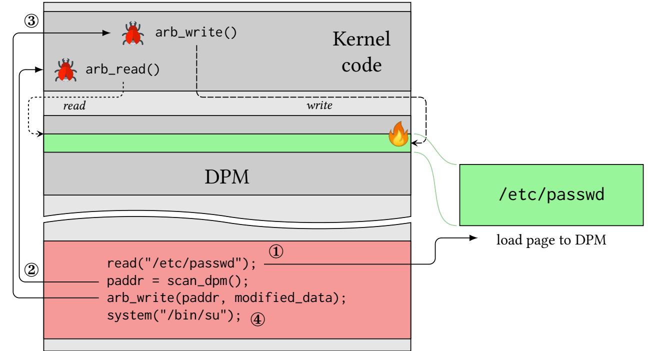

Figure 6: DPM-FPATCH attack on `/etc/passwd` file. Step ① loads the read-only file to the DPM, while step ② performs an arbitrary read call for each page read of the DPM to obtain the address where the file content is located. Next, step ③ carefully overwrites sensitive content of the file via the DPM to grant the attacker root privileges without authentication ④.

In summary, an attacker can perform DPM-FPATCH to modify the data of any user-accessible file. For our demonstration, the attacker modifies `/etc/passwd` to illegally indicate to the system that root does not require authentication on login, elevating the attacker's privileges.

12.2 Potential mitigation

Kemerlis et al. [40] showed the devastating outcome of user data accessibility via the DPM for control-flow hijacking attacks. To prevent this accessibility, their proposed mitigation, XPFO, prevents all accesses to user data via the DPM by either mapping a page in user space or the DPM, but never both. Since DPM-FPATCH does not rely on this mapping, XPFO does not prevent this attack variant. Moreover, according to Linux kernel developers [4, 20], who have extensively tested XPFO, the runtime overhead is above 30 %. Therefore, the XPFO patch was never merged into the Linux kernel.

13 DETAILED EVALUATION RESULTS

Binary size and compile time overhead. To enforce domain protection, our LLVM pass inserts functions for domain switching and validation. These inserted functions increase the binary size and the compile time. To illustrate both overheads, we compile an unmodified Linux kernel version 5.19 with clang version 15.0.1 as a baseline. We then compile our modified Linux kernel with our LLVM pass enabled for our proof-of-concept implementation. The results are that the binary size and compile time increase by 0.27 % and 1.5 ± 0.3%, respectively.

Micro-benchmarks. We use LMbench to evaluate the latency and bandwidth overhead for various benchmarks of our proof-of-concept implementation. We consider the baseline kernel version 5.19, DOPE-light, and DOPE, where DOPE-light protects the same data objects as our case study DOPE, except for user-accessible pages. We include DOPE-light to highlight the overhead caused

Table 4: Micro-benchmark results.

	Benchmarks	Baseline	Overhead in %	
			DOPE-light	DOPE
Latency in µs	syscall null	0.08	−0.1 ± 0.4	0.0 ± 0.4
	syscall open+close	1.03	107.7 ± 2.9	127.8 ± 2.9
	syscall read	0.15	0.6 ± 2.3	98.0 ± 2.6
	syscall write	0.11	0.0 ± 0.7	128.0 ± 0.7
	syscall fstat	0.16	16.0 ± 1.6	15.0 ± 0.4
	syscall pipe	3.65	1.1 ± 0.4	21.0 ± 0.1
	proc procedure	0.002	−1.6 ± 4.4	0.5 ± 1.4
	proc fork	62.5	27.0 ± 1.6	34.0 ± 1.8
	proc fork+exec	211	29.9 ± 1.0	37.0 ± 0.8
	proc shell	458	24.0 ± 0.5	29.0 ± 0.5
	sem	0.46	−7.7 ± 6.3	−5.0 ± 4.3
	pagefault	0.15	12.0 ± 0.5	11.0 ± 0.5
	dram page	1.63	1.0 ± 3.3	2.1 ± 2.5
	signal fault	0.42	37.0 ± 1.4	38.0 ± 0.7
	signal install	0.14	0.4 ± 1.0	0.5 ± 1.1
	signal catch	0.88	0.4 ± 0.4	0.6 ± 0.6
Bandwidth in MB/s	mem rd 4 k	160	0.0 ± 0.2	0.0 ± 0.2
	mem wr 4 k	110	−0.1 ± 0.1	0.0 ± 0.1
	mmap rd 4 k	51.4	0.3 ± 0.5	0.2 ± 0.4
	mmap rd 1 M	48.0	−0.7 ± 1.6	−3.0 ± 1.2
	file_rd o2c 4 k	2.71	43.0 ± 3.7	72.0 ± 1.1
	file_rd o2c 1 M	17.6	1.5 ± 1.1	4.7 ± 0.7
	file_rd io 4 k	6.70	6.5 ± 0.4	87.0 ± 0.5
	file_rd io 1 M	18.1	0.3 ± 0.9	4.1 ± 0.7

by protecting user-accessible pages. To achieve stable results, we run each benchmark 80 times and compute the mean and standard deviation.

Table 4 illustrates the evaluation results. The null syscall benchmark indicates that DOPE does not add any syscall entry or exit latency. All syscalls interacting with user-accessible pages have an increased runtime overhead because DOPE switches domains on every user-accessible data access. The micro-benchmarks open, read, and write, show the increased overhead with between 98 % to 128 %. Except for open, DOPE-light reduces the syscall's overhead to about 0 % compared to DOPE. The open syscall has a performance overhead of 110 ± 3% because of the inserted validation checks and domain switches for the credential domain. Similar to open, fstat also performs validation checks as illustrated with about 15 % runtime increase for DOPE and DOPE-light. Since the pipe syscall interacts with user-accessible data, it has an elevated overhead of 21 ± 0.1% for DOPE. The overhead for all three process operations, fork, fork+exec, and shell, are 29 % to 37 % for DOPE and 24 % to 30 % for DOPE-light. The synchronization syscall, sem, has negligible runtime overhead. Since, during the pagefault benchmark, a thread accesses page tables, DOPE and DOPE-light increase the performance overhead by about 11 % due to page table domain switches. We show that the overhead caused by a pagefault is negligible when considering the access time to a page from DRAM, as the overhead of the dram page benchmark is between 1 % to 2.1 %. A signal fault has an overhead of about 37 % for both DOPE and DOPE-light. The overhead of signal install and catch is negligible. We compute the total overhead by averaging over all overheads, resulting in an overhead of 32 % for DOPE and 17 % for DOPE-light.

With all four memory and mmap bandwidth benchmarks, Table 4 shows that DOPE does not add any runtime overhead on normal memory accesses, independent of how the memory is allocated. We run the file read benchmark with two parameters: open2close and io_only, and observe an overhead of 72 % to 87 % for a filesize of 4 kB. The overhead decreases to about 4 % by running the benchmark with a size of 1 MB. For DOPE-light, file read with the open2close parameter has an overhead of 43 % caused by the open syscall.

Phoronix Test Suite macro-benchmarks. Our benchmarks from Phoronix Test Suite split up into stress tests and real-world applications, as shown in Figure 5. Among the stress tests are one inter-process communication, one kernel scheduler, two filesystem, and one threaded I/O benchmarks. For the inter-process communication benchmark (IPC-benchmark), we observe that DOPE elevates the overhead by about 2 % independent of the used pipe (unnamed or named FIFO) and message size (1024 Byte or 4096 Byte). We run Schbench, which evaluates our scheduler, with two worker threads, each creating two, four, or six messenger threads. DOPE has an overhead between 2 % to 3.7 % for the scheduler benchmark, decreasing with more messenger threads. With Dbench, we perform two benchmark tests resulting in an overhead of 3 ± 0.1% for one and 1 ± 0.1% for six client threads. DOPE has an elevated performance overhead for LevelDB fill of 9.3 ± 1.5% due to extensive write syscall usage and, thus, extensive user-accessible domain switches. Since the LevelDB read benchmark caches read data in software, there are fewer syscalls and domain switches. The threaded I/O stress test (TIObench) has an overhead of 4.3 ± 3.1% for the read and 5.5 ± 1.8% the write benchmark.

Among the real-world applications are two web-server, two database, and four user application benchmarks (cf. Figure 5). The two web-server benchmarks, Apache and NGINX, has an runtime overhead of about 2 %, with the number of Apache requests having little effect on the overhead. DOPE affects the SQLite benchmark with an overhead of 3.4 ± 1.2%, while the in-memory Redis benchmark is affected with a low overhead of 0.4 ± 0.2%. Lastly, all four user applications, PHPBench, compress-lzma, and OpenSSL sha and rsa, results in a low overhead between 0.5 % to 1.3 %.

We observe an elevated standard deviation for various benchmarks, especially for those with multiple threads and high kernel execution time. However, the baseline and DOPE-enhanced kernel binary show similar levels of high standard deviation. Therefore, the noise properties of the kernel, such as hardware interrupts or context switches, contribute to the high standard deviation.

The average performance overhead of the Phoronix Test Suite macro-benchmarks is 2.3 %.

SPEC CPU 2017. We perform multiple speed macro-benchmarks of SPEC CPU 2017, as illustrated in Figure 5. All measured overheads of the speed macro benchmarks are below 1.8 %, which is in line with the user application macro-benchmarks from Phoronix Test Suite. We compute the overall overhead by averaging all results, leading in an overhead of 0.4 %.