DIRTYFREE: Simplified Data-Oriented Programming in the Linux Kernel

Yoochan Lee*, Hyuk Kwon†, and Thorsten Holz*
*Max Planck Institute for Security and Privacy (MPI-SP)
†Theori, Inc.
{yoochan.lee, thorsten.holz}@mpi-sp.org, pwn3r@theori.io

***Abstract*—With the advent of Kernel Control-Flow Integrity (KCFI), Data-Oriented Programming (DOP) has emerged as an essential alternative to traditional control-flow hijacking techniques such as Return-Oriented Programming (ROP). Unlike control-flow attacks, DOP manipulates kernel data-flow to achieve privilege escalation without violating control-flow integrity. However, traditional DOP attacks remain complex and exhibit limited practicality due to their multistage nature, typically requiring heap address leakage, arbitrary address read, and arbitrary address write capabilities. Each stage imposes strict constraints on the selection and usage of kernel objects.**

To address these limitations, we introduce DIRTYFREE, a systematic exploitation method that leverages the arbitrary free primitive. This primitive enables the forced deallocation of attacker-controlled kernel objects, significantly reducing exploitability requirements and simplifying the overall exploitation process. DIRTYFREE provides a systematic method for identifying suitable arbitrary free objects across diverse kernel caches and presents a structured exploitation strategy targeting security-critical objects such as `cred`. Through extensive evaluation, we successfully identified 14 arbitrary free objects covering most kernel caches, demonstrating DIRTYFREE's practical effectiveness by successfully exploiting 24 real-world kernel vulnerabilities. Additionally, we propose and implement two mitigation techniques designed to mitigate DIRTYFREE, effectively preventing exploitation while incurring negligible performance overhead (i.e., 0.28% and -0.55%, respectively).

I. INTRODUCTION

With the introduction of *kernel control-flow integrity* (KCFI) [1, 2], the era of *return-oriented programming* (ROP) [3–5] attacks in the kernel has seen a significant decline. This shift has led to a growing interest in *data-oriented programming* (DOP) [6–10]. Unlike ROP, which manipulates control-flow to achieve privilege escalation, DOP accomplishes similar goals through data-flow manipulation. In user-space attacks, the ultimate goal typically involves executing attacker-controlled code, making control-flow manipulation essential. In contrast, kernel exploitation primarily aims at privilege escalation, which can be achieved entirely by corrupting privilege-related data structures, making DOP a viable and well-suited strategy for kernel exploitation.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240527
www.ndss-symposium.org

Despite its potential, traditional DOP approaches are overly complex and impose significant restrictions, limiting their practical applicability. Traditional DOP exploits typically consist of three distinct stages: (i) leaking a heap address, (ii) achieving arbitrary address reads [11], and (iii) enabling arbitrary address writes [12, 13]. Each stage requires identifying and leveraging specialized kernel objects that satisfy specific constraints. Moreover, the conditions necessary for achieving each primitive differ substantially, making it challenging to satisfy all requirements simultaneously. Consequently, exploiting a vulnerability through traditional DOP attacks requires exceptionally strong exploitability to successfully fulfill the diverse and stringent conditions across all three stages.

To address this complexity, researchers from both academia and industry have explored simpler exploitation techniques [14–17]. A notable example is the *temporal cross-cache attack* [17], which exploits the *reclaim* mechanism of the Linux SLUB allocator to control page reuse. Subsequent work [18–20] has refined these techniques to better control which pages are reused. However, since these techniques rely on predictable page reuse, defenders have proposed a strong mitigation known as *SLAB Virtual* [21], which decouples virtual addresses from physical addresses. This mitigation, now adopted in security-focused environments like Google's kernel CTF [22] competitions, makes cross-cache attacks no longer feasible in practice.

In this paper, we introduce a novel and general exploitation technique that directly targets and corrupts security-critical kernel objects to escalate privileges. More specifically, our approach relies on memory corruption to replace low-privilege kernel objects with high-privilege objects, enabling attackers to escalate privileges effectively. By combining heap spraying with partial pointer overwrites, we reduce the number of stages in the DOP attack from multiple to a single, streamlined step in favorable conditions. This simplification significantly reduces the overall complexity and enables the exploitation of vulnerabilities that were previously considered too weak or impractical for conventional DOP approaches. As a result, such vulnerabilities can now be reliably exploited using our proposed approach.

To support this technique, we introduce a novel exploitation primitive called *arbitrary free*. This primitive enables attackers to forcibly free kernel objects at attacker-controlled memory addresses, reliably transitioning targeted kernel objects into a

use-after-free state. Using cross-cache freeing techniques, we can even free security-critical objects that reside in dedicated caches. A particularly notable advantage of the arbitrary free primitive is its minimal exploitation requirement: it only demands minimal pointer corruption—often as small as a one-byte overwrite—to successfully trigger a use-after-free scenario. This reduction in exploitability requirements significantly broadens the practicality and applicability of DOP attacks, allowing for successful exploitation in a wide range of kernel environments and configurations.

So far, the general idea of an arbitrary free primitive has not yet been fully explored or generalized across a wide range of vulnerabilities. First, the specific conditions and characteristics required for objects to qualify as arbitrary free objects have not been clearly defined. Although arbitrary free objects have been identified and utilized within certain caches, a comprehensive set of arbitrary free objects suitable for every cache remains unexplored. Second, the methodology for selecting target kernel objects that can reliably lead to privilege escalation using the arbitrary free primitive is largely unexplored. Consequently, existing studies typically leverage arbitrary free primitives merely as an intermediate step toward control-flow hijacking methods such as ROP, rather than directly facilitating privilege escalation purely through DOP.

To address these gaps, we present DIRTYFREE, a novel, systematic exploitation technique that simplifies DOP attacks by fully leveraging the arbitrary free primitive. First, we propose a systematic approach to identify arbitrary free objects. Specifically, we define arbitrary free objects as kernel objects containing pointers that are used as arguments for the kernel free function (i.e., `kfree()`). Using this definition, we can systematically identify suitable arbitrary free objects across various kernel caches, which enables us to successfully discover applicable objects in most general caches. In addition to identifying these objects, DIRTYFREE provides a structured exploitation methodology that could directly achieve privilege escalation using the arbitrary free primitive, *without* resorting to traditional control-flow hijacking. We specifically target the kernel object `cred`, which governs process privileges in the Linux kernel. Our technique outlines the steps necessary to exploit a vulnerability using arbitrary free: performing effective object spraying, ensuring the allocation of root credentials, and executing a reliable post-exploitation procedure that ultimately results in privilege escalation.

To demonstrate the effectiveness of DIRTYFREE, we conducted a thorough evaluation on Linux kernel v6.8, which is widely adopted as the default kernel in Ubuntu 24.04 LTS. We successfully identified 14 arbitrary free objects applicable to all caches except `kmalloc-8`. We then tested our approach against 31 real-world Linux kernel vulnerabilities, successfully exploiting 24 of them, thereby highlighting the practical applicability and reliability of DIRTYFREE. The seven failed vulnerabilities exhibit low exploitability, making it infeasible to leverage them to reach the arbitrary free primitive. Lastly, we compared DIRTYFREE with other DOP techniques, confirming that our approach significantly reduces complexity while improving exploitability across a wider range of scenarios.

Beyond introducing DIRTYFREE, we also propose two practical mitigation techniques to defend against arbitrary free exploitation. The first isolates arbitrary free objects in independent caches, ensuring they reside separately from other general cache objects. Consequently, the majority of kernel vulnerabilities can no longer manipulate pointers inside arbitrary free objects, effectively mitigating their exploitation. As a second mitigation, we introduce verification logic directly at the sites where `kfree()` operations occur, specifically checking whether a cross-cache free is being attempted. By enforcing this check, we ensure that `kfree()` operations exclusively free objects within their corresponding kmalloc caches, thereby preventing unintended cross-cache frees. These two mitigations effectively prevent exploitation via the arbitrary free primitive, while introducing minimal overhead (i.e., 0.28% and -0.55%, respectively).

To summarize, we make the following three contributions:

- We introduce DIRTYFREE, a novel and general exploitation technique that simplifies complex DOP attacks, enabling reliable privilege escalation on Linux systems.
- We demonstrated the practicality of DIRTYFREE by evaluating it against 31 real-world Linux kernel vulnerabilities, successfully exploiting 24 of them across a variety of configurations.
- We propose and implement two effective defense mechanisms against arbitrary free-based exploitation, achieving strong protection with minimal performance overhead (i.e., 0.28% and -0.55%).

We release the data and source code of DIRTYFREE at https://github.com/MPI-SysSec/DirtyFree to foster open science.

II. BACKGROUND

A. Kernel Heap Memory Management

The Linux kernel manages heap memory by subdividing pages into smaller slots to enhance performance and minimize memory fragmentation. Although the Linux kernel provides three distinct heap allocators [23–25], they all follow the same fundamental design. Specifically, each allocator uses a caching mechanism to efficiently manage objects, categorizing them either by object size (i.e., general caches) or by specific object type (i.e., dedicated caches). Because each cache exclusively uses its own set of pages, cross-cache interference and unintended memory overlap are inherently mitigated.

General Cache. The Linux kernel uses general-purpose caches (i.e., `kmalloc-*`) to efficiently manage objects of varying sizes. When an object is allocated, the requested size is rounded up to the nearest matching general cache. Because general caches group diverse objects based solely on size rather than type, unrelated objects may share the same memory region. As a result, memory corruption affecting one object can potentially interfere with others, making general caches a common source of kernel heap vulnerabilities.

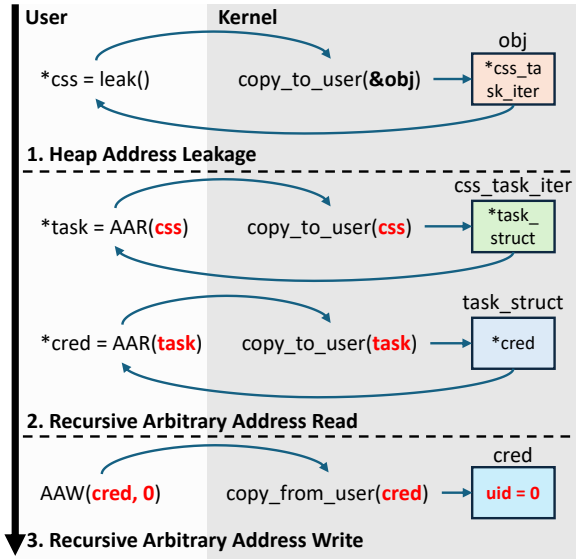

Fig. 1: Exploit flow of traditional DOP using three exploit primitives.

Dedicated Cache. The Linux kernel uses dedicated caches (e.g., `cred_jar` and `task_struct`) to manage security-critical objects, thereby enhancing overall system security through improved isolation. For instance, objects associated with privileges or permissions are particularly sensitive: if overwritten due to memory corruption, they can directly result in privilege escalation attacks. By placing such objects in separate, dedicated caches rather than mixing them with general-purpose allocations, the kernel reduces the risk of cross-object memory corruption. Importantly, each dedicated cache is responsible for managing a single, specific object type, further reducing the risk of memory corruption.

B. Data-Oriented Programming

Data-Oriented Programming (DOP) [6, 7, 15–17] is a powerful exploitation technique that allows attackers to achieve objectives such as privilege escalation *without* directly manipulating the program's control flow. Traditionally, control-flow-based exploits, such as return-oriented programming (ROP) [3, 4], have been the dominant attack methods. However, the emergence of kernel control-flow integrity (KCFI) [1] has significantly reduced the feasibility and popularity of ROP. Unlike ROP, DOP techniques remain effective since they rely exclusively on data manipulation, rendering them immune to KCFI defenses. Although researchers have studied numerous data-flow integrity mitigations [26–33] to defend against DOP attacks, none of these techniques have been integrated into the Linux mainline kernel. As a result, DOP continues to gain traction as a particularly and increasingly prevalent exploitation strategy.

Traditional DOP attacks generally follow three distinct stages: (i) heap pointer leakage, (ii) recursive arbitrary address read (AAR) [11], and (iii) arbitrary address write (AAW) [12, 13], as illustrated in Figure 1. Initially, an attacker exploits a memory corruption vulnerability to leak a pointer to a kernel heap object. Subsequently, the attacker leverages memory corruption again to recursively perform AAR operations using the leaked pointer to systematically traverse kernel objects, identifying privilege-related structures (e.g., `struct cred`). Finally, the attacker uses memory corruption to employ AAW to overwrite critical fields (e.g., setting the UID field in `struct cred` to zero), thereby achieving privilege escalation.

Strength. DOP exploits provide a stable and reusable exploit across versions and architectures. Control-flow hijacking exploits, such as ROP, must frequently identify new gadgets specific to each kernel version or architecture. In contrast, DOP attacks remain largely consistent and stable across versions and architectures, provided that the sizes of the objects used in exploitation do not change. This consistency simplifies adaptation and enhances the practical effectiveness of DOP techniques.

Limitation. As explained above, traditional DOP attacks rely on three distinct primitives, each of which requires a corresponding special kernel object. Specifically, objects suitable for heap leaks, AAR, and AAW each have different constraints that must be satisfied. Successfully satisfying all of them within a single exploit typically requires a highly exploitable vulnerability with precise control over memory layout and object interactions. As a result, such ideal conditions are rare in practice, often rendering traditional DOP techniques impractical for real-world kernel vulnerabilities.

C. Threat Model

In this paper, we use the following threat model: first, we assume the attacker has local access to the Linux kernel and aims to escalate privileges by exploiting a heap memory corruption vulnerability in the kernel. Additionally, we assume that widely-deployed exploit mitigations and kernel protection mechanisms provided by the upstream Linux kernel are enabled. These include KASLR [34], SMEP [35], SMAP [36], KCFI [1], SLAB virtual [21], and KPTI [37]. These mitigations ensure that the kernel address space is randomized, and that access to user-space memory during kernel execution is restricted. In addition, control-flow hijacking becomes infeasible, and temporal cross-cache attacks are not available. Finally, we assume no hardware side channels are available to aid kernel exploitation.

III. Details about Arbitrary Free Primitive

In this section, we provide a detailed explanation of the *arbitrary free* primitive. We highlight the key advantages of this primitive and outline the conditions necessary to achieve arbitrary frees. Furthermore, we describe the underlying mechanisms that enable cross-cache frees and discuss the significant practical challenges associated with this technique.

A. Strength of Arbitrary Free

An arbitrary free primitive allows an attacker to forcibly free an arbitrary kernel object of their choosing, as illustrated in Figure 2. Specifically, by exploiting a memory corruption vulnerability, an attacker can corrupt a particular kernel pointer

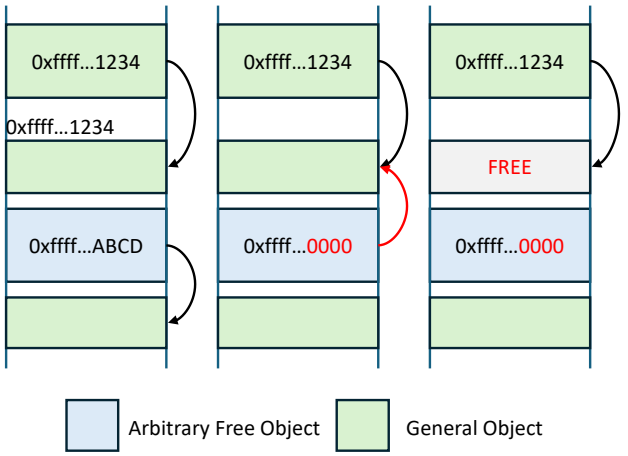

Fig. 2: Example of an arbitrary free primitive.

and subsequently invoke a `free` operation using that corrupted pointer. Since this procedure deviates from the standard process of freeing objects, the targeted object is freed while references to it may still exist. This can lead to a dangling pointer, which effectively places the object in a use-after-free state. In essence, the arbitrary free primitive enables an attacker to change every object into a use-after-free state.

Importantly, the arbitrary free primitive requires relatively weak conditions for successful exploitation. Unlike primitives such as arbitrary address read (AAR), which typically require overwriting at least 8 bytes to fully corrupt a kernel pointer, arbitrary free can be achieved through partial pointer corruption. Specifically, attackers only need to slightly modify the pointer to redirect it toward another kernel object. Consequently, in certain scenarios, even a one-byte overflow can be sufficient to achieve an arbitrary free primitive. This significantly lowers the bar for exploitation and makes the primitive widely applicable to a diverse range of kernel heap vulnerabilities.

B. Arbitrary Free Object

To be exploitable, an arbitrary free object must satisfy two key conditions. First, it must contain at least one pointer that references a kernel heap area, along with a method that can invoke `kfree()` using this pointer. Second, the arbitrary free object must be allocatable by an unprivileged user process, allowing an attacker to strategically place it adjacent to, or alias it with, a vulnerable kernel object. Additionally, the kernel function executing the `kfree()` operation must also be triggerable by an unprivileged user process. If either of these conditions is not met, exploitation becomes generally not feasible.

Moreover, for a kernel object to effectively serve as an arbitrary free object, there must be a sufficient time window between its allocation and the subsequent freeing operation. This window is crucial, as the attacker needs enough time to perform pointer corruption between these two events. Thus, if object allocation and the corresponding free occur within a single, immediate control-flow path, the available time window is too narrow for the attacker to intervene, rendering practical exploitation impossible.

C. Cross-Cache Free

To effectively leverage the arbitrary free primitive, an attacker must specifically target security-critical kernel objects. As previously discussed in §II-A, most security-critical objects are allocated within dedicated caches, whereas the majority of heap vulnerabilities arise from general caches. Thus, attackers inevitably face a scenario where they must free security-critical objects residing in dedicated caches from vulnerabilities located in general caches, a situation referred to as *cross-cache free*.

However, freeing objects allocated in dedicated caches typically requires precise knowledge of their corresponding cache. Specifically, dedicated caches mandate passing a pointer to the appropriate `struct cache` when invoking `kmem_cache_alloc()` and `kmem_cache_free()`. If an incorrect cache pointer is provided, cache metadata corruption occurs, often leading to a kernel panic. This limitation implies that objects allocated in a dedicated cache typically cannot be freed using objects from other caches.

To circumvent this constraint and perform a cross-cache free, attackers can use the `kfree()` function. Unlike `kmem_cache_free()`, `kfree()` does not require explicit information about the target cache. This design simplifies the management of dynamically-sized objects, such as *elastic objects* [11]. Internally, `kfree()` determines the appropriate cache solely from the given pointer and subsequently frees the object. Moreover, because `kfree()` can handle pointers from both general and dedicated caches, it enables attackers to conduct cross-cache frees. This capability allows attackers to trigger unintended use-after-free conditions on security-critical objects, even when the initial vulnerability lies in a separate cache.

D. Technical Challenges

Despite its powerful capability to transition arbitrary kernel objects into a use-after-free state, the arbitrary free primitive remains underexplored in kernel exploitation. Existing research and publicly available exploits have identified only a limited number of arbitrary free objects, leaving the primitive's broader applicability unverified. Consequently, its effectiveness for achieving privilege escalation has not yet been convincingly demonstrated and we tackle this open problem in this work. More specifically, we study the following two challenges:

C1: Systematically identifying arbitrary free objects. To facilitate more general adoption, a systematic approach to identifying suitable arbitrary free objects is necessary. Currently, such objects are selected in an ad-hoc manner, limiting their scope and hindering generalization. A robust identification methodology must consider various kernel memory allocation strategies, object lifetimes, and reuse patterns. Only by constructing a diverse and well-characterized collection of arbitrary free objects can the primitive generalize beyond isolated proof-of-concept scenarios and become an effective tool for exploiting a wider range of kernel vulnerabilities.

C2: Structured exploitation methodology. Another key challenge is the lack of a clear exploitation methodology to achieve privilege escalation through the arbitrary free primitive. Due to its relatively limited exposure, there are currently no well-established techniques for effectively leveraging this primitive. Even prior exploits [38, 39] that use the arbitrary free primitive typically involve complex ROP chains, adding significant complexity to the exploitation process. To address this issue, it is essential to first identify specific security-critical kernel objects whose freeing directly results in privilege escalation. Additionally, practical and reliable methodologies must be developed to reliably achieve privilege escalation through these objects.

IV. DirtyFree

In the following, we first provide a high-level overview of DirtyFree through a real-world example. Then, we explain how DirtyFree addresses the challenges associated with the arbitrary free primitive.

A. Overview

We illustrate the high-level operation of DirtyFree using a real-world Linux kernel vulnerability, CVE-2021-22555 [39], as an example. This vulnerability arises due to missing boundary checks, enabling an attacker to perform an out-of-bounds write by overwriting two additional bytes with null bytes. Notably, the vulnerable object is allocated within the `kmalloc-512` general cache.

As illustrated in Figure 3, DirtyFree comprises two main stages: before and after executing the arbitrary free primitive. In the first stage, DirtyFree begins by spraying kernel heap memory with credential (`cred`) objects holding user privileges (i.e., ❶), preparing the heap for exploitation. Then, we allocate the vulnerable object and the arbitrary free object adjacent. Note that several existing heap manipulation techniques (e.g., heap feng shui [40] or Pspray [41]) reliably achieve this adjacency; thus, we consider this step outside the scope of our paper. Next, an out-of-bounds write vulnerability is triggered to partially overwrite (2 bytes) the pointer within the arbitrary free object (i.e., ❷). Note that an additional information leakage step may be required when the vulnerability does not permit a partial overwrite. Consequently, this pointer no longer points to its original object but instead references one of the sprayed user privilege `cred` objects.

In the second stage, DirtyFree invokes the cross-cache free through the arbitrary free primitive (i.e., ❸), transitioning the targeted user-level `cred` object into a use-after-free state. Subsequently, we spray new `cred` objects containing root privileges across kernel heap memory (i.e., ❹). Due to the use-after-free condition, the previously freed user privilege `cred` object's memory slot becomes available and can thus be reliably reclaimed by one of the newly sprayed root privilege `cred` objects. Consequently, the original user privilege `cred` object is overwritten with the `cred` object carrying elevated privileges. As a result, the attacker's process inherits these overwritten credentials, successfully escalating privileges from an unprivileged user to root.

B. Identifying Arbitrary Free Object

Tracking down free operations. Recall that an arbitrary free primitive requires the presence of a free operation within kernel code. Therefore, our first step is to systematically identify and pinpoint all kernel functions that invoke free operations (e.g., variations of `kfree()`). We specifically focus on `kfree()` and its variants because, as described earlier in §III-C, `kfree()` uniquely supports cross-cache frees by automatically determining the appropriate cache from the provided pointer. Unlike `kmem_cache_free()`, which explicitly requires a cache argument and thus limits cross-cache exploitation, `kfree()` allows an attacker to reliably free kernel objects allocated in different caches.

Next, we determine whether the identified free operation can be triggered by a user-level process. If the free operation is not callable from userspace, it cannot be effectively used as part of an exploitation primitive. To verify this condition, we carefully analyze the kernel functions containing the free operation and systematically trace their call hierarchy to locate the root kernel function. If this root kernel function turns out to be a system call or is reachable from a system call path, we conclude that the corresponding free operation is callable by an unprivileged user. Consequently, such free operations are considered suitable for the arbitrary free primitive.

Lastly, we carefully track down the pointer that is used as an argument of the free operation. Specifically, we analyze whether the pointer involved is a local variable or not. If the pointer is indeed a local variable, we further examine the context in which it is initialized. When such initialization is performed using `kmalloc()`, we consider the corresponding object as *temporary*. In these cases, attackers lack a sufficient time window to perform pointer corruption, making pointers stored in local variables generally unsuitable for arbitrary free primitives.

Extracting arbitrary free object candidates. For identifying arbitrary free object candidates, we first pinpoint the origin of the pointer used as an argument in free operations. Specifically, starting from the taint sources indicating the corresponding pointer, we perform a backward data-flow analysis to examine the parent object that contains the pointer (e.g., the argument `ptr` in Figure 4). If the parent object is allocated in the stack or global memory regions, we exclude it because pointer corruption through heap memory corruption vulnerabilities is impractical in these regions. Conversely, if the parent object is allocated in the heap memory region (e.g., `ptr = ObjA->ptr` in Figure 4), it qualifies as a candidate arbitrary free object due to the feasibility of pointer corruption.

Filtering out arbitrary free object candidates. With the set of candidate objects identified, we proceed to filter them based on their suitability for exploitation. First, we analyze the allocation sites of each candidate object. Since our focus is specifically on vulnerabilities within general caches, objects allocated from

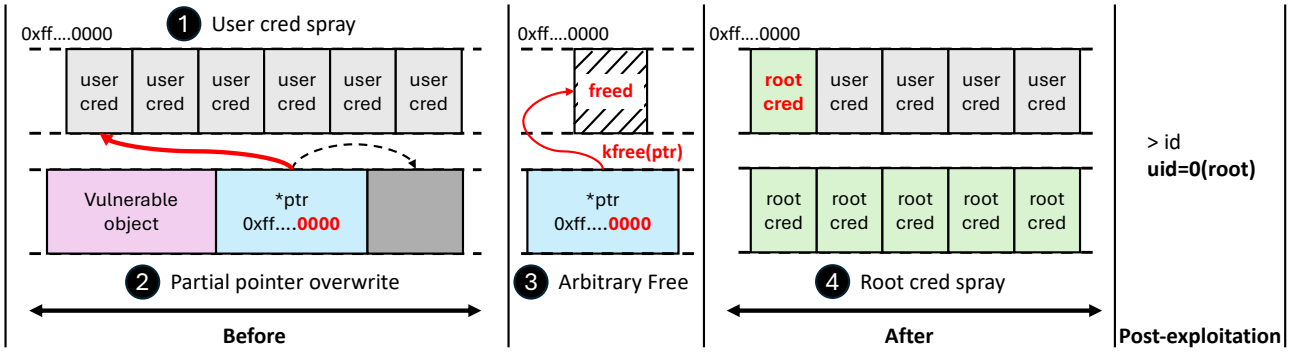

Fig. 3: General overview of exploiting CVE-2021-22555 using an arbitrary free primitive.

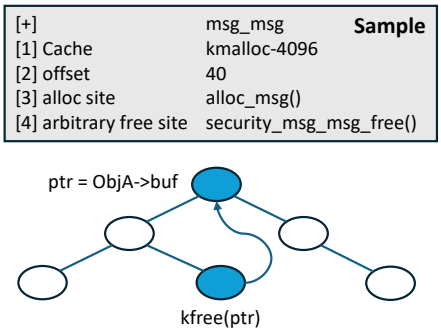

Fig. 4: High-level overview of a backward taint analysis starting from the `kfree()` function, where the argument `ptr` is derived from the field `buf` within the arbitrary free object `ObjA`.

dedicated caches are unsuitable for our exploitation scenario. Therefore, we carefully examine the allocation function and its return type for each candidate. If an object is allocated using `kmem_cache_alloc()`, this indicates allocation from a dedicated cache, and we exclude it from our candidate list. Conversely, objects allocated using variations of `kmalloc()` (i.e., implying general cache allocation) remain as valid candidates for further analysis.

Second, we eliminate temporary allocations from the candidate set. Similar to previous considerations, if a candidate object is allocated and deallocated within the same control-flow path, it lacks a sufficient time window for exploitation. In such scenarios, attackers cannot reliably corrupt the pointer between allocation and freeing operations, making exploitation impractical. Thus, we thoroughly analyze each candidate's allocation and deallocation paths, discarding those confined to a single control-flow path. This filtering step ensures that the remaining objects offer suitable lifetimes necessary for successful exploitation.

However, temporary allocations may still be exploitable in specific scenarios. Particularly, if a kernel-to-user data transfer function (e.g., `copy_to_user()`) occurs between allocation and deallocation, the previously insufficient time window can become exploitable and these temporary objects could become viable targets. This is because kernel-to-user data transfer functions introduce opportunities to extend these time windows through user-space memory access. Attackers can leverage Linux kernel features such as FUSE [42] or userfaultfd [43] to inject custom fault-handling logic into these memory accesses. When the kernel accesses user-space memory via kernel-to-user transfer functions, a user-defined fault handler is triggered. By deliberately introducing delays within this handler, attackers significantly extend the originally inadequate time window, thereby transforming a non-viable temporary object into a practically exploitable one. Apart from such exceptional cases, all other temporary allocations remain excluded from consideration.

Finally, we exclude candidate objects whose allocation or deallocation requires elevated privileges. Under our threat model, we assume an unprivileged attacker without special capabilities or permissions. Thus, any kernel functions involved in allocating candidate objects or performing arbitrary frees must be accessible without privileged rights. If these functions require elevated privileges or specific capabilities, standard user-level attackers cannot invoke them, rendering such objects unsuitable for practical exploitation. Consequently, we systematically remove these privilege-dependent candidates to ensure all remaining objects are exploitable by an unprivileged attacker.

C. Exploit Methodology

To maximize the effectiveness of the arbitrary free primitive, the critical step is selecting the appropriate kernel object to place into a use-after-free state. In our methodology, we specifically target the privilege-related `cred` object, as corrupting this structure directly enables privilege escalation. Based on this choice, we developed a robust exploitation strategy that reliably leverages the compromised `cred` object to escalate privileges.

User-privilege cred spray. Our exploit method begins by systematically spraying user credential (i.e., `cred`) objects within the kernel heap to ensure predictable memory placement. To effectively achieve this, we must allocate as many `cred` objects as possible. However, the allocation process must be selective: if other object types are allocated alongside `cred`, they can interfere with our heap-spraying efforts. Thus, it is critical to find a suitable system call that exclusively allocates

cred objects without introducing unrelated object allocations. Common methods, such as using the fork system call, are unsuitable for this purpose. This not only imposes strict process number limits, which prevent sufficient object spraying, but it also allocates additional object types like task_struct, disrupting our desired heap layout.

To overcome these limitations, we instead use the io_uring interface [44]. Specifically, invoking the capset() system call through the IORING_REGISTER_PERSONALITY flag results in the allocation of new cred objects by duplicating the current credentials without additional unrelated object allocation. Subsequently, calling io_uring_register() with the same IORING_REGISTER_PERSONALITY flag increments the reference counter for these credentials, effectively preventing their immediate deallocation. This method reliably provides a dense and predictable heap spray composed exclusively of user-level cred objects.

Root-privilege cred allocation. The user-privilege cred object freed via the arbitrary free primitive must be hijacked and replaced with a root-privilege cred object to achieve privilege escalation. To accomplish this, we need to systematically allocate kernel objects initialized with root-level credentials. There are multiple methods available to allocate objects containing root privileges. First, executing a setuid-root binary (e.g., su or sudo) triggers the kernel to allocate credential objects initialized with root privileges. Second, interacting with privileged daemon processes (e.g., sshd) that operate under root permissions can similarly result in the allocation of root credentials. Lastly, kernel workqueues can, under specific circumstances, be leveraged as another mechanism to obtain root privileges. Note that we chose the first method, which allows users to easily spray root cred structures.

Post Exploitation. After exploitation, the manipulated cred pointer now references a root-privileged credential object, granting the attacker root-level permissions to a single io_uring instance. However, since the process itself still lacks elevated privileges, additional steps are required to escalate the privileges for the entire process. First, it is necessary to identify which of the sprayed io_uring instances now holds the escalated privileges, and then leverage that instance to achieve complete process-level privilege escalation.

Our approach involves systematically attempting to open a privileged file (e.g., /etc/passwd) with write permissions through each sprayed io_uring instance. An io_uring instance with normal user-level privileges will fail to open this file, while the instance holding elevated privileges will succeed. This method allows us to pinpoint precisely which io_uring instance possesses root privileges. Finally, by writing the string dirtyfree:x:0:0 into the file, we effectively add a new user with root privilege without a password. Finally, by switching to the dirtyfree user, we obtain full root privileges.

V. Implementation

We implemented a prototype of DirtyFree on Linux kernel v6.8 using LLVM/Clang 12, incorporating static analysis techniques to identify arbitrary free object candidates. Specifically, we developed an LLVM pass to systematically locate potential arbitrary free objects throughout the kernel codebase. To facilitate comprehensive static analysis across multiple compilation units, we utilized wllvm [45], which combines multiple compilation units into a single LLVM IR bytecode file. This setup streamlined our analysis, ensuring consistent and accurate identification of candidate objects for subsequent exploitation.

Identifying Arbitrary Free Objects. The key characteristic of an arbitrary-free object is that it contains a pointer explicitly used as an argument to kfree(). To identify such pointers, we first locate all kernel instructions invoking kfree() and extract their pointer arguments. We then perform a backward analysis, traversing each pointer's use–def chain until we reach the source element of a struct type, the parent object. Next, we locate kmalloc() calls that allocate instances of this recovered parent object type, and select the object as a candidate only if such an allocation is found. We limit our analysis to kmalloc() allocations because DirtyFree targets arbitrary-free objects that reside in general caches.

Filtering Temporary Objects. Temporary objects provide almost no exploitable time window and are therefore discarded during candidate identification. For each candidate whose allocation and free occur within the same function, we use standard CFG reachability analysis to determine whether the two instructions (allocation and free) are reachable from one another. If they are, we classify the object as temporary. However, if a kernel-to-user data transfer function (e.g., copy_to_user) lies on any path between the two sites, as verified using the same reachability check, the object remains exploitable as we described in §IV-B and is retained rather than discarded.

VI. Evaluation

In this section, we evaluate the exploitation effectiveness of DirtyFree through a series of experiments and case studies. We study the following research questions:

- **RQ1.** How many arbitrary free objects does DirtyFree collect (§VI-A)?
- **RQ2.** How many vulnerabilities does DirtyFree exploit (§VI-B)?
- **RQ3.** How effective is it compared to other DOP techniques (§VI-C)?

Environment Setting. All experiments were performed inside a QEMU virtual machine (VM) running a Linux v6.8 kernel. The VM was configured with 2 CPUs and 4GB of RAM. For each vulnerability, we reintroduced it into the Linux v6.8 kernel and compiled a corresponding kernel and disk image for the VM. The host system was an Ubuntu 18.04.6 LTS machine with an Intel Xeon Gold 6209U CPU (40 cores) and 32GB of RAM, but all experiments were executed exclusively inside the QEMU environment.

A. Arbitrary Free Object Collection

Table I presents 14 arbitrary free objects that can be used for exploitation across different kernel caches. Our collected

TABLE I: Arbitrary free objects identified and confirmed.

Struct Name	Cache	Offset
Static-size objects		
landlock_hierarchy	kmalloc-16	0
landlock_ruleset	kmalloc-96	16
async_poll	kmalloc-96	64
perf_event_pmu_context	kmalloc-128	96
pipe_inode_info	kmalloc-192	152
mnt_idmap	kmalloc-192	8, 16, 80, 88
msg_queue	kmalloc-256	48, 192
io_ring_ctx	kmalloc-2048	264, 288, 304, 328, 896, 1152, 1308
Dynamic-size objects		
msg_msg	kmalloc-64 ~ kmalloc-4096	32, 40
msg_msgseg	kmalloc-16 ∼ kmalloc-4096	0
sem_array	kmalloc-512 ∼ kmalloc-8192	48
poll_list	kmalloc-16 ∼ kmalloc-4096	0
callchain_cpus_entries	kmalloc-16 ∼ kmalloc-4096	16, 24, ...
simple_xattr	kmalloc-32 ∼ kmalloc-8192	16

objects are applicable to all general slab caches except for `kmalloc-8`, which is rarely used in the kernel. Most kernel caches contain at least one arbitrary free object, making them viable targets for privilege escalation through DIRTYFREE. The table also includes the offset of the critical pointer within each object that can be leveraged by the arbitrary free primitive. This information allows an attacker to easily match a given vulnerability with a suitable arbitrary free object based on the nature and size of the pointer corruption. For example, if a vulnerability causes a 2-byte overflow within the `kmalloc-4096` cache, the `poll_list` object—allocated from the same cache and having a critical pointer at offset 0—would be an ideal candidate.

We found two candidate objects with notable characteristics. First, the object `poll_list` is unique in that its allocation and deallocation occur within the same control-flow path, classifying it as a temporary object. However, in the middle of allocation and deallocation, it invokes `copy_from_user`. As a result, we can control the time window by using either userfaultfd or FUSE, enabling practical exploitation despite the temporary nature of the object. Second, the object `callchain_cpus_entries` is notable due to its variable size, causing it to be allocated in different caches depending on its actual size. However, the object's size is directly tied to the number of CPU cores on the target system, making it a special case beyond user control. Thus, attackers must obtain prior knowledge about the CPU configuration of the targeted system to reliably use this object for exploitation.

False Positive and False Negative. As previously discussed, we rely on static analysis techniques to systematically identify arbitrary free objects. However, static analysis inherently comes with limitations, which can result in false positives (i.e., objects incorrectly identified as candidates) and false negatives, where legitimate candidate objects are missed.

Our tool initially reported 43 candidate objects, of which manual analysis confirmed 14 as true positives. A closer examination of the remaining cases revealed two sources of false positives. First, several candidate objects were manipulated (i.e., allocated or freed) only by kernel functions that require elevated privileges. Because such functions fall outside our attacker model, these objects are unusable and do not represent practical exploitation opportunities. Second, many arbitrary-free sites appeared only in error-handling paths. These paths are difficult (often impossible) for an unprivileged attacker to reach in a precise and repeatable manner, which makes the corresponding objects non-exploitable in realistic scenarios.

False negatives may also occur during our candidate identification process. Specifically, we compile the kernel source code with the optimization option `-O2`, and subsequently generate LLVM bitcode for analysis. However, optimization passes performed by the compiler can sometimes remove object symbols, leading to certain objects lacking identifiable names. Since our candidate selection heavily relies on object names, these unnamed objects cannot be identified and consequently result in false negatives. To mitigate this issue, compiling the kernel with the `-O0` optimization level could help preserve object symbols, thereby significantly reducing false negatives.

Applicability of Arbitrary Free Objects. We evaluated the applicability of our collected arbitrary free objects across Linux kernel versions v5.0 to v6.16, as summarized in Table IV in the appendix. Among the 14 objects, six are usable only starting from specific kernel versions, while one is usable only in versions below a certain threshold. Nevertheless, the remaining objects work across all tested versions and span allocations from `kmalloc-16` to `kmalloc-8192`, covering all general caches except kmalloc-8. These results demonstrate that our collected arbitrary free objects are broadly applicable across Linux kernel versions, offering reliable coverage over almost all general caches.

B. Effectiveness on Real-World Vulnerabilities

To further demonstrate the effectiveness of DIRTYFREE, we evaluate our method using real-world vulnerabilities.

Dataset. We collected a total of 31 Linux kernel vulnerabilities that were submitted to Google's Kernel Capture The Flag (KCTF) [46] over the past two years (2023–2024). Each of these vulnerabilities has publicly available exploits, making them particularly suitable for a practical and realistic evaluation. Note that these vulnerabilities were originally exploited using common exploit techniques such as ROP, which are invalid under our threat model (e.g., due to KCFI). Moreover, five of the vulnerabilities used arbitrary-free only for minor exploitability upgrades [12, 47], rather than as a primary exploitation technique. This insight underscores that DIRTYFREE is the first to use it effectively for privilege escalation. By selecting these publicly documented vulnerabilities, we ensure trans-

TABLE II: Exploitation on real-world vulnerabilities.

				Traditional DOP			DirtyCred	DirtyFree	
CVE	**Type**	**Cache**	**Exploitability**	**Leak**	**AAR**	**AAW**	**Exploit**	**w/o Leak**	**Exploit**
CVE-2023-3390	Use-After-Free	kmalloc-*	[*:*)=*	✔	✔	✔	✗	✗	✔
CVE-2023-3611	Out-Of-Bounds	kmalloc-8192	[0x1d78:0x1d80)=bitflip	✔	✔	✔	✗	✔	✔
CVE-2023-3776	Use-After-Free	kmalloc-128	Read	✗	✗	✗	✗	✗	✗
CVE-2023-3777	Use-After-Free	kmalloc-128	[0x58:0x5c)-=1	✗	✗	✗	✔	✗	✔
CVE-2023-4004	Double-Free	kmalloc-*	[*:*)=*	✔	✔	✔	✗	✔	✔
CVE-2023-4015	Use-After-Free	kmalloc-128	[0x58:0x5c)-=1	✗	✗	✗	✔	✗	✔
CVE-2023-4147	Use-After-Free	kmalloc-128	[0x58:0x5c)-=1	✗	✗	✗	✔	✗	✔
CVE-2023-4206	Use-After-Free	kmalloc-512	Read	✗	✗	✗	✗	✗	✗
CVE-2023-4207	Use-After-Free	kmalloc-512	Read	✗	✗	✗	✗	✗	✗
CVE-2023-4208	Use-After-Free	kmalloc-512	Read	✗	✗	✗	✗	✗	✗
CVE-2023-4244	Use-After-Free	kmalloc-128	[0x58:0x5c)-=1	✗	✗	✗	✔	✗	✔
CVE-2023-4569	Use-After-Free	kmalloc-256	[0x30:0x34)-=1	✔	✔	✗	✔	✗	✔
CVE-2023-4622	Use-After-Free	skbuff_head_cache	Read	✗	✗	✗	✗	✗	✗
CVE-2023-4623	Use-After-Free	kmalloc-1024	Read	✗	✗	✗	✔	✗	✔
CVE-2023-5197	Use-After-Free	kmalloc-128	[0x58:0x5c)-=1	✗	✗	✗	✔	✗	✔
CVE-2023-52620	Use-After-Free	kmalloc-128	[0x58:0x5c)-=1	✗	✗	✗	✔	✗	✔
CVE-2023-5345	Double-Free	kmalloc-*	[*:*)=*	✔	✔	✔	✗	✔	✔
CVE-2023-6111	Double-Free	kmalloc-*	[*:*)=*	✔	✔	✔	✗	✔	✔
CVE-2023-6817	Use-After-Free	kmalloc-192	[0x34:0x38)-=1	✔	✔	✔	✔	✗	✔
CVE-2023-6931	Out-Of-Bounds	kmalloc-2048	[0x810:0x818)+=1	✗	✗	✗	✔	✗	✗
CVE-2024-0193	Use-After-Free	kmalloc-256	[0x34:0x38)-=1	✔	✔	✗	✔	✗	✔
CVE-2024-1085	Double-Free	kmalloc-256	[*:*)=*	✔	✔	✗	✗	✔	✔
CVE-2024-1086	Double-Free	kmalloc-256	[*:*)=*	✔	✔	✗	✗	✔	✔
CVE-2024-26581	Double-Free	kmalloc-256	[*:*)=*	✔	✔	✗	✗	✔	✔
CVE-2024-26642	Use-After-Free	kmalloc-256	[0x34:0x38)-=1	✔	✔	✗	✔	✗	✔
CVE-2024-26809	Double Free	kmalloc-256	[*:*)=*	✔	✔	✗	✗	✔	✔
CVE-2024-26925	Double Free	kmalloc-256	[*:*)=*	✔	✔	✗	✗	✔	✔
CVE-2024-36972	Double Free	skbuff_head_cache	[*:*)=*	✗	✗	✗	✗	✔	✗
CVE-2024-39503	Use-After-Free	kmalloc-192	Read	✔	✔	✔	✔	✗	✔
CVE-2024-41010	Use-After-Free	kmalloc-2048	[0:8)=pointer or [0:8)=0	✗	✗	✔	✗	✗	✔
CVE-2024-53141	Out-Of-Bounds	kmalloc-*	[0:8)=value or [0:8)=pointer	✗	✗	✔	✔	✗	✔

parency and reproducibility of our evaluation process, clearly demonstrating the practical effectiveness of DirtyFree.

Result. Table II summarizes our evaluation results against real-world vulnerabilities. As demonstrated, DirtyFree successfully achieved privilege escalation on 24 out of 31 vulnerabilities, highlighting its effectiveness as a robust and general exploitation technique. Among these, only 10 vulnerabilities can be exploited without information leakage because they support partial pointer overwrites. This overwrite capability enables DirtyFree to complete the exploitation in a single step without relying on any address disclosure. All remaining vulnerabilities lack such capability and therefore require an information leak. In cases where vulnerabilities were successfully exploited, some required only their inherent exploitability. However, in cases where a single vulnerability was insufficient for direct exploitation, we utilized its exploitability to trigger a secondary vulnerability, ultimately achieving successful exploitation.

Case Studies. CVE-2024-53141 is an out-of-bounds write vulnerability. Interestingly, this vulnerability triggers out-of-bounds conditions in two distinct ways. One overwrites a pointer value, while the other overwrites a user-controlled value. However, the value overwrite is performed as an 8-byte operation, making partial overwrites infeasible. We utilized the pointer overwrite capability to leak a heap address, from which we inferred the location of sprayed cred objects. Then, leveraging the value overwrite, we replaced the inferred cred location with an address for an arbitrary free. Consequently, this reliably resulted in targeting a cred object, ultimately achieving privilege escalation with high probability.

CVE-2023-6817 and CVE-2024-26642 are use-after-free write vulnerabilities involving the `nft_chain` object, and thus share the same exploitability characteristics. Specifically, these vulnerabilities enable attackers to decrement a reference counter within the object, allowing controlled modification of internal values. Exploiting this property, we targeted an object containing a size field, repeatedly decreasing the reference counter to trigger an integer underflow. We then leveraged the corrupted size value to trigger an out-of-bounds write, which enabled us to overwrite an arbitrary free object involved in an arbitrary free operation. Consequently, we achieved privilege escalation by targeting and freeing a carefully positioned cred object.

CVE-2024-39503 is a use-after-free vulnerability that does not involve any write operation. However, it leverages a read operation to retrieve a value subsequently used as an offset. Specifically, it reads a pointer at an offset from another object's base pointer, and then invokes `kfree` on the retrieved pointer An attacker who controls the offset can hence trigger an arbitrary free operation at a desired memory address. We exploited this behavior to leak a heap pointer, and based on this information, successfully performed an arbitrary free targeting a previously sprayed cred object, ultimately achieving privilege escalation.

We employed an identical exploitation approach for all double-free vulnerabilities. The exploitation consisted of two

distinct stages, triggering the double-free condition twice. In the first stage, we leveraged the double-free to place an object capable of leaking memory into a use-after-free state, subsequently aliasing it with an object containing a heap pointer to leak a heap address. In the second stage, we again used the double-free to place an arbitrary-free object into a use-after-free state and allocated an object allowing user-controlled writes into the freed slot. Finally, we triggered the arbitrary free primitive targeting a cred object, achieving privilege escalation.

Failure Cases. The failure cases can be broadly classified into two categories. The first category involves vulnerabilities that lack sufficient exploitability to meet the conditions required for arbitrary free. Despite an in-depth analysis, these cases were unexploitable since overwriting pointers within the arbitrary free object was infeasible. The second category comprises vulnerabilities involving use-after-free conditions within dedicated caches. While prior exploits successfully leveraged temporal cross-cache attacks, our assumed threat model explicitly excludes temporal cross-cache attacks, making exploitation impossible under these constraints.

C. Comparison with other DOP techniques

To compare DIRTYFREE with other DOP techniques, we first surveyed real-world exploitation methods that fall within the DOP paradigm. Most candidate techniques (e.g., `core_pattern`) do not clearly describe the exploitation requirements or object constraints needed for reproduction. Only traditional DOP [11, 12] and DirtyCred [16] provide sufficient detail to evaluate applicability. We therefore excluded the remaining techniques from our comparison, since they cannot be assessed in a systematic or reproducible way. For traditional DOP and DirtyCred, we assessed applicability by checking whether each vulnerability satisfied the object requirements defined in their respective work. This approach is necessary because none of these three techniques provides proof-of-concept code for their identified objects, which makes empirical reproduction challenging.

Traditional DOP. Among the tested vulnerabilities, only six satisfied the three primitives that constitute traditional DOP. The first limitation arises from the strict preconditions associated with each primitive. Since every vulnerabilities typically show a limited exploitability, it is rare for any single vulnerability to satisfy all three primitive conditions simultaneously. In contrast, DIRTYFREE requires meeting only a single primitive in special case, allowing it to apply to a significantly broader set of vulnerabilities.

The second limitation is that Eloise and BridgeRouter identify objects only in specific caches. As a result, several vulnerabilities became unexploitable regardless of their inherent exploitability because the required objects were simply unavailable in the caches they exercised. DIRTYFREE, by contrast, identifies arbitrary-free objects across *all* caches used by vulnerabilities, thereby eliminating failures caused by incomplete object coverage.

DirtyCred. Our evaluation shows that 14 vulnerabilities can be exploited using the DirtyCred technique. For out-of-bounds and use-after-free cases, its performance closely matches that of DIRTYFREE, demonstrating that both techniques are effective in these scenarios. The only deviation is CVE-2023-3611, which occurs in `kmalloc-8192`. In this case, DirtyCred was unable to identify a suitable object within the same cache, whereas DIRTYFREE succeeded. A more notable distinction emerges for double-free vulnerabilities. DirtyCred depends on cross-cache behavior to exploit such cases, but this strategy is incompatible with our threat model, which incorporates *slab-virtual* as defense and therefore prevents cross-cache interactions. As a result, DirtyCred cannot exploit *any* of the double-free vulnerabilities, while DIRTYFREE remains effective under the same constraints.

VII. DEFENSE AGAINST DIRTYFREE

In this section, we propose a set of defense mechanisms aimed at mitigating the exploitation capabilities of DIRTYFREE. Recall that DIRTYFREE exploits two primary factors within the Linux kernel's memory management. First, arbitrary free objects are currently not isolated, allowing vulnerabilities occurring in general caches to directly utilize and manipulate these objects. Consequently, attackers can exploit these vulnerabilities to transition security-critical objects, typically managed in dedicated caches, into vulnerable use-after-free states. Second, the `kfree` function does not perform cache validation checks, enabling attackers to perform cross-cache frees. As a result, objects allocated within dedicated caches can inadvertently be freed through general cache vulnerabilities, significantly increasing exploitability.

Considering these factors, one might consider adopting one of the following two approaches:

1) Allocate arbitrary free objects in isolated, dedicated caches to prevent their exploitation by DIRTYFREE.
2) Enforce strict cache validation checks prior to calling `kfree()`, effectively preventing unintended cross-cache frees.

We explore both approaches. Importantly, each mitigation is sufficient on its own to block DIRTYFREE, so deploying either one is enough to prevent exploitation.

A. Mitigation #1: Isolating Arbitrary Free Object

Design. To mitigate the threat posed by arbitrary free objects, we isolate these objects into dedicated caches. Specifically, we create a new dedicated cache for each type of object, such as `io_uring_ctx_cache`. In other words, each of our newly created dedicated caches exclusively manages a single object type. This approach is intended to prevent scenarios where exploitation becomes feasible due to different types of objects coexisting within a single cache.

For static-sized objects, creating new dedicated caches is sufficient; however, dynamic-sized objects require a different approach due to their variable sizes. This limitation occurs because standard dedicated caches, allocated via `kmem_cache_alloc`, can only handle fixed-size objects. To address this issue, we adopt the slab bucket approach [48], a mitigation specifically designed to allow dedicated-cache

```
/* Implemented Code Begin */
void validate_af(void *obj)
{
    if (virt_addr_valid(obj))
    {
        struct slab *slab = virt_to_slab(obj);
        if(strstr(slab->slab_cache->name, "kmalloc"))
        {
            panic("Cross-Cache Free is detected!");
        }
    }
}
/* Implemented Code End */

void free_msg(struct msg_msg *msg)
{
    ...
    validate_af(msg);
    kfree(msg); // Arbitrary Free Primitive
    ...
}
```

Fig. 5: An example code of cache validation mitigation.

allocation for dynamic-sized kernel objects. When utilizing the slab bucket mechanism, the kernel creates dedicated caches of varying sizes at cache initialization time through the `kmem_buckets_create` function, making it highly suitable for handling dynamic-sized arbitrary free objects. Note that enabling this feature requires compiling the Linux kernel with the `CONFIG_SLAB_BUCKET` configuration option.

Implementation. For static-sized arbitrary free objects, we directly modified the Linux kernel source code to allocate these objects into dedicated caches. Specifically, we introduced new dedicated caches by invoking `kmem_cache_create` for each static-sized arbitrary free object type. To ensure proper allocation, we replaced the original allocation routines with `kmem_cache_alloc`, explicitly placing objects within their corresponding dedicated caches. Likewise, we updated relevant kernel code paths to use `kmem_cache_free` for object deallocation, thus effectively enforcing the isolation of arbitrary free objects.

In contrast, for dynamic-sized arbitrary free objects, we modified the kernel source code to utilize bucket caches. Specifically, we introduced a new function that creates bucket caches by calling `kmem_buckets_create` during kernel initialization. Additionally, we changed the object allocation function from `kmalloc` to `kmem_buckets_alloc`. Note that we did not modify the deallocation logic, as the bucket cache mechanism currently lacks a dedicated free function.

B. Mitigation #2: Cache Validation

Design. The second method focuses on preventing cross-cache frees. Recall that cross-cache free exploits rely on the fact that `kfree` determines the appropriate cache based solely on the pointer provided as input, allowing attackers to illegally free objects allocated in dedicated caches. To address this issue, we add validation logic immediately before invoking `kfree`, ensuring the object being freed indeed belongs to a general cache. Specifically, our validation checks the associated cache metadata for the object to confirm it matches an expected general cache structure, thereby preventing unintended or malicious cross-cache operations. If the kernel is structured correctly, `kfree` should only free objects allocated within general caches. As a result, any attempt to free objects residing in dedicated caches can be detected and blocked by our validation logic, significantly reducing exploitability.

Implementation. As shown in Figure 5, we implemented a prototype of the proposed cache validation mitigation in the Linux kernel. To implement this method, we insert a validation step before each call to `kfree` on arbitrary free objects. We begin by using `virt_to_slab` to retrieve the corresponding `slab` structure from the object's pointer. From this structure, we extract the `slab_cache` field to identify the cache the object belongs to. We then examine the cache name to determine whether the object resides in a general cache or a dedicated cache. This check allows us to enforce that only general cache objects are passed to `kfree`, preventing potential cross-cache free.

C. Evaluation for Mitigations

To demonstrate that our proposed mitigations are effective in multiple aspects (i.e., security, performance, and regression), we conducted a comprehensive evaluation.

Security Evaluation. We evaluated whether the two proposed mitigations could effectively prevent exploitation via DIRTYFREE. In conclusion, we found that all 24 vulnerabilities previously exploitable using DIRTYFREE were successfully mitigated, effectively blocking privilege escalation through this approach.

Specifically, Mitigation #1 prevents arbitrary free objects from being utilized with vulnerabilities arising in general caches. In other words, arbitrary free objects can no longer be aliased or positioned adjacent to vulnerable objects. This is because each arbitrary free object now resides in its own dedicated cache, preventing them from sharing memory pages with general cache objects. Previously, temporal cross-cache attacks could circumvent such limitations, but under our defined threat model—where cross-cache attacks are explicitly disallowed—no alternative bypass methods remain.

Mitigation #2 specifically blocks cross-cache free operations. Unlike Mitigation #1, arbitrary free objects can still be aliased or adjacent to vulnerable objects. However, it prevents cross-cache frees targeting dedicated caches containing security-critical objects by enforcing additional verification checks. As cross-cache free is a critical component of DIRTYFREE, this mitigation directly undermines its effectiveness. Consequently, attackers can no longer leverage the efficiency offered by DIRTYFREE and must resort instead to more complex and less efficient traditional DOP techniques.

Performance Evaluation. We evaluate the performance impact of our mitigations across three classes of workloads. First, we use systemd-analyze [49] to measure any overhead during kernel and user-space initialization. Second, we run LMbench [50] to quantify microarchitectural latency across core kernel operations such as system calls, file I/O, and IPC.

TABLE III: Performance comparison of the Linux kernel without mitigation and with our two proposed mitigations applied. For latency metrics, lower values indicate better performance, whereas for throughput metrics, higher values indicate better performance.

Test	w/o miti	Miti #1	Overhead	Miti #2	Overhead
Systemd-Analyze (s)					
Kernel	1.210	1.265	+4.55%	1.225	+1.24%
User	1.005	1.009	+0.40%	1.006	+0.10%
Geomean			**+2.46%**		**+0.67%**
LMBench - latency (ms)					
syscall	0.8737	0.8869	+1.52%	0.8802	+0.75%
read	0.8971	0.8953	-0.20%	0.8716	-2.85%
write	0.5536	0.5422	-1.98%	0.5438	-1.77%
stat	0.9503	0.9496	-0.07%	0.9698	+2.05%
fstat	0.7658	0.7602	-0.74%	0.7411	-3.25%
open/close	1.7740	1.7853	+0.64%	1.7825	+0.48%
select (10 fd)	0.5834	0.5857	+0.40%	0.5884	+0.86%
select (100 fd)	1.4468	1.5038	+3.95%	1.4326	-0.98%
pipe	8.4400	8.4365	-0.04%	8.3035	-1.60%
fork+exit	104.5472	104.8061	+0.25%	105.6737	+1.07%
fork+execve	338.5625	339.9375	+0.41%	340.2286	+0.50%
fork+/bin/sh	867.8571	855.7143	-1.41%	868.3077	+0.05%
UNIX sock	11.8290	11.9978	+1.43%	11.7161	-0.99%
UDP	9.0979	9.0724	-0.28%	9.3117	+2.36%
TCP	12.7894	12.7601	-0.23%	12.8181	+0.22%
Geomean			**+0.23%**		**-0.22%**
Application Benchmarks - Throughput					
Nginx (req/s)	26054.9	26949.7	-3.43%	26971.3	-3.52%
Apache (req/s)	12913.9	12691.8	+1.72%	13299.6	-2.99%
lighttpd (req/s)	28435.29	29274.00	-2.95%	28673.51	-0.84%
Redis - Set (ops/s)	94744.10	96525.09	-1.88%	96525.09	-1.88%
Redis - Get (ops/s)	93766.61	96153.85	-2.55%	95057.03	-1.38%
memcached (ops/s)	91721.80	93340.85	-1.77%	93467.88	-1.90%
Geomean			**-1.80%**		**-2.08%**

Finally, we assess end-to-end application performance using five macrobenchmarks (i.e., Nginx, Apache, lighttpd, Redis, and memcached), reporting their throughput under static or in-memory workloads. All experiments are repeated three times, and we report the geometric mean. The results are summarized in Table III.

Overall, we observe that both mitigations introduce negligible performance overhead. Boot-time measurements using systemd-analyze show only minor increases in kernel and user-space initialization times. The small kernel-side overhead in Mitigation #1 simply comes from creating dedicated caches for arbitrary-free objects during boot. LMbench results show that all measured latency values remain within normal measurement noise. In addition, all application benchmarks also show no meaningful throughput degradation. Together, these results confirm that our mitigations are practical, as the added logic applies only to identified objects and therefore does not interfere with common fast paths in the kernel.

Regression Evaluation. To verify that our proposed mitigations do not affect overall system stability, we performed stress tests using the Linux Test Project (LTP) [51]. These tests are designed to stress various kernel subsystems and detect potential regressions. We ran the full LTP test suite on two kernels, each incorporating one of our proposed mitigations. All tests completed successfully without any kernel crashes, hangs, or mitigation-induced functional deviations. This result is consistent with the design of both mitigations. Mitigation #1 merely changes the allocation site from `kmalloc()` to `kmem_cache_alloc()`, following the standard kernel practice used when moving objects from general caches to dedicated slab caches. Therefore, it does not alter object semantics or introduce unintended side effects. Mitigation #2 adds a cache-identifier lookup before `kfree()`, but this operation relies solely on slab metadata that the kernel already maintains for pointer-to-cache resolution, and thus does not introduce new behavior or additional failure modes.

VIII. Discussion

Next, we discuss additional issues that have not yet been addressed and outline potential directions for future work.

A. Noise

Like other exploitation techniques, DirtyFree is also susceptible to interference caused by system noise. First, interference may occur during the layout manipulation phase, where the attacker attempts to place the vulnerable object adjacent to or aliased with the arbitrary-free object. In such cases, concurrent memory allocations by other processes may disrupt the intended memory layout. Second, after the arbitrary free of a user-level `cred` structure is triggered, privilege escalation may fail if another process allocates a `cred` object *before* a root `cred` allocation occurs. Although various techniques, such as CPU pinning, can be used to avoid such interference, we did not cover them in this work. Nonetheless, applying these techniques could further reduce the impact of noise and improve the reliability of DirtyFree-based exploitation.

B. Other Security-Critical Objects

We believe that other security-critical objects can also serve as potential targets for DirtyFree. In this paper, we focused on the `cred` structure as our security-critical object and demonstrated a reliable method to achieve privilege escalation through its corruption. However, the same methodology could be applied to alternative objects whose compromise similarly enables privilege escalation. For instance, security-critical objects identified in DirtyCred [16], which allow arbitrary file writes via file name swapping, could also be targeted. More broadly, we anticipate that both publicly known and yet-to-be-discovered security-critical objects may be viable targets for DirtyFree, provided that suitable arbitrary-free primitives can be identified or constructed.

C. Other Operating Systems

We expect that DirtyFree can also be applied to UNIX-based systems such as macOS and FreeBSD. Both platforms already feature arbitrary-free primitives, and their heap management designs are similar to Linux's SLUB allocator. A notable difference is their use of *zones* instead of caches: security-critical objects are isolated in dedicated zones, yet cross-zone frees via `kfree` or `free` remain possible. Thus, adapting DirtyFree would primarily require identifying suitable security-critical objects for privilege escalation. In such cases, cross-zone frees could offer a practical path to elevated privileges.

D. Limitations

Static Analysis. Although our identification procedure is systematic, it is intentionally conservative. In particular, our temporary-object filtering relies on a reachability heuristic that may over-approximate certain cases, potentially leading to false negatives. This limitation stems from the analysis design rather than implementation details. Incorporating more path-sensitive reasoning could further broaden the set of arbitrary-free objects supported by DIRTYFREE.

Applicability of Attack. DIRTYFREE shares the same general limitation with all exploitation techniques: successful exploitation requires a match between the conditions of the target object and the exploitability constraints of the underlying vulnerability. If this matching does not hold, the vulnerability cannot be exploited using DIRTYFREE. This limitation is inherent to any technique that relies on object reallocation semantics and is not unique to our approach.

Defenses. The limitations of our mitigations stem from their reliance on identifying arbitrary-free objects. Our defenses assume that all such objects are known so they can either be isolated into dedicated caches or validated through additional checks; if new arbitrary-free objects are discovered in the future, further updates would be required to extend protection. However, this maintenance requirement is not unique nor unusual—OS defenses commonly rely on incremental updates. For example, `zone_require` [52], which prevents temporal cross-zone attacks in macOS and iOS, must be updated as new security-critical objects are identified. Our mitigations fall into the same category: practical, incremental defenses whose effectiveness depends on periodically incorporating newly discovered objects.

IX. RELATED WORK

Control-Flow Exploits. Many existing works [13, 53–61] have explored how to achieve privilege escalation when control-flow hijacking is possible. For instance, FUZE [13] automated the analysis of exploitability for use-after-free vulnerabilities to determine whether they could lead to control-flow hijacking. Koobe [53] automated the analysis of out-of-bounds vulnerabilities, identifying precisely which memory regions could be overwritten and with what values. Retspill [54] demonstrated that, given a control-flow hijacking primitive, userspace data could persist in the kernel stack and leveraged this property for privilege escalation. Finally, SRH [55] identified exploitable system registers and devised methods for achieving privilege escalation by utilizing gadgets capable of controlling these registers. However, these techniques have all been mitigated following the introduction of kernel control-flow integrity [1].

Data-Flow Exploits. Existing works have explored diverse techniques [11, 12, 16–20, 62] related to data-oriented programming. Traditional DOP (as instantiated by Eloise [11] and BridgeRouter [12]) reconstructs a three-stage exploitation chain consisting of an information leak, an arbitrary read primitive, and an arbitrary write primitive. In practice, however, it is rare for a single vulnerability to satisfy all three prerequisites, which fundamentally limits the applicability of traditional DOP. In contrast, DIRTYFREE does not require building a full multi-stage chain. Instead, DIRTYFREE often requires only a single primitive (arbitrary free) when partial overwrites are available, enabling exploitation of vulnerabilities that cannot meet all three traditional DOP prerequisites. DirtyCred [16] demonstrated that the file structure can be repurposed to achieve privilege escalation using a data-flow-oriented approach. While its technique is applicable to certain OOB and UAF cases, it fundamentally relies on temporal cross-cache behavior to exploit double-free bugs, which is incompatible with hardened settings such as slab-virtual. Roughly 30% of recent kCTF vulnerabilities fall into this category, placing them outside of DirtyCred's scope. In contrast, DIRTYFREE remains applicable even under slab-virtual: because DIRTYFREE performs cross-cache frees via *kfree()* rather than relying on temporal allocator behavior, it can exploit double-free vulnerabilities that DirtyCred cannot handle under the same threat model.

Reliability of Exploits. Various studies [40, 41, 63–65] have been conducted to improve exploit reliability. Heap feng shui [40] proposed techniques to precisely control the heap layout, positioning the vulnerable and target objects adjacent to each other by performing heap grooming and intentionally creating holes. Kyle *et al.* [64] recognized reliability issues associated with kernel heap exploitation and systematically studied combinations of various exploit techniques to determine the most effective methods across different environments. Pspray [41] employed timing side-channels to detect when new memory pages are allocated, significantly increasing the reliability of exploiting out-of-bounds and use-after-free vulnerabilities. ExpRace [63] addressed scenarios where certain race conditions fail to trigger due to very short race windows; it leveraged performance degradation attacks to extend these race windows, thus improving exploitation success rates. All these techniques aim at enhancing vulnerability reliability, making them orthogonal and complementary to our research.

X. CONCLUSION

Until now, the arbitrary free primitive has primarily served as an auxiliary technique used to enhance the exploitability of vulnerabilities with very limited inherent exploitability. As a result, publicly known arbitrary free objects have remained scarce, and the full potential of the arbitrary free primitive has not been widely recognized. In this paper, we conducted a systematic analysis to identify arbitrary-free objects in general kernel caches and introduced DIRTYFREE, demonstrating that, under suitable conditions, privilege escalation can be achieved with only a single arbitrary free primitive. Upon examining existing kernel defense mechanisms, we found that none provide an effective or practical mitigation against the threat posed by DIRTYFREE. To address this gap, we proposed two targeted mitigations specifically designed to mitigate DIRTYFREE. We conclude that, by effectively preventing the misuse of arbitrary free objects and eliminating cross-cache free behavior, these defenses can significantly reduce the overall threat surface and impact of DIRTYFREE.

XI. Ethical Considerations

The exploitation technique presented in this paper is intended solely to advance the security of the Linux kernel and is not meant to support or enable malicious activity. All kernel vulnerabilities used in our experiments were obtained from Google's Kernel CTF [46] repository and have already been patched prior to our evaluation. No public disclosure occurred prior to our evaluation. Furthermore, all experiments were conducted within isolated virtual environments, ensuring that no public or live systems were affected. We intentionally avoid disclosing any details that could pose a security risk to the Linux kernel or its users. Finally, we release all data and source code used in this study at https://github.com/MPI-SysSec/DirtyFree to facilitate reproducibility and responsible research.

Acknowledgement

We thank all reviewers for their thoughtful feedback. This work was supported by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA – 390781972).

References

[1] Google, "Kernel control flow integrity," 2022, https://source.android.com/docs/security/test/kcfi.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[3] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2007.

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

[5] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[6] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.

[7] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, Aug. 2005.

[8] B. Johannesmeyer, A. Slowinska, H. Bos, and C. Giuffrida, "Practical data-only attack generation," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, Aug. 2024.

[9] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Security Symposium*, Washington, DC, Aug. 2015.

[10] J. Pewny, P. Koppe, and T. Holz, "Steroids for doped applications: A compiler for automated data-oriented programming," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[11] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2020.

[12] D. Xie, D. He, W. You, J. Huang, B. Liang, S. Gan, and W. Shi, "Bridgerouter: Automated capability upgrading of out-of-bounds write vulnerabilities to arbitrary memory write primitives in the linux kernel," in *Proceedings of the 46th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2025.

[13] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, Aug. 2018.

[14] J. Hu, J. Zhou, Q. Tang, and W. Shen, "Pagejack: A powerful exploit technique with page-level uaf," 2024, https://i.blackhat.com/BH-US-24/Presentations/US24-Qian-PageJack-A-Powerful-Exploit-Technique-With-Page-Level-UAF-Thursday.pdf.

[15] Theori, "Reviving the modprobe_path technique: Overcoming search_binary_handler() patch," 2025, https://theori.io/blog/reviving-the-modprobe-path-technique-overcoming-search-binary-handler-patch.

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

[17] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[18] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, and S. Mangard, "Slubstick: Arbitrary memory writes through practical software cross-cache attacks within the linux kernel," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, Aug. 2024.

[19] G. Ziyi, D. K. Le, Z. Lin, K. Zeng, R. Wang, T. Bao, Y. Shoshitaishvili, A. Doupé, and X. Xing, "Take a step further: Understanding page spray in linux kernel exploitation," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, Aug. 2024.

[20] N. Wu, "Dirty pagetable: A novel exploitation technique to rule linux kernel," 2024, https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[21] M. Rizzo, "introduce config_slab_virtual." 2023, https://lore.kernel.org/lkml/20230915105933.495735-9-matteorizzo@google.com/#r.

[22] Google, "Kernel exploits recipes notebook." 2022, https://docs.google.com/document/d/1a9uUAISBzw3ur1aLQqKc5JOQLaJYiOP5pe_B4xCT1KA/edit.

[23] "Slab allocator," https://www.kernel.org/doc/gorman/html/understand/understand011.html.

[24] Corbet, "The slub allocator," 2007, https://lwn.net/Articles/229984/.

[25] M. Mackall, "slob: introduce the slob allocator," 2005, https://lwn.net/Articles/157944/.

[26] J. Kim, J. Park, Y. Lee, C. Song, T. Kim, and B. Lee, "Petal: Ensuring access control integrity against data-only attacks on linux," in *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Oct. 2024.

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

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

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

[30] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 29th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

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

[32] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs." in *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2022.

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

[34] K. Cook, "Kernel address space layout randomization," *Linux Security Summit*, 2013.

[35] Intel, "Supervisor mode execution prevention," https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/servers/platforms/intel-pentium-silver-and-intel-celeron-processors-datasheet-volume-1-of-2/005/intel-supervisor-mode-execution-protection-smep/.

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

[37] Jonathan Corbet, "The current state of kernel page-table isolation," 2017, https://lwn.net/Articles/741878/.

[38] h0mbre, "Escaping the google kctf container with a data-only exploit," https://h0mbre.github.io/kCTF_Data_Only_Exploit/.

[39] Nguyen, "Cve-2021-22555: Turning `\x00\x00` into $10000," https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html#vulnerability.

[40] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, vol. 2007, 2007.

[41] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "Pspray: Timing side-channel based linux kernel heap exploitation technique," in *Proceedings of the 32nd USENIX Security Symposium*, Anaheim, CA, Aug. 2023.

[42] Linux, "Fuse's introduction in the linux kernel user's and administrator's guide." 2022, https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[43] Linux, "Userfaultfd's introduction in the linux kernel user's and administrator's guide." 2022, https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.

[44] pql, "Fourchain - kernel." 2022, https://org.anize.rs/HITCON-2022/pwn/fourchain-kernel.

[45] "Whole-program llvm," https://github.com/travitch/whole-program-llvm.

[46] Google, "Google kernel ctf," https://github.com/google/security-research/tree/master/kernelctf.

[47] H. Zhang, J. Liu, J. Lu, S. Chen, T. Han, B. Zhang, and X. Gong, "Reviving discarded vulnerabilities: Exploiting previously unexploitable linux kernel bugs through control metadata fields," in *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS)*, Oct. 2025.

[48] A. M. Kees Cook, "Introduce dedicated bucket allocator." 2024, https://lore.kernel.org/lkml/202403251609.1F681B5D@keescook/t/.

[49] "Systemd-analyze," https://www.freedesktop.org/software/systemd/man/systemd-analyze.html.

[50] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis."

[51] P. Larson, "Testing linux with the linux test project," in *Ottawa Linux Symposium*, vol. 108, 2002.

[52] B. Azad, "The core of Apple is PPL: Breaking the XNU kernel's kernel," 2020, https://googleprojectzero.blogspot.com/2020/07/the-core-of-apple-is-ppl-breaking-xnu.html.

[53] W. Chen, X. Zou, G. Li, and Z. Qian, "Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium*, Boston, MA, Aug. 2020.

[54] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting user-controlled data to burn away linux kernel protections," in

Proceedings of the 32nd USENIX Security Symposium, Anaheim, CA, Aug. 2023.

[55] J. Miller, M. Ghandat, K. Zeng, H. Chen, A. H. Benchikh, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "System register hijacking: Compromising kernel integrity by turning system registers against the system," in *Proceedings of the 34th USENIX Security Symposium*, Aug. 2025.

[56] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, Aug. 2014.

[57] W. Yong, "Ret2page: The art of exploi4ng use-after-free vulnerabili4es in the dedicated cache," *Black Hat USA*, 2022.

[58] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks," in *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, Aug. 2012.

[59] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014.

[60] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2017.

[61] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, Aug. 2019.

[62] S. Han, S.-J. Kim, W. Shin, B. J. Kim, and J.-C. Ryou, "Page-oriented programming: Subverting control-flow integrity of commodity operating system kernels with non-writable code pages," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, Aug. 2024.

[63] Y. Lee, C. Min, and B. Lee, "Exprace: Exploiting kernel races through raising interrupts," in *Proceedings of the 30th USENIX Security Symposium*, Online, Aug. 2021.

[64] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for K(H)eaps: Understanding and improving Linux kernel exploit reliability," in *Proceedings of the 31st USENIX Security Symposium*, Aug. 2022.

[65] Y. Chen and X. Xing, "Slake: facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

APPENDIX

A. *Reliability of* DIRTYFREE

To demonstrate the reliability of DIRTYFREE, we conducted additional experiments that measure its success rate in isolation. To this end, we constructed a synthetic arbitrary-free object. For this purpose, we add a dedicated system call that provides three controlled operations as shown in Figure 6. The first operation allocates a synthetic object and sets its pointer field to the address of another allocated object. The second operation overwrites the lower two bytes of the corresponding pointer. Lastly, we validate that the corrupted pointer now refers to the sprayed user-privilege `cred` object. We intentionally avoid triggering the actual arbitrary free to prevent a kernel panic, which would interfere with the success-rate measurement. Each experiment consists of 10 rounds, and each round performs 10,000 exploit attempts under two conditions: an idle system and a busy system stressed using `stress-ng`.

Result. We observe that DIRTYFREE achieves a success rate of 95.6% in the idle state and 87.4% in the busy state. Our manual analysis of the failure cases reveals two primary reasons. First, in both settings, some failures occur when the corrupted pointer happens to land on the same page as the original pointer, which prevents the intended redirection. Second, in the busy environment, we confirmed that continuous process creation can temporarily interfere with credential spraying, although this effect is limited and does not lead to substantial degradation.

Discussion. Note that we intentionally do not evaluate reliability using real-world CVE exploits. Such exploits often fail for reasons that are orthogonal to the correctness of DIRTYFREE, such as stabilization effects, object-placement randomness, or inherent unpredictability of race conditions. Moreover, end-to-end exploits are typically executed repeatedly until they succeed without triggering a kernel panic. As long as eventual success is possible, their practical reliability does not strongly depend on how many intermediate attempts fail. For these reasons, end-to-end exploit outcomes do not provide a meaningful or precise measure of the reliability of DIRTYFREE itself.

TABLE IV: Applicability of arbitrary free objects across Linux kernel versions from v5.0 to v6.16.

Struct Name	Effective Version
landlock_hierarchy	v5.13 ∼ v6.16
landlock_ruleset	v5.13 ∼ v6.16
async_poll	v5.7 ∼ v6.16
perf_event_pmu_context	v6.2 ∼ v6.16
pipe_inode_info	v5.0 ∼ v6.16
mnt_idmap	v6.2 ∼ v6.16
msg_queue	v5.0 ∼ v6.16
io_ring_ctx	v5.1 ∼ v6.16
msg_msg	v5.0 ∼ v6.10
msg_msgseg	v5.0 ∼ v6.16
sem_array	v5.0 ∼ v6.16
poll_list	v5.0 ∼ v6.16
callchain_cpus_entries	v5.0 ∼ v6.16
simple_xattr	v5.0 ∼ v6.16

```c
struct arbifree { char *ptr; };
struct arbifree *syn;

SYSCALL_DEFINE2(dirtyfree_test, int, cmd, int, id)
{
  if(cmd == 0)
  {
    syn = kzalloc(sizeof(struct arbifree), GFP_KERNEL);
    syn->ptr = kzalloc(128, GFP_KERNEL);
  }
  else if(cmd == 1)
  {
    syn->ptr = (char *)((unsigned long)syn->ptr & ~0xFFFFUL);
  }
  else if(cmd == 2)
  {
    struct slab *slab = virt_to_slab(syn->ptr);
    if(slab && strstr(slab->slab_cache->name, "cred_jar"))
    {
      struct cred *tmp = (struct cred *)syn->ptr;
      if(__kuid_val(tmp->uid) == id)
        return 1;
    }
  }
  return 0;
}
```

Fig. 6: A synthetic arbitrary free object code.