Breaking Isolation: A New Perspective on Hypervisor Exploitation via Cross-Domain Attacks

Gaoning Pan*†, Yiming Tao‡, Qinying Wang§‡, Chunming Wu‡, Mingde Hu*†, Yizhi Ren*†⋆, Shouling Ji‡
*Hangzhou Dianzi University, ‡Zhejiang University, §EPFL
†Zhejiang Provincial Key Laboratory of Sensitive Data Security and Confidentiality Governance
Email: {pgn, 20227001, renyz}@hdu.edu.cn, {taoym, wuchunming, sji}@zju.edu.cn, qinying.wang@epfl.ch

***Abstract*—Hypervisors are under threat by critical memory safety vulnerabilities, with pointer corruption being one of the most prevalent and severe forms. Existing exploitation frameworks depend on identifying highly-constrained structures in the host machine and accurately determining their runtime addresses, which is ineffective in hypervisor environments where such structures are rare and further obfuscated by Address Space Layout Randomization (ASLR). We instead observe that modern virtualization environments exhibit *weak memory isolation* — guest memory is fully attacker-controlled yet accessible from the host, providing a reliable primitive for exploitation. Based on this observation, we present the first systematic characterization and taxonomy of Cross-Domain Attacks (CDA), a class of exploitation techniques that enable capability escalation through guest memory reuse. To automate this process, we develop a system that identifies cross-domain gadgets, matches them with corrupted pointers, synthesizes triggering inputs, and assembles complete exploit chains. Our evaluation on 15 real-world vulnerabilities across QEMU and VirtualBox shows that CDA is widely applicable and effective.**

I. Introduction

Virtualization technology has been widely adopted in data centers, cloud computing, testing environments, and other scenarios [1]. These use cases place increasing demands on virtualization security, making it a critical focus for vendors and developers. As the cornerstone of virtualization, the hypervisor plays a pivotal role in managing and isolating virtual machines (VMs) within cloud environments, which makes it a high-value target for potential attackers. By exploiting vulnerabilities in the hypervisor, attackers can escalate privileges and gain control over the host system from a guest VM. This enables them to perform a range of malicious activities, such as data theft, system manipulation, or even a complete takeover of the cloud infrastructure—a scenario commonly referred to as *virtual machine escape*. This form of attack has attracted growing attention from the security community, and numerous vulnerabilities and exploits have been uncovered in recent years [2], [3], [4], [5], [6], [7], [8], [9], [10].

Among these vulnerabilities, *pointer corruption* stands out as a particularly dangerous and common consequence, typically caused by issues like use-after-free (UAF) or out-of-bounds (OOB) memory access. These vulnerabilities can corrupt address values in memory, allowing attackers to hijack pointers and redirect them to attacker-controlled locations. Such hijacked pointers may then be dereferenced to perform arbitrary memory access, enabling the attacker to read from or write to arbitrary locations in the hypervisor address space, potentially affecting sensitive data or control flow. Our investigation shows that pointer corruption is a frequent consequence of hypervisor vulnerabilities. An analysis of QEMU CVEs from the past five years indicates that about 23.9% ultimately result in pointer corruption (see Table I).

Despite their severity, exploiting pointer corruptions in hypervisors remains challenging due to indirection and complexity. Unlike in kernel or user-mode environments, exploitable memory structures in hypervisors are difficult to identify due to their scarcity and high degree of customization. To make matters worse, their address layout is further obscured by memory isolation mechanisms and address space layout randomization (ASLR), often requiring stringent conditions, such as additional vulnerabilities that leak address information, which are difficult to obtain. This renders existing exploitation frameworks, originally designed for kernel and user-mode environments [11], [12], limited, as they typically rely on locating those exploitable structures. As a result, existing exploits targets hypervisor tend to be case-specific and hand-crafted [13], [6], [7], [5], requiring deep manual analysis to identify suitable victim structures that expose read/write capabilities. To date, many of these vulnerabilities remain underexploited or entirely unexplored in practice, which limits our collective understanding and may cause security-critical bugs to be overlooked. This raises an alarming question: *how to design a systematic framework for exploiting pointer corruption in hypervisor that generalizes across both diverse vulnerability types and hypervisor platforms?*

In this work, we conduct the first systematic study of exploiting hypervisor pointer corruptions through cross-domain memory interactions. Instead of identifying attack-controllable structure in host machines, we introduce guest memory as a

⋆ Corresponding Author: Yizhi Ren (renyz@hdu.edu.cn)

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.[23|24]xxxx
www.ndss-symposium.org

reusable and attacker-controlled primitive. Our analysis reveals that the asymmetric isolation between host and guest in most modern virtualization systems can be leveraged to escalate exploitation. Specifically, while guest access to host memory is restricted, the host retains the ability to freely dereference pointers into guest memory. By redirecting corrupted pointers to crafted payloads in guest memory, attackers can reliably perform operations such as arbitrary memory writes, buffer manipulation, and fake object injection. We refer to this general exploitation paradigm applicable to most hypervisors as Cross-Domain Attacks (CDA). CDA transforms the challenge of finding exploitable host structures into a more tractable problem—leveraging guest-controlled memory as a stable, visible, and influenceable foothold for exploitation. Despite its power and generality, this attack surface has remained largely overlooked and lacks systematic exploration in prior work.

TABLE I: Classification of QEMU CVEs (2019–2024) by root cause and corruption capability.

Vulnerability Category	Ptr. Corr.	Data-only Corr.	No Corr.	Total
Use-After-Free	12	1	2	15
OOB Write	18	12	0	30
OOB Read	0	0	14	14
Integer Overflow	1	5	1	7
Uninitialized Variable	1	0	0	1
Information Leak	0	0	13	13
Logic/Crash and Others	0	0	54	54
Subtotal	**32**	**18**	**84**	**134**
Percentage	**23.9%**	**13.4%**	**62.7%**	**100.0%**

Ptr. Corr. = vulnerabilities that may alter pointer values. Data-only Corr. = vulnerabilities affecting data but not pointers. No Corr. = vulnerabilities with no corruptive effect on memory.

To bridge the gap in understanding CDA in real-world, we develop a framework to automatically identify CDA-capable code paths and synthesize end-to-end exploits for different types of vulnerabilites that cause pointer corruptions. Starting from a PoC that triggers a corrupted pointer, our CDA framework locates hypervisor code segments that dereference guest-sourced pointers and identifies candidate gadgets for exploitation. It then applies a trace-guided input synthesis strategy to generate gadget-triggering inputs, and incrementally assembles complete exploits by aligning memory layouts and scheduling system interactions. Our system achieves a high degree of automation and demonstrates broad applicability across different hypervisor platforms and vulnerability types.

To evaluate the effectiveness and generality of CDA, we conducted comprehensive experiments. We first analyzed the distribution of cross-domain gadgets across multiple hypervisors and found that nearly all major attack surfaces expose code paths containing such gadgets, suggesting CDA's wide applicability. We then measured the success rate of placing cross-domain gadgets in memory under different conditions. The results show that stack layouts exhibit substantial gadget coverage around common entry paths, while heap allocations provide even broader and more uniform opportunities for placing guest-derived pointers—making CDA-compatible layouts

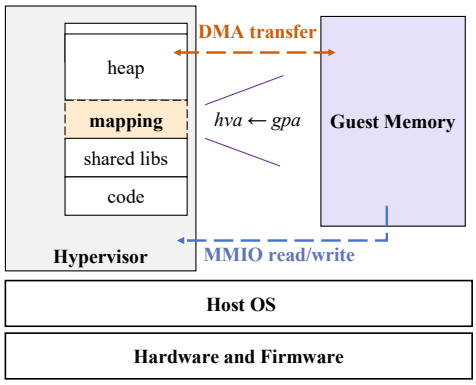


Fig. 1: Architecture of Type-2 Hypervisors.

consistently achievable in practice. Additionally, we applied our technique to 15 real-world vulnerabilities—13 in QEMU and 2 in VirtualBox, and successfully achieved exploitation in all cases. These results confirm that CDA is broadly applicable and effective for exploiting hypervisor pointer corruption vulnerabilities. Finally, the automated components of CDA introduce only acceptable and predictable overhead.

The main contributions of this work are as follows:

- A systematic characterization of cross-domain attacks (CDA), including their root causes and four pointer-use variants, establishing CDA as a general paradigm for upgrading pointer-corruption capabilities.
- A prototype system is developed to automate the proposed exploitation approach. The experimental data are now available at an repository https://xxx. The source code will be released upon publication.
- A comprehensive evaluation is conducted to demonstrate the coverage of cross-domain gadget identification within the hypervisor, the generality across various pointer corruption scenarios, and the effectiveness in exploiting real-world hypervisor vulnerabilities.

II. Background

A. Workflow of Hypervisor Exploitation

In virtualized environments, attackers typically exploit vulnerabilities in the hypervisor from a guest system in order to gain control over the host system, a process commonly referred to as *virtual machine escape*. In this work, we focus on the typical Type-2 virtualization in the host, where the hypervisor runs as a user-space process and is thus subject to conventional user-space memory protection mechanisms, such as Address Space Layout Randomization (ASLR) and Non-Executable (NX). To exploit the hypervisor, attackers must construct a reliable exploitation chain that begins with a memory corruption vulnerability. This chain typically involves gradually upgrading capabilities, bypassing multiple layers of protection, and ultimately achieving arbitrary code execution within the hypervisor context.

B. Guest Memory Management

Virtualization technology allows multiple virtual machines to share the same physical memory resources. The memory perceived by each guest operating system is actually an abstraction provided by the hypervisor through memory virtualization. As illustrated in Figure 1, the hypervisor typically allocates a contiguous region of memory in the host user space, referred to as the *Host Virtual Address (HVA)* space, via `mmap`. This region is then exposed to the guest as its *Guest Physical Address (GPA)* space. The *GPA-to-HVA* mapping is registered with the host kernel via system calls, enabling the kernel to construct hardware-assisted page tables to support efficient memory address translation and access. From the host's perspective, in addition to conventional user-mode memory segments, the hypervisor process includes a distinct memory-mapped region known as *Guest Memory*, which is used to support the execution of the guest operating system and its applications. Through virtualization, the hypervisor enforces strict boundary isolation on this memory region: the guest can only access and manage its own guest memory, and is prohibited from accessing memory outside of its assigned space. This isolation ensures both memory safety and logical separation between the guest and the host. When communication between the guest and host is required, it is typically facilitated through Memory-Mapped I/O (MMIO) and Direct Memory Access (DMA). MMIO is used for issuing control commands, while DMA is employed for high-throughput data transfer.

C. Exploiting Pointer Corruption Vulnerabilities

The basic concept of exploiting a pointer corruption vulnerability is to redirect a corrupted pointer to a memory region that enables the attacker to escalate their capabilities. Typically, the attacker targets a security-critical object, such as one that contains control information or a function pointer, and prepares the memory layout so that this object becomes accessible through the corrupted pointer. The pointer itself may be hijacked through direct overwrites, indirect memory manipulations, or side effects of other vulnerabilities. Once the corrupted pointer is dereferenced during subsequent execution, the attacker gains the ability to access or influence unintended memory regions, which can further lead to arbitrary reads or writes, unauthorized access, or control-flow manipulation.

D. Scope and Assumption

This work focuses on the automatic exploitation of pointer corruption vulnerabilities in type-2 hypervisors. We assume that the adversary has already obtained a pointer corruption primitive that allows modifying a host-side pointer via a guest-triggered vulnerability. The attacker is also assumed to possess full privileges within the guest VM, aligning with real-world cloud environments where attackers generally control the entire virtual machine. Additionally, we assume that standard user-mode protection mechanisms, such as address space layout randomization (ASLR) are enabled by default. The automatic capability upgrade technique proposed in this work does not violate these protections. Notably, type-2 hypervisor

```
1  // guest-controlled offset
2  void usbredir_buffered_bulk_packet(...,
       ↪ uint8_t *data, size_t data_len, ...) {
3      size_t i = choose_offset(...);
4      ...
5      bufp_alloc(dev, data + i, len, status, ep,
           ↪ data);  // interior ptr
6  }
7
8  int bufp_alloc(USBRedirDevice *dev, uint8_t *
       ↪ data, uint16_t len, ...) {
9      ...
10     if (bufpq_should_drop(dev, ep)) {
11         free(data);  // free(data + i): not
               ↪ chunk base
12         return -1;
13     }
14     ...
15 }
```

Fig. 2: Simplified illustration of the mistaken-free vulnerability in QEMU (CVE-2021-3682). An interior pointer `data+i` is passed downward and later freed, corrupting heap metadata.

setting is widely used in multi-tenant cloud infrastructures (e.g., Alibaba Cloud [14]) and aligns with prior hypervisor fuzzing research [9], [15]. Importantly, we do not assume that the attacker can directly execute arbitrary code on the host or invoke host-side functions at will. All exploitation steps must occur strictly along the natural logic of the hypervisor's existing code paths, as triggered by legitimate guest–host interactions.

III. Motivation and Our Exploitation

In this section, we use a motivating example (§III-A) to motivate our exploitation §III-B) and clarify our assumptions (§II-D).

A. A Running Example

Figure 2 presents a simplified example of the mistaken-free vulnerability (CVE-2021-3682) in QEMU. In this case, an interior pointer `data + i`, derived from a guest-controlled offset, is passed into the function `bufp_alloc`. This value is received as the parameter `data`, which is later freed within the callee. As a result, the memory allocator incorrectly invokes `free()` on a non-base pointer, corrupting internal heap metadata. This behavior constitutes a classic case of *pointer corruption*, and may be leveraged to manipulate subsequent allocations depending on allocator behavior and attacker-controlled buffer contents. At the time of writing, there is no public exploit available for this vulnerability.

Intuitively, to exploit this vulnerability, the attacker corrupts the internal metadata of a heap chunk, redirecting future allocations to a chosen address and causing them to overlap with critical in-memory structures. However, exploiting such pointer corruption in hypervisors is highly challenging. First, exploitable structures are rare and subject to strict constraints on chunk size and pointer offset, making reliable alignment

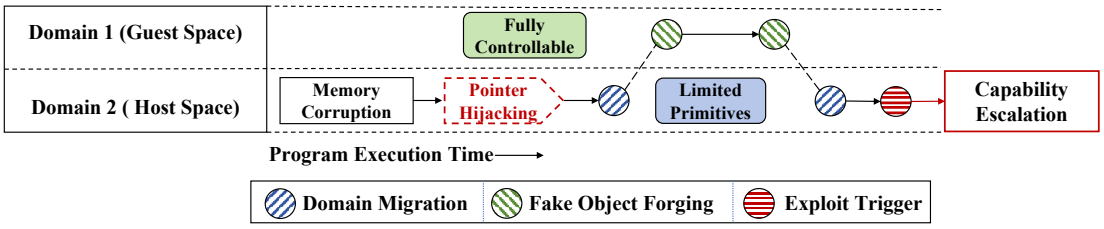


Fig. 3: CDA approach for exploiting pointer hijacking primitives.

difficult. Second, due to address space layout randomization (ASLR), the attacker lacks visibility into the runtime addresses of potential targets. Gaining precise control typically requires an additional information leak, significantly raising the bar for successful exploitation.

Limitations of Existing Techniques. The most related efforts on locating exploitable in-memory structures are found in kernel exploitation, where attackers have devised various strategies to identify and abuse kernel objects [11], [12]. However, such approaches are less applicable to hypervisors. The set of controllable in-memory structures is significantly smaller, and successful pointer hijacking often demands precise field alignment—constraints that vary across vulnerabilities and cannot be satisfied by generic layouts. These factors hinder the generalizability of structure-oriented search techniques in hypervisor contexts. To date, no systematic or automated exploitation methodology has been established for hypervisor environments.

Overlooked Isolation Flaws. However, existing exploitation techniques commonly assume strong guest-host memory isolation in the hypervisor, enforcing a strict separation between guest and host memory usage. In this model, attackers are restricted to user mode-like capabilities, significantly limiting the exploitation potential. Yet this assumption does not always hold in practice. Scavenger [16] had already demonstrated that once a corrupted host pointer was redirected into guest memory, the hypervisor could access guest-controlled content without restriction. This broke isolation boundaries and allowed guest memory to participate in host-side heap operations, enabling far stronger primitives than previously considered.

New Insight: Modern hypervisors exhibit a fundamental asymmetry: although a guest cannot directly read or write host memory, the host routinely accesses guest memory with few restrictions. This *weak isolation* exposes a contiguous, attacker-controlled region directly within the host address space. From the attacker's perspective, every byte of guest memory is controllable, making it a stable and highly manipulable foothold for exploitation. Redirecting a corrupted host pointer into this region circumvents the scarcity of suitably aligned host structures and sidesteps heap-layout uncertainty, opening a new avenue for precise and reliable hypervisor exploits.

```
// Guest (Attacker)
fake = guest_alloc_page();
    ↪ (1) allocate fake object
mmio_write(MMIO_ADDR, map_gpa(fake));
    ↪ (2) send fake object's GPA
...
fake->ops = final_attacker_ops;
    ↪ (5) modify fake object to
        finalize the exploit

// Host (Hypervisor)
s->ptr = (Req *)gpa_to_hva(gpa);
    ↪ (3) vulnerability overwrites
        pointer to fake object
qemu_free(s->ptr);
    ↪ (4) host uses attacker object
```

Fig. 4: Code example to show the workflow of CDA.

B. Our Exploitation

Driven by the new insights, we present a *systematic characterization* of how pointer hijacking primitives in hypervisors can be exploited through Cross-Domain Attacks (CDA) — a class of techniques that enable capability escalation without relying on exploitable structures, crafted I/O primitives, or additional information leaks. It provides a more general and reusable exploitation strategy, significantly reducing the complexity of identifying usable structures and constructing fragile, highly specialized payloads, thus ensuring the success of complete exploitation.

Our approach involves two core steps: *cross-domain migration* and *fake object forging*, as illustrated in Figure 3. In the first step, a pointer hijacking primitive is repurposed to redirect the host-side pointer to the guest memory space, enabling cross-boundary injection of attacker-controlled data. In the second step, this redirected pointer is used to access a carefully crafted fake object placed in guest memory, resulting in capability escalation.

Step 1: Cross-Domain Migration. It is achieved by placing a special type of code snippet that leaves a host virtual address (HVA) pointing to guest memory in the host address space.

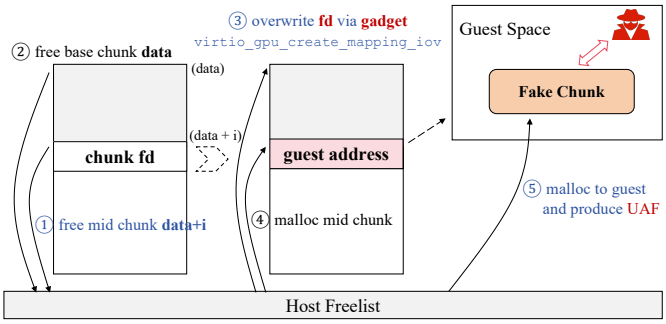


Fig. 5: Exploiting the running example.

These snippets are part of the communication mechanism between the guest and the host, responsible for establishing the mapping between a GPA and a corresponding HVA, and for facilitating data transfer. We refer to such code snippets as *cross-domain gadgets*. Through careful memory layout manipulation during the window between vulnerability triggering and pointer dereference, the residual guest-HVA pointer[1] left by these gadgets can be reused to redirect the corrupted pointer into the guest memory space.

Step 2: Fake Object Forging. It is achieved by carefully placing a forged object within the guest memory, at the location pointed to by the hijacked pointer. This fake object mimics the layout and semantics expected by the hypervisor's internal logic—such as containing function pointers, capability flags, or structured data fields. When the corrupted pointer is later dereferenced by the host, the hypervisor transparently accesses this attacker-controlled payload, unknowingly operating on manipulated data. This enables the attacker to upgrade the exploit capability without modifying any host-side data or bypassing memory protections directly.

Code Sample of CDA Workflow. As illustrated in the Figure 4, the guest first allocates a page to store a forged object and issues an MMIO write to trigger the vulnerable host code path (Steps 1–2). During this host-side processing, a memory corruption bug (e.g., overflow or UAF) allows the attacker to overwrite an internal hypervisor pointer with an arbitrary value. By supplying the host virtual address (HVA) corresponding to the forged guest page, the attacker forces the hypervisor to treat attacker-controlled guest memory as a legitimate host object (Step 3). When the hypervisor later performs an operation such as *free()* or a structure dereference on this corrupted pointer (Step 4), it unknowingly operates on the forged object. The guest can then modify the same memory page at any time, enabling it to install the final payload that will be executed upon the hypervisor's subsequent dereference (Step 5).

Exploiting Running Example. To exploit the vulnerability in the running example, we use the `virtio_gpu_create_mapping_iov` function as a cross-domain gadget, as illustrated in Figure 5. This gadget is capable of producing a guest-HVA pointer within a heap-allocated object, and its chunk size is compatible with the vulnerable structure.

When the vulnerability is triggered (① in Figure 5), the attacker first frees a subregion within a heap buffer, specifically, the pointer at `data + i`, and then subsequently frees the original buffer `data`. This sequence creates overlapping chunks that simultaneously reside in the hypervisor's freelist. Following this (③ in Figure 5), the attacker invokes the selected cross-domain gadget to allocate a new chunk and uses heap spraying to place it at the reclaimed data location. At this point, the metadata of the freed chunk at `data + i` is modified to overwrite its `fd` pointer with a guest address. This manipulation causes a fake chunk located in guest memory to be inserted into the host freelist. As a consequence (⑤ in Figure 5), the next heap allocation made by the hypervisor retrieves this fake chunk, effectively returning a pointer into guest-controlled memory. Since the attacker has full read/write control over guest memory, this transforms the vulnerability into a standard Use-After-Free primitive. The attacker can then reliably craft payloads within guest memory to corrupt critical host data structures or redirect control flow, thereby achieving a full virtual machine escape.

CDA Variants. To further demonstrate the expressiveness and versatility of the Cross-Domain Attack (CDA) model, we identify and categorize four representative variants, each characterized by the dereference semantics of the hijacked pointer and the resulting impact. These variants include arbitrary Code Execution (**CDA$^{\mathbf{A}}$**), where the attacker gains control over the program counter or function pointer to execute arbitrary code; Information Leakage (**CDA$^{\mathbf{I}}$**), which enables reading sensitive data by redirecting pointers to attacker-observable memory; Critical Data Overwriting (**CDA$^{\mathbf{O}}$**), which targets security-critical fields such as credentials or control flags; and Chunk Confusion (**CDA$^{\mathbf{C}}$**), which manipulates allocator metadata to corrupt heap integrity. These variants illustrate the broad range of exploitation consequences achievable through CDA under different vulnerability scenarios. A formal taxonomy and detailed case studies for each variant are provided in Appendix B.

Technical Challenges. To perform the exploitation described above, an adversary must address several key challenges. First, identifying valid gadget candidates is difficult due to their implicit and inconsistent patterns across the hypervisor codebase. Without explicit semantics or uniform signatures, systematically locating these translation points is highly non-trivial. Second, establishing a viable match between a corrupted pointer and a gadget requires understanding the memory characteristics of the gadget-produced pointer. Lacking such knowledge makes the matching process substantially more ambiguous and error-prone. Third, triggering deep gadget paths requires inputs that satisfy complex and hidden control-flow conditions. The vast input space and lack of intermediate feedback hinder direct exploration. Finally, integrating all components into a complete exploit demands precise coordination of control flow, memory layout, and time

[1]A guest–HVA pointer is a host-side pointer whose value is a host virtual address (HVA) that corresponds to a guest memory region.

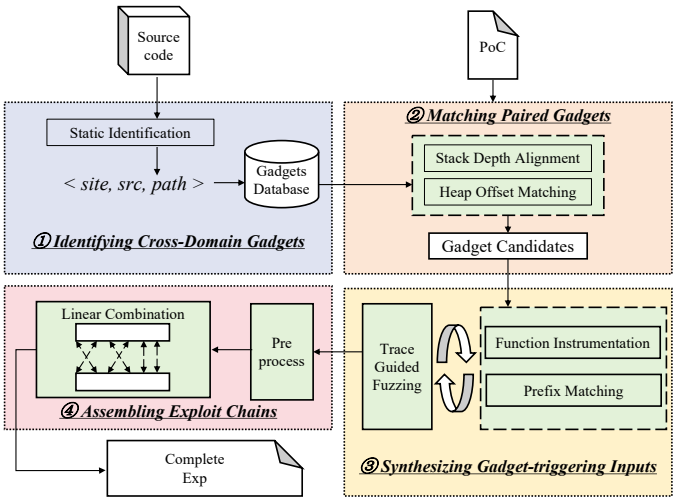


Fig. 6: The overall workflow of CDA framework.

window.

IV. Methodology

Figure 6 outlines the workflow of the CDA framework, which is divided into four sequential stages. At a high level, the framework takes as input (i) the hypervisor's source code and (ii) a proof-of-concept (PoC) that triggers a pointer corruption, and produces as output an exploit that redirects the corrupted pointer into the guest memory—achieving one of the four CDA variants. Starting from a pointer corruption vulnerability, we first statically identify cross-domain gadget instances within the hypervisor and organize them into a structured database (§IV-A). Given a PoC, we then match it with suitable gadget candidates based on memory layout compatibility—differentiating between heap- and stack-based corruption models (§IV-B). Next, we synthesize concrete guest inputs that reliably trigger the selected gadgets, using a trace-guided fuzzing strategy to incrementally drive execution through the intended call chains (§IV-C). Finally, a complete exploit is assembled by integrating the original PoC and the gadget-triggering input, forming a coherent execution sequence that redirects the corrupted pointer to attacker-controlled guest memory and completes the capability escalation (§IV-D). Subsequent operations after the pointer redirection are beyond the scope of this paper, as they follow standard exploitation procedures.

A. Identifying Cross-Domain Gadgets

Identifying cross-domain gadgets involves statically analyzing the hypervisor codebase to extract all code sites that perform GPA-to-HVA translation under guest influence. These translation functions map guest physical addresses to host virtual addresses, during which guest-HVA pointers are temporarily or globally stored in host memory. Empirically, we find that these sites represent the primary contexts where guest-derived addresses are likely to reside in host memory, creating opportunities for pointer redirection. In addition, we further identify their guest-accessible invocation paths. Cross-domain gadgets are thus necessary to bridge corrupted host pointers to attacker-controlled guest memory when ASLR prevents direct address targeting, capturing code-reuse scenarios where existing instructions leave guest addresses accessible. The full list of hypervisor functions responsible for address translation is provided in Appendix A.

Definition: Cross-Domain Gadgets. $\mathcal{G} = \{(site, src, path) \mid site \in \mathcal{S}, src \in \mathcal{T}, path \in \mathcal{P}\}$ denotes the set of all cross-domain gadgets. Each element $g \in \mathcal{G}$ is a 3-tuple that uniquely identifies a gadget instance within the hypervisor. Here $site$ refers to a program location where a GPA-to-HVA translation is performed, src denotes a guest-controllable operand that flows into the translation logic, and $path$ represents the static call chain from a guest-facing interface to the translation site.

To identify such gadgets in practice, we conduct a static analysis over the hypervisor codebase. We begin by locating all translation sites that perform GPA-to-HVA mappings, including standard functions and subsystem-specific logic. For each site, we trace the origin of the GPA operand to determine whether it can be influenced by the guest, either through descriptor structures, MMIO registers, or I/O buffers. Sites with no viable guest-controlled sources are discarded.

Focusing exclusively on *guest-influenced translations* enables precise control over the resulting host virtual address. This controllability eliminates the need for imprecise memory spraying or blind guessing across the entire guest memory space, and allows the attacker to steer the corrupted pointer to a specific location in guest memory. As a result, this constraint significantly improves the reliability and precision of subsequent exploit stages.

For each remaining candidate, we extract its associated call chain using static source-to-sink analysis, which identifies guest-reachable paths that lead from external interfaces to the translation site. This path provides crucial context for evaluating gadget reachability and constructing triggering inputs in later stages. Each verified ($site$, src, $path$) triple is then inserted into a structured gadget database, forming the set $\mathcal{G}$.

Example. Table II presents a representative example of a cross-domain gadget extracted from QEMU's NVMe device. The gadget is triggered via an MMIO write, where a guest-controlled field of type `dma_addr_t` is translated into a HVA and stored in a pointer variable named `ram_ptr`. This pointer is later used as an access target and can be abused as a redirection target for a corrupted pointer during exploitation. Since `ram_ptr` holds a guest-derived address, an attacker who hijacks a corrupted pointer to target `ram_ptr` can effectively manipulate host logic to operate on attacker-controlled guest memory. This makes the gadget suitable in CDA-style exploitation. A full list of cross-domain gadgets are shown in Table VI in the appendix.

B. Matching Paired Gadgets

Building upon the cross-domain gadget database, we now describe how to match a given pointer hijacking vulnerability with a suitable gadget instance to enable exploitation. The

TABLE II: A cross-domain gadget extracted from QEMU's NVMe device.

Gadget Family	Upper Function	Translation Function	HVA Variable	GPA Source Field	Trigger Type	Call Path
DMA gadget	dma_memory_write	address_space_write	ram_ptr	s→tx_descriptor	MMIO	`nvme_mmio_write` → `nvme_process_db` → `stl_le_pci_dma` → `stl_le_dma` → `dma_memory_write`

Algorithm 1: Paired Gadget Matching Strategy

Input: Gadget database $\mathcal{G}$, corrupted pointer metadata M, pointer region $R \in \{\text{stack}, \text{heap}\}$
Output: Set of matched gadgets $\mathcal{G}_{\text{match}}$

1 $\mathcal{G}_{\text{match}} \leftarrow \emptyset$;
2 **if** $R = \text{stack}$ **then**
3 **foreach** $g = (site, src, path) \in \mathcal{G}$ **do**
4 $g.depth \leftarrow \text{Length}(path)$;
5 $p_{\text{depth}} \leftarrow M.\text{stack\_depth}$;
6 **foreach** $g \in \mathcal{G}$ **do**
7 **if** $g.depth = p_{\text{depth}}$ **then**
8 $\mathcal{G}_{\text{match}} \leftarrow \mathcal{G}_{\text{match}} \cup \{g\}$;
9 **else if** $R = \text{heap}$ **then**
10 **foreach** $g \in \mathcal{G}$ **do**
11 **if** $\text{IsStoredAsStructField}(g)$ **then**
12 $(g.size, g.offset) \leftarrow \text{ExtractStructInfo}(g)$;
13 $(s_{\text{size}}, o_{\text{offset}}) \leftarrow M.\text{heap\_layout}$;
14 **foreach** $g \in \mathcal{G}$ **do**
15 **if** $g.size = s_{\text{size}}$ **and** $g.offset = o_{\text{offset}}$ **then**
16 $\mathcal{G}_{\text{match}} \leftarrow \mathcal{G}_{\text{match}} \cup \{g\}$;
17 **return** $\mathcal{G}_{\text{match}}$

database contains numerous program paths that generate guest-address HVAs at different memory offsets, and the key insight is that our framework aims to *pair a vulnerability with the gadget whose residual guest-HVA pointer is spatially aligned with the corrupted pointer*. This spatial alignment allows the gadget's leftover HVA to overwrite the corrupted pointer exactly, thereby transforming an otherwise unexploitable corruption into a valid cross-domain redirection primitive.

Specifically, we focus on identifying a *paired gadget* whose pointer write behavior can be aligned with the corrupted pointer produced by a proof-of-concept (PoC) input. Based on an empirical study of prior hypervisor vulnerabilities, we observe that corrupted pointers typically reside in either the stack or the heap. These two regions exhibit fundamentally different allocation semantics and memory layouts, which in turn necessitate distinct pairing strategies. For stack-resident pointers, layout compatibility depends on relative stack depth. For heap-resident pointers, it depends on object structure and field offset. We therefore design two complementary gadget matching procedures, one for each case, formalized in Algorithm 1.

Stack-Based Matching. This strategy formalizes the intuition that if a gadget deposits a guest-HVA pointer at a specific stack depth, and the corrupted pointer resides at the same depth during the crash, then the two can be treated as aligned. That is, we assume that matching the relative call depth is sufficient to ensure that the memory locations coincide, thereby enabling layout reuse. For each gadget in the database, we first annotate its relative stack depth based on the length of its static call chain (lines 2–4). Given a crashing PoC, we extract the depth of the corrupted pointer within the call stack (line 5), and then select gadgets with matching stack depth (lines 6–8). These are returned as candidate paired gadgets that can be deployed via controlled call sequences.

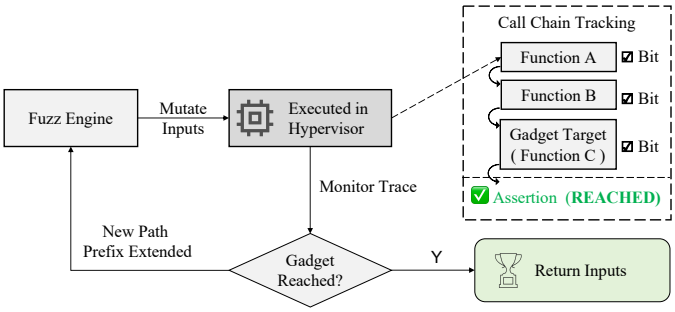


Fig. 7: Trace-guided fuzzing workflow.

Heap-Based Matching. In contrast to the stack, heap memory is governed by allocator behavior and object layout. Heap-based corruptions often target structure fields embedded within dynamically allocated chunks. Our insight is that if a gadget writes a guest-HVA pointer into a structure field with a known size and offset, and if a corrupted pointer resides at the same layout position, then the attacker can feasibly align the two via object spraying. We first identify gadgets that store guest-HVA pointers as part of structure fields by performing backward dataflow analysis (lines 9–11), annotating each with its structure size and offset. Then, from the PoC, we extract the corrupted object's size and the field's offset (line 12). Layout-aligned gadgets are selected as viable heap-matching candidates (lines 13–16).

C. Synthesizing Gadget-triggering Inputs

Given a set of matched gadget instances produced, the next step is to synthesize concrete guest inputs that can reliably trigger the execution of these gadgets. This is essential for enabling end-to-end exploit construction, but presents a practical challenge: gadgets typically reside deep within the hypervisor's call chains, and the conditions under which they are executed may be highly constrained. Applying naive coverage-guided fuzzing is insufficient, as the search space is

Algorithm 2: Trace-Guided Input Synthesis for Gadget Triggering

Input: Gadget G with call chain $[f_1, f_2, \ldots, f_n]$
Output: Input x that triggers G via the intended call chain

1 Instrument each function $f_i \in [f_1 \ldots f_n]$ to update `bitmap[i]` on execution;
2 Instrument f_n with a termination signal (e.g., assertion/crash);
3 Initialize `corpus` $\leftarrow \emptyset$;
4 **while** not timeout **do**
5 $x \leftarrow$ MUTATEINPUT();
6 RUNGUEST(x);
7 $P \leftarrow$ LONGESTVALIDPREFIX($[f_1 \ldots f_n]$, `bitmap`);
8 **if** P is longer than any previous prefix **then**
9 Add x to `corpus`;
10 **if** f_n reached via valid prefix **then**
11 **return** x;
12 **return** $\perp$; `// No trigger found within time bound`

enormous and lacks effective guidance. To address this, we design a *trace-guided input synthesis strategy* that incrementally steers fuzzing toward executing the target gadget, as shown in Algorithm 2 and visualized in Figure 7. The core idea is that our strategy focuses on guiding the fuzzing process by monitoring progress along the call chain leading to the target gadget. Instead of blindly generating random inputs, the strategy tracks the path through the call chain and selectively preserves inputs that extend the chain towards the intended target. This turns the problem into a guided search over partial call chains, which significantly reduces the search space and improves the likelihood of successfully triggering the gadget.

The synthesis strategy consists of three key steps. ① For each candidate gadget, we instrument every function along its call chain to set a corresponding bit in a reserved region of the global coverage bitmap upon execution (line 1). The final target function—the gadget's translation site—is instrumented with a termination signal `assert` to serve as a concrete indicator of successful triggering (line 2). ② During fuzzing, the system repeatedly generates and mutates guest inputs (line 5), then executes them within the virtualized environment (line 6), monitoring which chain elements have been traversed. After each execution, we extract the *longest valid prefix* of the chain that has been traversed in the correct order (line 7). Inputs that extend this prefix are marked as *interesting* and added to the fuzzing corpus (lines 8–9), promoting incremental exploration. This mechanism gradually drives the fuzzer toward deeper chain coverage. ③ To ensure that the observed function visitation reflects the intended chain, we enforce strict caller verification: each function is only marked as reached if invoked directly by its expected predecessor. This constraint is embedded into the LongestValidPrefix check (line 7) to eliminate spurious matches arising from unrelated execution paths. Once an input successfully reaches the gadget's target function via the full intended chain (line 11), the fuzzer terminates and returns the corresponding input (line 12). The result of this stage is a collection of synthesized inputs corresponding to gadgets that have been confirmed executable through their annotated control paths.

D. Assembling Exploit Chains

Given a PoC that triggers a pointer corruption vulnerability and a synthesized gadget-triggering input, the final step is to construct an end-to-end exploit that redirects control or data flows across the guest–host boundary. This process involves stitching together the vulnerability trigger with a validated cross-domain gadget to form a coherent exploit chain.

We begin by decomposing the PoC into three semantic stages: the *preparation phase*, which sets up memory layout and execution context; the *trigger phase*, which produces the corrupted pointer; and the *exploitation phase*, which dereferences the corrupted pointer. The preparation phase includes both payload placement and runtime stabilization. In heap-based scenarios, this entails spraying controlled structures into the guest heap to ensure consistent placement. In stack-based cases, we suppress asynchronous interference (e.g., signal delivery, multithread scheduling) by disabling preemption and isolating execution to a clean context, thereby ensuring a deterministic stack layout for the pointer corruption to manifest reliably.

To integrate the selected gadget, we insert the synthesized gadget-triggering input at a position in the PoC's execution flow determined by the vulnerability type. The input is placed either before or after the corruption trigger, depending on the vulnerability type (e.g., post-trigger for UAF, pre-trigger for stack corruption). Our system accommodates both patterns by supporting flexible interleaving of the gadget-triggering input with the PoC logic. This ensures that the corrupted pointer, once created, is redirected to a valid and attacker-controlled guest memory region by invoking a gadget instance identified in IV-A and matched in IV-B. This intermediate step preserves execution continuity while re-routing the pointer dereference target, effectively bridging the gap between vulnerability and capability elevation.

Following gadget invocation, the program proceeds to dereference the now-translated pointer. Depending on the semantics of the gadget and the nature of the dereference, this access may directly manipulate sensitive data structures or initiate privilege escalation. The overall exploit chain, which includes vulnerability activation, gadget activation, and capability upgradation, is illustrated in Figure 8, which shows the temporal and logical structure of this composition.

E. Extensibility and Portability

Our framework is designed with modularity and adaptability in mind, enabling it to generalize across different hypervisors and vulnerability types. To port the framework to a new

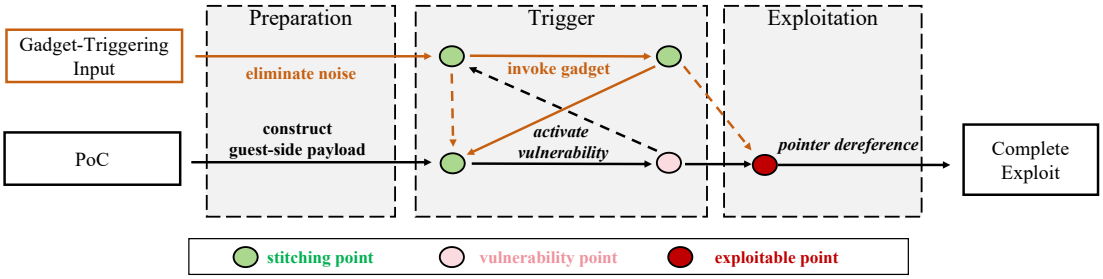


Fig. 8: Workflow for assembling an exploit chain.

hypervisor, developers only need to perform lightweight and localized adaptations while keeping the core pipeline for gadget identification, matching, and exploit synthesis unchanged. Specifically, our static analysis step requires extracting three categories of information from the hypervisor's developer documentation: (1) the **base translation functions** (e.g., `bhyve`'s `vm_map_memory`), (2) the **I/O entry points** (e.g., `e1000_write_reg`), and (3) the **types and definitions of guest-influenced variables** (e.g., `vm_paddr_t`).

V. Evaluation

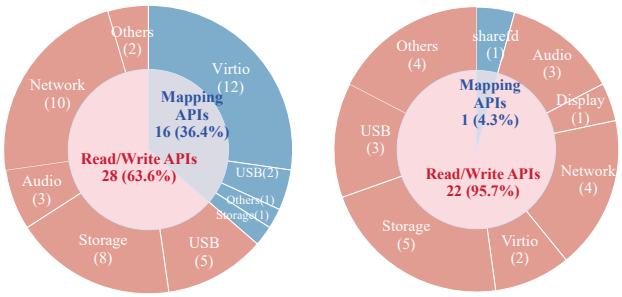
To demonstrate the practicality of CDA, we collected a series of metrics to establish its prevalence in virtualization environments. We focused on QEMU (used either standalone or together with KVM) and VirtualBox, which are representative hypervisor software. Our goal is to address the following research questions:

- **RQ1:** How prevalent are cross-domain gadgets, and what is their distribution within hypervisors?
- **RQ2:** How frequently do cross-domain gadgets produce guest–HVA pointers?
- **RQ3:** How does CDA perform in actual exploit scenarios?

For our static analysis, we utilized CodeQL, while function stack frames and variable offsets were extracted at the binary level using Ghidra. To validate the actual exploit scenarios, each experiment was conducted on a server equipped with an Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz (72 cores) and 64GB of RAM. Both the host and guest operating systems were 64-bit Ubuntu 20.04 LTS.

A. Gadget Prevalence

To understand how cross-domain gadgets originate inside real hypervisors, we begin with the static analysis stage of our CDA framework. First, we extract all GPA-to-HVA translation functions in both QEMU and VirtualBox. These are all APIs that accept guest-supplied GPAs and convert them into host-accessible addresses under guest influence. Using documentation-guided patterns and interprocedural tracing, our analysis identifies **44 translation functions in QEMU** and **23 in VirtualBox**. Given that virtual devices represent the primary attack surface and constitute the largest portion of the hypervisor's code [8], [17], [9], we use device type as the basic unit of analysis when categorizing these translation



(a) Distribution in QEMU. (b) Distribution in VirtualBox.

Fig. 9: Statistics on the number of virtual devices with GPA-to-HVA translation functions in the hypervisor.

```
1 int lsi_mem_read(LSIState *s, dma_addr_t addr,
      ↪ void* buf, dma_addr_t len)
2 {
3     if (s->mode & LSI_DMODE_SIOM) {
4         address_space_read(*s->pci_io_as,
              addr, buf, len);
5     } else {
6         pci_dma_read(s, addr, buf, len);
7     }
8 }
```

Fig. 10: A gadget in QEMU's `lsi_mem_read` function that conditionally invokes either `address_space_read` or `pci_dma_read` based on the DMA mode.

functions. Across both systems, these translation functions fall empirically into two major semantic categories: **read/write APIs** and **mapping APIs**.

As shown in Figure 9, **read/write APIs constitute the majority—63.6% in QEMU and 95.7% in VirtualBox**. These APIs appear broadly across device families such as storage, USB, network, and block devices. Their high frequency reflects their fundamental role in moving data across the guest–host boundary. Because these APIs are embedded in many device I/O paths, they create numerous translation points through which guest-controlled GPAs flow. For example, Figure 10 presents code from the `lsi` storage device in QEMU, where calls to `pci_dma_read` and `address_space_read` temporarily materialize guest addresses as local variables during execution—an inherent

consequence of C's calling conventions.

Mapping APIs account for the remaining 36.4% of translation functions in QEMU and 4.3% in VirtualBox. Although fewer in number, mapping APIs often generate structured heap-allocated objects, such as descriptor tables or DMA mapping entries, in which guest addresses are embedded. These objects persist beyond the scope of individual API calls and may be reused or accessed by multiple devices, giving mapping-derived translation points a broader and more durable influence. In QEMU, mapping APIs are primarily used in virtio and USB subsystems, consistent with the design of high-throughput and paravirtualized devices. VirtualBox, by contrast, uses mapping APIs almost exclusively in the HGCM shared-file subsystem, reflecting its more centralized architecture. While our analysis also observes mapping patterns in initialization-only structures (e.g., `deviceState`), these memory regions are not attacker-controlled and are therefore excluded from our statistics.

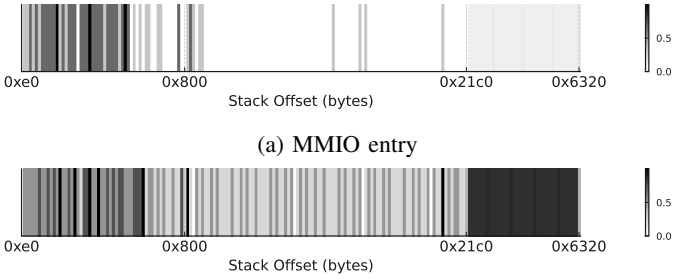
Second, using these translation sites as anchors, we enumerate all concrete cross-domain gadgets on QEMU. For each translation function, we trace the propagation of guest-influenced GPAs along its call chain and record the contexts where these translated values interact with host pointer operations. This process yields **772 gadget instances**, which naturally cluster into **eight major gadget families**, each representing a distinct translation semantic and operand-usage pattern. Table VI in the appendix summarizes these families, their representative APIs, and the underlying translation semantics.

Overall, this analysis shows that GPA-to-HVA translation logic is deeply embedded across hypervisor device-emulation code, and both read/write and mapping APIs serve as systematic and recurring sources of cross-domain gadgets. The large number and diversity of these gadgets highlight the structural ease with which guest-controlled addresses permeate hypervisor execution paths, underscoring the practicality of constructing CDA attacks across a wide range of devices and translation contexts.

B. Pointer Presence

Since our CDA exploitation pipeline relies on reusing guest-derived pointers left in the hypervisor's memory, the availability and distribution of such pointers fundamentally determine whether a corrupted host pointer can be redirected to guest memory. To assess the feasibility of CDA across real virtualization workloads, we perform a comprehensive measurement of guest-pointer coverage in QEMU's stack and heap.

1) Stack Analysis: To evaluate where guest-derived pointers naturally appear during hypervisor execution, we analyzed the stacks of QEMU's three major entry paths—`MMIO` (memory-mapped I/O handlers), `bh` (bottom-half handlers), and `timer` (timer callbacks)—and aggregate all guest-pointer residues beneath their entry frames. Because `MMIO` uses a dedicated VCPU-thread stack while `bh` and `timer` callbacks share the main-loop stack, we analyzed them separately. `MMIO` handlers



(a) MMIO entry

(b) Timer/BH entry.

Fig. 11: Distribution and coverage of guest-derived pointers in the QEMU stack.

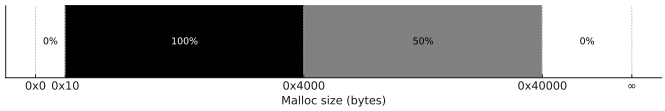


Fig. 12: Heap coverage of guest-derived pointers in QEMU.

are invoked on VM exits, whereas `bh/timer` callbacks execute asynchronously on the main-loop thread. The resulting heat-strip visualizations (Figure 11) reveal a consistent and structured distribution pattern.

Across all entry paths, **shallow stack frames (roughly 0–0x200)** show relatively sparse residual pointers for `MMIO` due to multiple initialization layers, whereas `bh/timer`—invoked directly from the event loop—exhibit noticeably denser activity in the same region. **Mid-stack frames (approximately 0x200–0x800)** form a clear high-density band for all entry types, corresponding to the core device-emulation and DMA read/write routines that frequently spill guest-HVA pointers. Beyond this point, the **0x800–0x21c0** interval appears as a lighter, more heterogeneous transition region. This segment contains fewer residues and reflects device-specific behaviors; nonetheless, residues still appear intermittently, providing occasional placement candidates for CDA. A markedly different pattern emerges in the **deep stack region (around 0x21c0–0x6320)**. As shown in the figure 11, this band extends even further under `bh/timer` execution, where virtio backends produce a long, continuous dark band driven by regularly structured DMA and DMA-mapping APIs, which leave guest addresses at both aligned and non-aligned offsets. This deep region is not only broader but also more uniform for `bh/timer` than `MMIO`, creating a large space where corrupted host pointers can reliably be redirected into guest memory.

Overall, these stacked dense bands—spanning shallow, mid, transition, and deep regions—show that QEMU repeatedly leaves guest-derived pointers across large portions of its execution stacks. Consequently, CDA remains feasible for pointer corruptions occurring at a wide variety of stack depths, regardless of the specific device or entry path involved.

2) Heap Analysis: We further examine where guest-derived pointers appear in QEMU's heap allocations. By grouping

allocations by size and visualizing pointer residues (Figure 12), we observe a clear and highly structured pattern of coverage across malloc size classes. Allocations in the **0x10–0x4000** range exhibit **consistently full coverage**, meaning guest-controlled pointers can be positioned reliably at these sizes. In the **0x4000–0x40000** range, coverage remains substantial—typically around half of all possible positions—due to regular spacing and layout properties of QEMU's allocation structures. Beyond these ranges, residues become scarce, and coverage drops to zero.

Our measurement is based directly on malloc size classes, reflecting how real heap manipulation works. In practice, objects of the same allocation size can be reliably steered into the same bins and co-located via standard heap fengshui techniques. This allows an attacker to position a corrupted heap object onto a chunk that already contains guest-pointer residues, making matching feasible whenever the allocation sizes align. Thus, evaluating coverage at the granularity of malloc size directly indicates which heap objects can practically host residue-bearing chunks during exploitation.

A key observation is that a wide variety of virtio backends naturally create **elastic guest-pointer placement opportunities** in these size classes: their internal structures and padding patterns allow the guest to control both the size and alignment of embedded addresses, resulting in a broad, contiguous spectrum of heap objects in which guest-HVA pointers can be positioned. Importantly, these patterns are not tied to a single data structure or device type; instead, they emerge repeatedly across multiple virtio implementations, making the effect systematic rather than device-specific. A representative example illustrating this elastic layout behavior is provided in Appendix C.

Since most heap objects involved in I/O processing fall within these size ranges, nearly all heap-based pointer corruptions in QEMU can be paired with suitable guest-pointer residues. Consistent with this observation, our automated search identifies over a hundred residuable heap locations within one hour, confirming that heap layouts provide extensive and repeatable opportunities for CDA redirection.

C. Exploit Practicality

To comprehensively evaluate the practicality of our approach, we examined all pointer-corruption vulnerabilities listed in Table I, and successfully exploited 15 vulnerabilities in QEMU and VirtualBox. Our framework successfully exercised all four CDA variants across a broad spectrum of vulnerability classes, including heap overflows, out-of-bounds accesses, UAF, double free, and uninitialized free, and across diverse device types such as slirp, USB, NVMe, and multiple virtio subsystems. Among the 15 evaluated vulnerabilities in QEMU and VirtualBox, CDA enabled reliable exploitation in all cases, including several vulnerabilities that had previously been considered highly challenging to exploit in practice, such as CVE-2021-3682, CVE-2020-2575, and Scavenger. These results demonstrate that guest memory can serve as a stable and reusable primitive when traditional host-centric

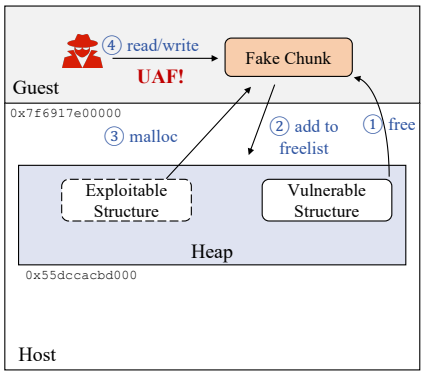


Fig. 13: The workflow of QEMU NVMe case's exploitation based on the CDA attack.

strategies fail. The CDA variant used for each vulnerability is summarized in Table III.

A closer examination of the variant distribution reveals clear patterns that align with the structural characteristics of different vulnerability types. CDA^I most frequently appears in out-of-bounds read vulnerabilities because redirecting a corrupted pointer to guest memory naturally produces attacker-controlled information leaks. CDA^O is widely applicable to overwrite-based bugs, where pointer-adjacent fields such as lengths, flags, or buffer descriptors can be reliably redirected to guest memory. CDA^C often arises in UAF-style vulnerabilities, including double free and uninitialized free, since the attacker can steer guest-mapped chunks into the host's freelist and effectively convert guest objects into host-allocated ones. CDA^A appears less frequently because control-sensitive function pointers are relatively sparse in hypervisor structures; however, when such pointers exist, CDA^A enables reliable control-flow hijacking, as demonstrated in CVE-2020-2575 and CVE-2023-3180. These observations indicate that CDA variants occur naturally based on the underlying bug structure, and CDA provides a unified exploitation strategy that adapts consistently across them.

Case Study: QEMU NVMe Uninitialized Free. The bug occurs when an `sglist` structure is freed without proper initialization, causing the hypervisor to free an attacker-influenced pointer. Under conventional exploitation models, the attacker would need to find a host-side structure that matches the size and layout of `sglist`, place it at an appropriate heap location, and identify device-specific primitives capable of manipulating its pointer fields. These conditions are difficult to satisfy in practice. CDA removes these obstacles. As illustrated in Figure 13 in the appendix, our framework identifies a virtio-based cross-domain gadget that leaves guest addresses in allocator-sized chunks, representing the exploitable structures. Through standard heap grooming, this gadget output is placed into the host's freelist, ensuring that the vulnerable `sglist` structure receives a guest-derived pointer when allocated. During the uninitialized free, this guest-controlled fake chunk is inserted into the host's freelist and becomes indistinguishable from a regular host allocation,

TABLE III: We successfully exploited 15 real-world vulnerabilities using CDA, with 13 of them being in QEMU and 2 in VirtualBox. Among them, we use **A** to represent executing weird machine (Fig 14a), **I** to represent information leakage (Fig 14b), **O** to represent overwriting critical data (Fig 14c), and **C** to represent chunk confusion (Fig 14d). These are the types of CDA variants used for these vulnerabilities.

	CVE id/Name	Device	Vulnerability Type	CDA Variants	Impact	Success
QEMU	CVE-2019-6778	slirp	Heap overflow	**O**,**C**	RCE	✓
	CVE-2019-14378	slirp	Heap overflow	**O**,**C**	RCE	✓
	CVE-2020-7039	slirp	Heap overflow	**O**,**C**	RCE	✓
	CVE-2020-14364	USB	OOB	**A**,**I**	RCE	✓
	CVE-2021-3682	USB redirector device	Mistake free	**O**,**C**	RCE	✓
	CVE-2021-3929	Nvme	UAF	**O**,**C**	RCE	✓
	CVE-2023-3180	virtio-crypto	Heap overflow	**A**,**I**	RCE	✓
	CVE-2023-6693	virtio-net	Stack overflow	**I**	Info leak	✓
	Scavenger	NVMe	Uninitialized free	**O**,**C**	RCE	✓
	Fixes: 1733eebb9e7	virtio-iommu	OOB read	**I**	Info leak	✓
	CVE-2024-3446	virtio-gpu	Double free	**O**,**C**	RCE	✓
	CVE-2024-8612	virtio-blk	OOB read	**I**	Info leak	✓
	Fixes: 62dbe54c	virtio-sound	Heap overflow	**A**	RCE	✓
VirtualBox	CVE-2020-2575	usb-ohci	Uninitialized heap	**A**	RCE	✓
	CVE-2020-2758	VHWA	UAF	**A**,**I**	RCE	✓

giving the attacker full read-write access to a chunk now treated as host memory.

This transformation captures the core capability of CDA^C. Once the uninitialized free is redirected to a guest-controlled chunk, the vulnerability becomes a stable UAF primitive that already provides full attacker control over the freed object. From this point onward, the remaining exploitation steps follow well-established and standardized UAF exploitation workflows, such as steering the reallocated chunk toward security-critical structures or converting the primitive into control-flow hijacking. When stronger capabilities are desired, this UAF condition can also be transformed into other CDA variants. CDA^O enables metadata manipulation, CDA^I enables targeted information disclosure, and CDA^A becomes possible when a callable field is present. The NVMe case therefore illustrates how CDA turns a structurally difficult vulnerability into a reliable exploitation substrate, while still allowing further capability escalation through other CDA variants when needed.

Analysis of Failed Cases. During our evaluation, we also analyzed the remaining cases in Table I where CDA was not achieved. We manually examined all the listed pointer corruption vulnerabilities and found that the failures generally fall into two categories: (1) some corrupted pointers are never dereferenced afterward, preventing further exploitation; (2) in other cases, the corrupted values are not attacker-controllable, making it infeasible to redirect it to a usable payload.

D. Overhead Analysis of the CDA

We measured the dynamic overhead of CDA by evaluating two steps: input synthesizing (Sec.IV-C) and exploit assembling (Sec.IV-D), as shown in IV. Five representative vulnerabilities across QEMU and VirtualBox were selected to cover typical device types and bug categories. For input generation, we report the average time (ten runs) required to find the first usable, low-noise cross-domain gadget. The results show stable completion times ranging from 268 to 531 seconds, depending on gadget-family complexity. For exploit building, the automated combination of primitives incurs negligible cost. However, a small manual adjustment is required to realign heap-grooming offsets, since certain gadget-triggering inputs introduce minor side effects that slightly shift the heap layout. This validation step results in an overall average build time of 14.6 to 19.2 minutes across vulnerabilities. Overall, these measurements show that CDA incurs an acceptable and predictable dynamic overhead, enabling end-to-end exploit construction with manageable cost.

CVE id/Name	Device	Input Gen.	Exploit Build
CVE-2024-3446	virtio-gpu	268 s	17.8 min
CVE-2024-8612	virtio-blk	274 s	18.3 min
CVE-2021-3682	USB-redir	517 s	19.2 min
Scavenger	NVMe	531 s	14.6 min
CVE-2023-3180	virtio-crypto	522 s	18.9 min

TABLE IV: Dynamic runtime (10 runs) of input synthesizing and exploit assemblying across representative vulnerabilities.

VI. Discussion

A. Possible Defense Mechanism

Memory Access Control. We believe that preventing the host from accessing guest memory by default could serve as a viable defense strategy against CDA. In the kernel space, defenses such as Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Prevention (SMAP) [18] already prevent kernel-mode code from directly accessing user-mode memory, effectively isolating different privilege levels. A similar mechanism could be adopted in the hypervisor setting: the host would be restricted from accessing guest memory by default, with hardware enforcing a strong access barrier between host and guest.

Such isolation would prevent unintended or malicious host interactions with guest memory, significantly reducing the risk of CDA exploits that rely on redirecting host pointers into

guest-controlled regions. However, certain operations—such as device emulation or management tasks—legitimately require the host to access guest memory. To support these needs securely, access could be selectively permitted through dedicated APIs or hardware instructions that temporarily relax the isolation under tightly controlled conditions. Inspired by the `STAC` and `CLAC` instructions used by SMAP, hypervisors could adopt similar mechanisms to grant temporary, auditable access to guest memory only when explicitly invoked.

Gadget Reduction. An alternative line of defense focuses on reducing the presence of cross-domain gadgets. To achieve this, hypervisors should avoid storing raw guest-HVA pointers in host memory. Instead, guest memory references can be represented using opaque tokens such as handles or offsets, have no direct dereference semantics. When memory access is required, these tokens are resolved through a secure lookup mechanism to obtain the corresponding HVA. Another complementary approach is to cryptographically encode guest-HVA pointers, similar to pointer authentication (PAC), before storing them, and to decode them only upon verified use. This ensures that no valid guest address remains in cleartext within host memory, thereby preventing potential misuse. These designs eliminate the presence of dereferenceable guest addresses in host memory, thereby blocking a key prerequisite for CDA-based exploitation.

B. Potential Existence of Additional Gadgets

Following a best-effort approach, we collected translation functions from official documentation and validated them through static analysis. However, undocumented routines may still exist and inadvertently retain guest-HVA pointers, forming implicit cross-domain gadgets beyond our current coverage. While our analysis captures the major patterns observed in practice, we acknowledge that additional cases could emerge and warrant further investigation.

C. Limitations on Confidential VMs

While our CDA framework demonstrates effective cross boundary exploitation in traditional virtualized environments, its applicability to Confidential Virtual Machines (CVMs), such as those protected by AMD SEV-SNP or Intel TDX, is limited. These platforms enforce strict memory encryption and isolation: guest memory and CPU state are encrypted and integrity protected, so that the hypervisor or host cannot directly read or modify guest data. As a result, key primitives required by CDA will be restricted.

However, shared pages remain accessible in plaintext to the host, such as GHCB pages, TDX shared buffers, and virtio shared rings [19]. If the hypervisor does not properly validate and constrain its use of these shared regions, CDA-style attacks may still reappear, since the attacker fully controls these buffers. Overall, although CVMs significantly reduce the gadget surfaces and translation visibility needed by CDA, the fundamental idea of cross-domain pointer manipulation remains relevant and merits further exploration in isolation-enhanced environments.

D. Applicability to KVM Integration

Within the QEMU virtualization setting considered in this work, our scope is confined to the userspace portion, where guest memory is mapped into the process address space and exposed as host-side HVAs. At the same time, we note that the KVM subsystem (Kernel-based Virtual Machine [20]) also maintains HVA pointers as part of its normal operation. Although this work does not examine whether similar CDA-like behaviors could arise at the kernel boundary, we do not rule out this possibility. A systematic investigation of cross-domain pointer interactions within pure KVM or other kernel-level virtualization paths remains promising future work and may reveal additional opportunities for understanding CDA beyond the userspace components of the hypervisor.

E. Applicability to Other Scenarios

The core concept of CDA revolves around exploiting the weak isolation between two distinct memory spaces. By taking advantage of this weak separation, an attacker can manipulate the memory layout of the target space by using the controllable environment of their own space, thereby facilitating successful exploitation. This approach is particularly dangerous because it leverages the interaction between two spaces that lack rigorous boundary validation, making a wide range of systems vulnerable to this type of attack. Essentially, any environment where two isolated memory spaces interact—such as Software Guard Extensions (SGX), Trusted Execution Environments (TEE), microservices architectures, and inter-process communication systems—can be at risk if they do not enforce strict boundary checks. The concept becomes especially effective in minimalistic or constrained architectures, where available primitives are limited, but one space remains fully controllable by the attacker. In such scenarios, where a weak boundary exists between the interacting domains, an attacker can manipulate one domain to influence the other, bypassing conventional security defenses. The potential impact of CDA-style attacks is broad, extending to any system where domain separation is not rigorously maintained. This underscores the critical need for robust boundary validation and strict isolation policies to protect against such sophisticated attacks, as any weakness in these areas could provide an entry point for exploitation.

VII. Related Work

Hypervisor Vulnerability Discovery. In the field of virtualization, vulnerability discovery has always been a focal point for security researchers. Early efforts began with dump fuzzing [21], [22], [23], followed by advancements in addressing data input issues [8], [10], [24], [25], and later incorporating generation of high-quality test case sequences [26], [17], [27], [28] and hardware-assisted methods [29], [9]. For example, Hyper-Cube [24] utilizes a custom operating system to achieve high-throughput, multi-dimensional fuzzing. Nyx [29] employs full-system snapshots and a hardware-assisted coverage framework to fuzz hypervisors. V-Shuttle [8] uses DMA redirection to flatten nested structures, while MORPHUZZ [10] reshapes the input space of virtual devices by

collecting feedback from the core hypervisor's interface APIs. ViDeZZo [17] leverages static analysis to extract dependencies within and between messages. Over time, virtualization vulnerability discovery techniques have made significant progress. However, vendors are increasingly focused on whether these vulnerabilities can be exploited in cloud environments [30].

Kernel Exploitation. The development of kernel exploitation techniques has progressed through several stages, gradually forming a rich methodological framework. Early research primarily focused on bypassing various kernel protections to ensure successful exploitation (e.g., [31], [32]). As the field matured, the research focus shifted toward generating exploits for specific vulnerabilities, exemplified by tools like Collision, Fuze, Koobe, and Exprace [33], [34], [35], [36], [37], [38], [39], driving significant advancements in the precision and stability of exploit generation. Building on this foundation, further research has focused on enhancing the stability of heap exploitation [40], [41] and actively exploring new exploitation primitives [42], [11], [43], [44], [45]. New exploitation methods continue to emerge, such as the DirtyCred and PSpray techniques [46], [47], injecting fresh perspectives into kernel exploitation. Meanwhile, the advent of the Automated Exploit Generation (AEG) technology has further increased automation [48], [49], making exploitation more efficient and accessible. These advancements indicate that kernel exploitation has now established a solid technical foundation and a mature toolkit. By comparison, exploitation in virtualization systems remains far less developed. Although cross-domain ideas such as ret2usr [32] show how kernels can redirect execution into user memory, these techniques do not translate to hypervisors, which lack direct access to guest memory and must instead rely on GPA-to-HVA translation. CDA is novel in that it systematically reveals when hypervisors unintentionally reintroduce guest-HVA pointers into host memory, formalizes these behaviors into four semantic variants, and turns guest memory into a practical, reusable exploitation substrate.

Hypervisor Exploitation. Current VM-to-hypervisor exploitation research has produced numerous practical VM-escape attacks across both open-source and proprietary hypervisors—including QEMU [3], [2], [16], [50], [51], VirtualBox [52], [53], VMware [6], [7], [5], [13], [54], and ESXi [4]. These attacks span virtual devices, network stacks, and shared services and primarily rely on identifying exploitable host-side structures such as function pointers or corrupted objects. As a result, prior techniques remain highly ad-hoc, heavily dependent on expert-crafted primitives, and tied to specific host-resident artifacts. Moreover, existing work is almost entirely **host-centric**, treating guest memory only as attacker input rather than as a first-class exploitation substrate. Although prior studies (e.g., [16]) showed isolated cases where hypervisors may dereference guest-controlled addresses, they did not characterize the underlying conditions, generality, or semantic variants of such cross-domain behaviors. **CDA departs fundamentally from these earlier VM-to-hypervisor attacks** by systematizing this phenomenon: we identify the root causes of cross-domain pointer flows, formalize four canonical variants based on pointer-use semantics, and demonstrate that these opportunities arise broadly across hypervisor codebases. This work therefore elevates CDA from scattered empirical observations to a **general, structured exploitation paradigm**—a perspective not captured in prior literature.

VIII. Conclusion

In virtualization environments, exploiting pointer corruption vulnerabilities is particularly challenging due to the scarcity of exploitable data structures in the host. Traditional techniques often lack a stable foothold for redirecting corrupted pointers, limiting their effectiveness. In this paper, we provide the first systematic characterization of CDA, which leverages weak guest-host isolation to redirect corrupted host pointers to attacker-controlled payloads in guest memory. Our findings reveal that cross-domain attacks pose a significant and emerging threat to the security of existing virtualization infrastructures. This transforms guest memory into a reusable and controllable primitive for exploitation. Additionally, we develop an automated system to identify cross-domain gadgets and construct exploitation chains, demonstrating the approach's practicality across real-world hypervisors. We hope this work raises awareness in the virtualization and cloud security communities. We hope this work raises awareness in the virtualization and cloud security communities about the risks of implicit trust in guest memory and motivates stronger isolation mechanisms.

Ethics Consideration

In conducting our research, we focus exclusively on known hypervisor vulnerabilities, all of which have already been patched by vendors. Furthermore, all experiments are conducted in controlled local environments, fully isolated from any public cloud infrastructure. As such, our work does not pose any risk to production systems or real-world deployments, and does not raise any ethical concerns. In addition, we have proactively communicated our findings to the QEMU development team to discuss the broader implications of CDA and potential mitigation strategies. While the developers are still assessing the long-term impact, we have proposed several mitigation directions based on our analysis.

References

[1] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," International Journal of Innovative Technology and Exploring Engineering, vol. 2, no. 3, pp. 222–225, 2013.

[2] Z. Shao, J. Weng, and Y. Zhang, "3d red pill: A guest-to-host escape on qemu/kvm virtio devices," Black Hat Asia, 2020.

[3] N. Elhage, "Virtunoid: A kvm guest! host privilege escalation exploit," Black Hat USA, vol. 2011, 2011.

[4] H. Zhao, Y. Zhang, K. Yang, and T. Kim, "Breaking turtles all the way down: An exploitation chain to break out of vmware esxi." in WOOT@ USENIX Security Symposium, 2019.

[5] "Pwning vmware, part 2: Zdi-19-421, a uhci bug, 2020," https://nafod.net/blog/2020/02/29/zdi-19-421-uhci.html, 2020.

[6] "Speedpwning vmware workstation," https://www.synacktiv.com/sites/default/files/2020-10/Speedpwning_VMware_Workstation.pdf, 2020.

[7] "The great escapes of vmware: A retrospective case study of vmware guest-to-host escape vulnerabilities," https://www.blackhat.com/docs/eu-17/materials/eu-17-Mandal-The-Great-Escapes-Of-Vmware-A-Retrospective-Case-Study-Of-Vmware-G2H-Escape-Vulnerabilities.pdf, 2017.

[8] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 2197–2213.

[9] A. Bulekov, Q. Liu, M. Egele, and M. Payer, "{HYPERPILL}: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface," in 33rd USENIX Security Symposium (USENIX Security 24), 2024, pp. 919–935.

[10] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1221–1238.

[11] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1165–1184.

[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 2025 IEEE Symposium on Security and Privacy (SP). IEEE, 2025, pp. 810–828.

[13] "A ctf-style escape journey on vmware workstation," https://hitcon.org/2020/slides/A%20CTF-Stype%20Escape%20Journey%20on%20VMware%20Workstation.pdf, 2020.

[14] "Convert the format of an image in alibaba cloud," https://www.alibabacloud.com/help/en/ecs/user-guide/convert-the-format-of-an-image, 2025.

[15] Z. Ma, Q. Liu, Z. Li, T. Yin, W. Tan, C. Zhang, and M. Payer, "Truman: Constructing device behavior models from os drivers to fuzz virtual devices," in 32nd Annual Network and Distributed System Security Symposium, NDSS, 2025, pp. 24–28.

[16] G. Pan, X. Lin, X. Ying, J. Wang, and C. Wu, "Scavenger: Misuse error handling leading to qemu/kvm escape," Black Hat Asia, 2021.

[17] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "Videzzo: Dependency-aware virtual device fuzzing," in 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023, pp. 3228–3245.

[18] "Supervisor mode access prevention," https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention, 2023.

[19] "Confidential computing 101," https://docs.enclaive.cloud/confidential-cloud/technology-in-depth/amd-sev/technology/fundamentals#:~:text=In%20standard%20use%20cases%2C%20VMs,for%20securing%20I/O%20traffic.&text=AMD%20SEV%20VMs%20determine%20whether,various%20threats%20in%20virtualized%20environments, 2024.

[20] "Kernel virtual machine," https://linux-kvm.org/page/Main_Page, 2025.

[21] S. Bleikertz, "Xenfuzz," https://www.openfoo.org/blog/xen-fuzz.html, 2021.

[22] "Viridian fuzzer," https://github.com/mwrlabs/ViridianFuzzer, 2021.

[23] M. Gorobets, O. Bazhaniuk, A. Matrosov, A. Furtak, and Y. Bulygin, "Attacking hypervisors via firmware and hardware," Black Hat USA, 2015.

[24] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Hyper-cube: High-dimensional hypervisor fuzzing." in NDSS, 2020.

[25] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "Vdf: Targeted evolutionary fuzz testing of virtual devices," in Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings. Springer, 2017, pp. 3–25.

[26] C. Myung, G. Lee, and B. Lee, "{MundoFuzz}: Hypervisor fuzzing with statistical coverage testing and grammar inference," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1257–1274.

[27] Y. Liu, S. Chen, Y. Xie, Y. Wang, L. Chen, B. Wang, Y. Zeng, Z. Xue, and P. Su, "Vd-guard: Dma guided fuzzing for hypervisor virtual device," in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 1676–1687.

[28] Z. Zhang, G. Pan, R. Wang, Y. Tao, Z. Pan, C. Tu, M. Zhang, Y. Li, Y. Shen, and C. Wu, "Insvdf: Interface-state-aware virtual device fuzzing," in 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE Computer Society, 2025, pp. 727–727.

[29] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 2597–2614.

[30] S. E. Marios Pomonis, "Virtual escape; real reward: Introducing google's kvmctf," https://security.googleblog.com/2024/06/virtual-escape-real-reward-introducing.html, 2024.

[31] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in USENIX security symposium, 2009, pp. 383–398.

[32] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 957–972.

[33] 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 SIGSAC Conference on Computer and Communications Security, 2015, pp. 414–425.

[34] 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 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 781–797.

[35] 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 Conference on Security Symposium, 2020, pp. 1093–1110.

[36] Y. Lee, C. Min, and B. Lee, "Exprace: Exploiting kernel races through raising interrupts." in USENIX Security Symposium, 2021, pp. 2363–2380.

[37] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities." in USENIX Security Symposium, 2019, pp. 1187–1204.

[38] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying." in NDSS, 2017.

[39] H. Cho, J. Park, J. Kang, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers," in 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020.

[40] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 1707–1722.

[41] 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 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 71–88.

[42] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in Proceedings of the 29th USENIX Conference on Security Symposium, 2020, pp. 1111–1128.

[43] J. Koschel, P. Borrello, D. C. D'Elia, H. Bos, and C. Giuffrida, "Uncontained: Uncovering container confusion in the linux kernel," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 5055–5072.

[44] R. Wang, K. Chen, C. Zhang, Z. Pan, Q. Li, S. Qin, S. Xu, M. Zhang, and Y. Li, "{AlphaEXP}: An expert system for identifying {Security-Sensitive} kernel objects," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4229–4246.

[45] E. Avllazagaj, Y. Kwon, and T. Dumitraș, "{SCAVY}: Automated discovery of memory corruption targets in linux kernel for privilege escalation," in 33rd USENIX Security Symposium (USENIX Security 24), 2024, pp. 7141–7158.

[46] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, 2022, pp. 1963–1976.

[47] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "Pspray: Timing {Side-Channel} based linux kernel heap exploitation technique," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6825–6842.

[48] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 763–779.

[49] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "Maze: Towards automated heap feng shui." in USENIX Security Symposium, 2021, pp. 1647–1664.

[50] "Venom: (cve-2015-3456)," https://access.redhat.com/zh_CN/articles/1447963, 2015.

[51] Q. Li, G. Pan, H. He, and C. Wu, "Matryoshka trap: Recursive mmio flaws lead to vm escape."

[52] "Breaking out of the box: Technical analysis of virtualbox vm escape with windows lpe," https://www.synacktiv.com/sites/default/files/2023-10/hexacon_breaking_out_of_the_box.pdf, 2023.

[53] "Pwn2own 2020: Oracle virtualbox escape," https://starlabs.sg/blog/2020/09-pwn2own-2020-oracle-virtualbox-escape/, 2020.
[54] "Urb excalibur: The new vmware all-platform vm escapes," https://i.blackhat.com/Asia-24/Presentations/Asia-24-Jiang-URB-Excalibur-The-New-VMware-All-Platform-VM-Escapes.pdf, 2024.

APPENDIX A
ADDRESS TRANSLATION FUNCTIONS

Table V lists the hypervisor functions responsible for guest physical address (GPA) to host virtual address (HVA) translation across two widely used virtualization platforms. In QEMU, this functionality is implemented by functions such as `address_space_map()` and other variants, which manage memory mapping and access for guest memory regions. In VirtualBox, similar roles are performed by functions such as `PGMPhysWrite()` and PGMR3PhysBulkGCPhys2CCPtrExternal(), which provide fine-grained control over guest-to-host address resolution and data access.

Category	Translation Functions

QEMU	address_space_map();
	address_space_unmap();
	address_space_read_full();
	address_space_read();
	address_space_write();

Virtualbox	PGMPhysWrite(); PGMPhysRead(); PGMR3PhysBulkGCPhys2CCPtrExternal(); PGMR3PhysBulkGCPhys2CCPtrReadOnlyExternal(); PGMR3PhysGCPhys2CCPtrExternal();

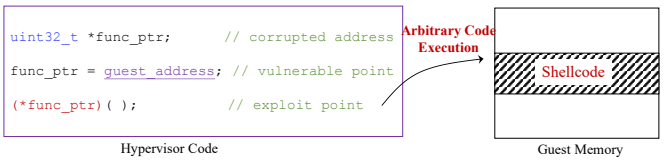
TABLE V: Hypervisor address translation functions.

APPENDIX B
CDA VARIANTS

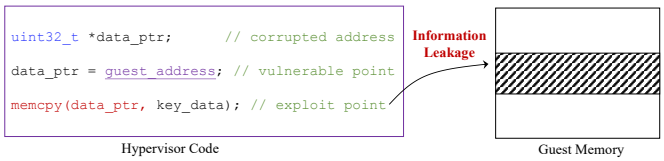
We summarize four variants of CDA, categorized by the usage context of the corrupted pointer—namely when the vulnerable code performs a **free**, **call**, **read**, or **write** operation on it—which collectively cover all possible exploitation effects of CDA, as detailed below:

1) Arbitrary Code Execution (CDA^{A}): We begin by introducing the first CDA variant: arbitrary code execution, as illustrated in Figure 14a. This variant targets cases where a pointer corruption vulnerability results in a corrupted function pointer, which is subsequently dereferenced by the hypervisor. Instead of attempting to locate a valid function address within the host—an approach made difficult by ASLR, limited code reuse gadgets—CDA takes a more flexible path by redirecting the corrupted pointer to guest memory. Guest memory, while isolated from the guest's perspective, is still directly accessible by the host. Thus, once the corrupted function pointer is invoked, the control flow is transparently transferred to an attacker-controlled region in guest space, where a staged shellcode resides. This effectively bypasses the need for reliable host-side code reuse primitives or shellcode injection mechanisms.

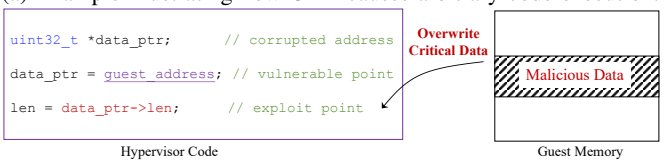
2) Information Leakage (CDA^{I}): Next, we introduce the second CDA variant: sensitive information leakage, as illustrated in Figure 14b. This scenario arises when a pointer corruption vulnerability leaves behind a corrupted data pointer that is later used in a write operation. Instead of pointing to a legitimate buffer in host memory, CDA redirects this



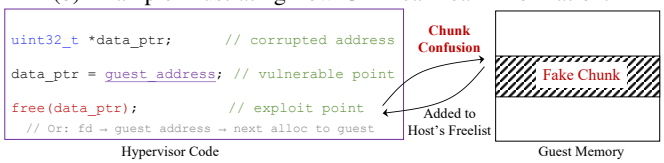
(a) Example illustrating how CDA causes arbitrary code execution.



(b) Example illustrating how CDA can leak information.



(c) Example illustrating how CDA can overwrite critical data.



(d) Example illustrating how CDA can craft chunk confusion.

Fig. 14: CDA variants to exploit pointer corruption vulnerabilities.

pointer to an attacker-controlled address within the guest memory space. As a result, when the hypervisor writes data to the corrupted pointer, it unintentionally writes directly into guest memory—effectively leaking sensitive information from the host to the guest. This variant enables a side-effect-free and stealthy data exfiltration channel, without requiring any interaction through standard I/O mechanisms such as DMA buffers or MMIO transactions. Traditional interfaces often impose strict constraints, such as requiring pre-allocated transfer objects, matching data formats, or bounded transfer sizes. CDA bypasses these limitations and eliminates the need for additional communication logic.

3) Critical Data Overwriting (CDA^O): We now present the third CDA variant: critical data overwriting, as illustrated in Figure 14c. In this scenario, a pointer corruption vulnerability leaves behind a corrupted data pointer that is later used in a read operation. CDA redirects this pointer to guest memory, causing the hypervisor to read attacker-controlled data directly from the guest space. As a result, the guest-controlled input is implicitly trusted and used to update sensitive fields within the hypervisor. This variant effectively allows the attacker to inject arbitrary values into critical data structures without the need for traditional write primitives. Since the guest space is entirely under the attacker's control, the attacker can craft malicious data with fine-grained control over content, layout, and size, making the overwrite both precise and stealthy. By abusing the host's implicit trust in the source of a read operation, this variant converts a seemingly benign data corruption into a powerful logic corruption primitive, enabling further exploitation such as privilege manipulation, control flag flipping, or fake object injection.

4) Chunk Confusion (CDA^C): Finally, we describe the last CDA variant: chunk confusion, as illustrated in Figure 14d. This attack occurs when the corrupted address produced by a pointer corruption vulnerability is involved in memory allocation or deallocation, leading to inconsistencies in the hypervisor's heap management logic. We formalize two representative scenarios:

(1) If the corrupted address is used in a `free()` operation, CDA redirects it to guest memory, causing the hypervisor to erroneously free an attacker-crafted fake chunk residing in the guest space. This fake chunk is then inserted into the host's freelist and treated as a valid memory region by the host-side allocator. As a result, future memory allocation requests from the hypervisor may return pointers to guest memory, allowing the attacker to gain full read/write access to sensitive host data structures allocated on top of it. This effectively enables a guest-assisted heap spraying primitive, leveraging the hypervisor's own memory manager.

(2) Similarly, when the corrupted value affects heap metadata—such as a manipulated `fd` pointer in a free chunk—CDA can redirect the allocator's next chunk candidate into guest space. During the next allocation, the hypervisor mistakenly allocates memory from the attacker-controlled guest region. This allows the guest to pre-position payloads or metadata that will later be interpreted as legitimate host structures, enabling logic corruption or type confusion attacks without violating memory access protections.

These two patterns demonstrate how CDA can subvert the hypervisor's memory allocator by injecting crafted objects from the guest domain, enabling precise control over the heap layout and allocation behavior in host space. Unlike traditional heap exploits that require complex heap feng shui in host memory, CDA provides a lightweight and deterministic alternative.

Appendix C
Example of Elastic Cross-Domain Gadget

To illustrate the versatility of elastic cross-domain gadgets, we present a representative example from the virtio-GPU device. As shown in Figure 15, the device allocates a data structure whose size is directly controlled by the guest (line 3). This structure contains an array of `iovec` entries, each embedding a length field (line 7) and a guest physical address (line 8). The resulting layout forms a one-to-one mapping table of guest addresses, with entries spaced at regular 0x10-byte intervals.

Because both the total allocation size and the number of embedded guest addresses are determined by the guest, this layout acts as a highly flexible **elastic cross-domain gadget**, capable of adjusting its size, alignment, and pointer positions

TABLE VI: Summary of cross-domain gadgets in the QEMU: Each gadget is triggered by a guest-controllable field and mapped to host address space via `MMIO` or `Timer/BH` mechanisms.

Gadget Family	Upper Function	Translation Function	HVA Variable	GPA Source Field	Trigger Type	Count
DMA gadget	dma_memory_map	address_space_map	ad→lst	AHCIPortRegs→fis_addr	MMIO (4), BH (0)	4
	pci_dma_map	address_space_map	ring→page	txd.addr	MMIO (10), BH (0)	10
	dma_memory_read	address_space_read_full	ram_ptr	s→tx_descriptor	MMIO (81), BH (21)	102
	dma_memory_write	address_space_write	ram_ptr	s→tx_descriptor	MMIO (91), BH (16)	107
USB gadget	usb_packet_map	address_space_map	packet→iov	sgl→sg[num_sg].base	MMIO (3), BH (11)	14
	get_dwords	address_space_read_full	ram_ptr	q→qhaddr	MMIO (1), BH (35)	36
	put_dwords	address_space_write	ram_ptr	q→qhaddr	MMIO (2), BH (44)	46
	xhci_dma_read_u32s	address_space_read_full	ram_ptr	sctx→pctx	MMIO (22), BH (8)	30
	xhci_dma_write_u32s	address_space_write	ram_ptr	sctx→pctx	MMIO (17), BH (4)	21
	xhci_write_event	address_space_write	ram_ptr	intr→er_start	MMIO (17), BH (5)	22
Virtio gadget	virtqueue_map_desc	address_space_map	iov[num_sg].iov_base	desc[num_sg].addr	MMIO (40), BH (16)	56
	virtio_gpu_create_mapping_iov	address_space_map	iov[num_sg].iov_base	desc[num_sg].addr	MMIO (0), BH (2)	2
Display gadget	cpu_physical_memory_map	address_space_map	data	s→dispc.l[0].addr[0]	MMIO (1), BH (0)	1
Block device gadget	dma_blk_cb	address_space_map	dbs→iov	req→sg.qsg	MMIO (4), BH (6)	10
SCSI gadget	lsi_mem_read	address_space_read	ram_ptr	s→dsp	MMIO (4), BH (0)	4
	lsi_mem_write	address_space_write	ram_ptr	s→dsp	MMIO (4), BH (0)	4
PCI gadget	pci_dma_read	address_space_read_full	ram_ptr	r→bdbar	MMIO (59), BH (120)	179
	pci_dma_write	address_space_write	ram_ptr	desc.buffer_addr	MMIO (42), BH (22)	64
SDHCI gadget	sdhci_do_adma	address_space_read_full	ram_ptr	dscr.addr	MMIO (12), BH (2)	14
	sdhci_sdma_transfer_multi_blocks	address_space_read_full	ram_ptr	s→sdmasysad	MMIO (9), BH (1)	10
	sdhci_sdma_transfer_multi_blocks	address_space_write	ram_ptr	desc.buffer_addr	MMIO (9), BH (1)	10
Total						**772**

```
int virtio_gpu_create_mapping_iov(VirtIOGPU *g
    ↪ , uint32_t nr_entries, struct iovec **
    ↪ iov)
{
    *iov = g_malloc0(sizeof(struct iovec) *
        ↪ nr_entries);
    for (i = 0; i < nr_entries; i++) {
        uint64_t addr = ents[i].addr;
        uint32_t len = ents[i].length;
        (*iov)[i].iov_len = len;
        (*iov)[i].iov\_base =
            ↪ dma_memory_map(addr, &len);
    }
    return 0;
}
```

Fig. 15: A elastic gadget in `virtio_gpu_create_mapping_iov` that calls `dma_memory_map` with a guest-provided address `addr` and length `len`. The return value is stored in `iov_base`, forming a host-accessible pointer derived from guest input.

to match diverse heap-based exploitation needs. Importantly, similar elastic patterns emerge across multiple virtio backends, indicating that this capability is inherent to their allocation and structure design rather than a device-specific anomaly.

APPENDIX D
DETAILED CROSS-DOMAIN GADGETS

To support our analysis of CDA, we provide a comprehensive breakdown of gadget instances across various device subsystems. Table VI summarizes 776 gadget instances identified in our study, categorized by gadget family, trigger function pair, and their guest-controlled GPA source field.