# Page Phantoms: Zero-I/O Tampering of the Linux Page Cache

Jia Jia

**Abstract**

Virtual Machine Introspection (VMI) and in-guest Endpoint Detection and Response (EDR) systems make it increasingly difficult to stealthily tamper with files inside virtual machines. In high-value deployments, the host continuously hashes guest memory—especially kernel data structures such as inodes—while an in-guest kernel module hooks system calls, the Virtual File System (VFS), and storage I/O. Together they form a “god-mode” dual-layer defense.

This paper presents Page Phantoms, a set of techniques that bypass this dual-layer defense by abandoning the traditional I/O path and attacking instead the lifecycle of reclaimable physical pages in the Linux kernel. By abusing the interaction between the page cache, XArray, and the Multi-Generational LRU (MGLRU) algorithm, we demonstrate two attacks. The first is a file-system-based Zero-I/O attack, in which file contents are modified purely in memory and later rolled back to a consistent state without ever triggering file-system or disk I/O initiated by the attacker. The second is a memory-page-based Zero-I/O attack that achieves stealthy disk persistence by forging page-cache pages, inserting them into the page-cache tree, and letting the kernel’s own reclaim and write-back machinery commit them to disk.

We evaluate our techniques against a hardened environment in which the host VMM monitors guest inodes via GPA-level hashing, and the guest EDR hooks VFS and page-cache APIs and reports suspicious call stacks. The attack uses only exported, stable kernel symbols and remains reliable under realistic server workloads, while evading both host-side and guest-side monitoring.

Virtualization greatly increases the observability of guest systems from the host. With frameworks such as LibVMI and KVM-VMI, a hypervisor can inspect a guest’s memory and execution state from outside the VM, without assistance from any in-guest agent. At the same time, security vendors deploy EDR modules inside the guest, which hook system calls, the VFS layer, and block-device I/O to detect file tampering and kernel-structure anomalies.

In high-value production environments these two capabilities are often combined. On the guest side, the EDR hooks system calls and VFS entry points, hooks key page-cache APIs such as `generic_perform_write`, `filemap_read`, and `filemap_write_and_wait`, and can optionally export call stacks to the host for out-of-band integrity checks. On the host side, the VMM uses VMI to monitor the integrity of selected guest-physical regions (for example, inodes and task structures) by periodically hashing them, and it operates independently of any trusted in-guest agent.

Even if an attacker removes the in-guest EDR entirely, the host can still raise alerts based on deviations in guest memory. This dual-layer defense has effectively become the de-facto baseline for Linux security in virtualized environments. This work asks: can an attacker still perform useful file tampering inside such a VM in a way that is both practical and prohibitively expensive to monitor? We answer “yes” by steering away from the usual I/O stack and instead targeting the lifecycle of reclaimable physical pages. Rather than attacking system calls or persistent filesystem metadata directly, we manipulate the internal data structures that govern how file-backed pages are cached, aged, reclaimed, and written back.

## 1 Threat Model and Attack Goals

### 1.1 Threat Model

We assume the following environment and attacker capabilities:

1) The attacker has already obtained kernel-level privileges inside the guest (for example, via a local privilege-escalation exploit or a malicious kernel module). We are in the post-exploitation phase.
2) Inside the guest, an EDR kernel module: Hooks system calls and VFS entry points; Hooks page-cache APIs (for example, `generic_perform_write`, `filemap_read`, `filemap_write_and_wait_range`); May report suspicious call stacks back to the host.
3) On the host side, the VMM:
   Uses VMI to compute hashes over selected guest-physical regions, especially inode structures and other core kernel objects; Raises alerts on unexpected changes and does not rely on any trusted in-guest component.
4) Neither the host nor the guest kernel itself are compromised.

### 1.2 Attack Goals

Given this environment, the attacker aims to modify the contents of target files inside the guest and, optionally:Keep the modification purely in memory (no on-disk changes); or Persist the modification to disk.

While doing so, the attacker must:

1) Avoid monitored system calls, VFS operations, and page-cache write-back APIs;
2) Avoid detectable modifications to monitored kernel structures such as inodes;
3) Maintain system stability and tolerate realistic server workloads.

## 2 Background

### 2.1 Memory Model and Buddy Allocator

Mainline Linux typically uses the SPARSEMEM + VMEMMAP memory model, representing every physical frame with a `struct page` (and its higher-level abstraction `struct folio`). At the lowest level, the buddy allocator manages pools of free pages, supplying them to higher-level allocators, including:

1) The page cache (file-backed pages);
2) Anonymous memory (user-space heap/stack and `mmap` regions);
3) Slab/SLUB allocators for kernel objects;
4) `vmalloc` regions, and others.

### 2.2 Page Cache and XArray

The page cache is an in-kernel caching layer for file data. When a process accesses a regular file:

1) The kernel first looks for the corresponding page in the page cache.
2) On a hit, data is served directly from memory.
3) On a miss, the data is read from disk and the freshly fetched page is inserted into the cache for future use.

Most executables, shared libraries, logs, configuration files, and scripts traverse this cache. Even when Direct I/O is used, I/O is still ultimately backed by physical pages; they simply participate through a different path.

Each file is represented by an `address_space` object, which owns an XArray field `mapping->i_pages`:

1) XArray is a core kernel data structure optimized for managing indexed objects.
2) In the page cache, it maps file offsets (page indices) to physical pages (folios).

### 2.3 LRU and MGLRU

Historically, Linux used a classic LRU (Least Recently Used) algorithm:

1) Newly accessed or loaded pages enter an active list.
2) Pages that are not accessed for some time may be demoted to an inactive list.
3) When the kernel needs to reclaim memory, it primarily selects victims from the inactive list.

This simple design is easy to implement but prone to page thrashing: pages can be evicted shortly before being required again, forcing redundant disk I/O.

To address this, Linux introduced MGLRU (Multi-Generational LRU), which was merged into mainline in version 6.1 and is increasingly enabled by default. Its core ideas are:

1) Pages are grouped into multiple generations based on recency of use.
2) An aging process periodically moves long-unused pages toward older generations.
3) A promotion process uses hardware PTE access bits to detect recently accessed pages and promote them back to the youngest generation.
4) Reclaim focuses on the oldest generation, improving efficiency and preserving truly hot pages.

In practice, MGLRU significantly reduces thrashing and provides a more accurate notion of the “working set”.

### 2.4 Reclaimable Physical Pages

In this paper a reclaimable physical page is defined as a page-sized physical frame that has backing storage and can be paged in and out under LRU/MGLRU management.
Such pages originate from the buddy allocator and are broadly divided into:

1) Anonymous pages: user stack, heap, `mmap` regions, Direct I/O buffers, and so on;
2) File-backed pages: page-cache pages, `tmpfs`, `shmem`.

**Important nuances:**

1) Pseudo-filesystems such as `/proc` do not use the page cache and have no real backing store, so they lie outside the LRU/MGLRU reclaimable set.
2) `tmpfs` and `shmem` are special: their pages are stored in XArray like regular file pages, but in LRU/MGLRU and reclaim policy they are treated similarly to anonymous, swappable pages.

Note that “reclaimable” is a logical property, not a guarantee. On systems where swap is disabled, anonymous pages may have no backing store; they cannot actually be written out even though they still participate in MGLRU as logically reclaimable pages.

### 2.5 Relationship Between XArray and LRU/MGLRU

When a physical page (folio) is inserted into a file’s XArray as a page-cache entry—for example, via `filemap_add_folio()`—the kernel also links it into the LRU or MGLRU lists.

Key consequences are:

1) Every file-backed page that resides in the page cache also appears in the LRU/MGLRU sets.
2) XArray is a per-file view that answers “which pages belong to this file at which offsets?”.
3) LRU/MGLRU is a global view that answers “across the entire system, which pages are the best candidates for reclaim?”.
4) When the kernel needs to free memory, it follows the ordering suggested by LRU/MGLRU to pick victim pages.
5) During reclaim, dirty pages are flushed to storage; clean pages are returned to the buddy allocator for reuse.

This dual view—per file via XArray, global via MGLRU—is central to building the Page Phantoms attack.

## 3 Defining Zero-I/O

In the context of this work, Zero-I/O does not mean that no disk I/O happens at all. Instead, it means:
No observable I/O initiated by the attacker through monitored APIs or code paths.
Page Phantoms supports two complementary flavors of Zero-I/O.

3.1 File-System-Based Zero-I/O

1) The attacker never opens, reads, or writes the target file through normal filesystem APIs.
2) Instead, they locate and modify the physical pages that currently cache the file's contents.
3) After modification, these pages are detached from the file and returned to the buddy allocator as reclaimable free pages.
4) The tampered data only exists transiently in memory; no file-system calls or disk I/O are directly issued by the attacker.

3.2 Memory-Page-Based Zero-I/O (with persistence)

1) The attacker does not modify the original file-mapped pages.
2) A fresh physical page is allocated and forged into a valid page-cache page.
3) This forged page is inserted into the file's page-cache XArray, mimicking a normal page-in event.
4) The kernel's background write-back threads later flush this page to disk, making the change persistent.
5) All I/O is initiated by the kernel's own reclaim and write-back mechanisms; the attacker does not call the usual page-cache APIs that EDR/VMM monitor.

Both flavors operate entirely within internal memory-management structures, bypassing the traditional monitored I/O path.

## 4 Locating Reclaimable Physical Pages

The first step in the attack is to locate reclaimable pages of interest, especially those backing sensitive files such as `/etc/passwd`, `/etc/shadow`, and login logs. We evaluated three approaches.

4.1 Method 1: Full Memory Scan via SPARSEMEM + VMEMMAP

The most straightforward idea is to walk every `struct page` described by the memory model:

1) Enumerate every physical frame.
2) Filter out kernel-only pages: slab/SLUB objects, kernel stacks, `vmalloc` regions, and so on.
3) Distinguish file-backed pages from anonymous pages.
4) Identify which of them are reclaimable and potentially interesting.

In practice, this "brute-force scan" is severely limited:

1) Modern servers have enormous numbers of pages; traversing all `struct page`s is extremely expensive.
2) There is no global lock protecting the entire enumeration, so page state can change during the scan, introducing races.
3) The filtering logic is complex, error-prone, and fragile.

In our experiments, this approach proved too slow and too unstable to be a viable attack primitive.

4.2 Method 2: Page-Cache APIs and Direct XArray Traversal

A more structured approach is to reuse high-level page-cache APIs or traverse XArray trees directly.

Using page-cache APIs:

1) Functions such as `pagecache_get_page()` require prior knowledge of the target `address_space` and offset index.
2) Calling these APIs directly from a kernel module is easy to spot in stack-trace-based monitoring and is an obvious signature for EDR/VMM.

Direct XArray traversal:

1) There is no global XArray; each inode has its own private `i_pages` tree.
2) The system may have a huge number of inodes; scanning all of them to build a global view is costly.
3) XArray is also used by `shmem`, DAX, and device-backed memory (for example, PMEM, GPU VRAM), injecting significant noise from outside the usual buddy-allocator paths.

Overall, method 2 is cumbersome, noisy, and slow, and most of its APIs are precisely where defenders like to place hooks.

4.3 Method 3: Leveraging LRU/MGLRU Directly

The third method exploits the fact that LRU/MGLRU already maintains a noise-filtered set of reclaimable pages under well-defined locking.

**Advantages include:**

1) No need to scan all `struct page`s.
2) No need to traverse every inode and its page-cache tree.
3) LRU/MGLRU lists already contain the pages we care about, including reclaimable file-backed pages.

In Linux 6.x systems with MGLRU enabled (now common), this method is particularly attractive, because hot, frequently used file pages are retained in memory longer, giving attackers a stable target and a well-structured map of physical pages.

Page Phantoms adopts method 3 as its foundation.

## 5 The Opportunity Created by MGLRU

5.1 Limitations of Classic LRU

From an attacker’s perspective, classic LRU has two major drawbacks.

1) Short-term view and page thrashing. Newly loaded pages enter the active list; after a period of inactivity they may be misclassified as cold and moved to the inactive list; under reclamation they are quickly evicted. For programs that are quiet at first and later become busy, important pages are likely to be discarded

just before they are needed again, causing page thrashing and making target pages short-lived.

2) Bias against file pages. Code analysis of `mm/vmscan.c` reveals that, in cache-trim mode, reclaim focuses exclusively on file pages and largely ignores anonymous pages; even in normal mode, file pages tend to be favored as victims. As a result, file cache is discarded aggressively even when memory pressure is not truly severe. This hurts performance and makes target pages in the file cache even shorter-lived.

In short, under classic LRU, an attacker must continually race the reclaim thread to locate and operate on target pages before they are evicted.

5.2 MGLRU as a High-Precision Hunting Ground

MGLRU was designed to improve performance, not to weaken security. Nevertheless, its design yields two side effects that Page Phantoms exploits.

1) Stable targets. Frequently accessed files—such as `/etc/passwd` and `/etc/shadow`—are preserved in younger generations for longer periods. Their physical pages stay in the cache longer and more predictably, giving attackers a comfortable time window for locating and manipulating them.
2) A structured “memory map”. MGLRU’s generational lists are essentially a well-organized map of reclaimable pages. By traversing these lists, an attacker can systematically inspect candidate pages and pinpoint the exact `struct page` that backs a chosen file.

We stress that MGLRU is not itself a vulnerability. It is a legitimate performance optimization. Page Phantoms simply uses it as a high-precision hunting tool.

## 6 Attack Overview

Conceptually, Page Phantoms abandons the left-hand “monitored I/O path” (system calls → VFS → filesystem → disk I/O) and instead targets the right-hand “silent lifecycle” of reclaimable physical pages:

1) File accesses populate the page cache with file-backed pages.
2) LRU/MGLRU manages these pages as they age and are eventually written back or dropped.
3) Production VMM monitoring typically focuses on static or semi-static regions: kernel code and data segments, task structures, and inodes. Highly dynamic page-cache pages are difficult to baseline and are rarely monitored at the same granularity.

Page Phantoms leverages this structural blind spot via MGLRU + XArray:

1) Use MGLRU’s generational lists to locate the physical page that backs a target file.
2) Isolate the page from reclaim and the buddy allocator, freezing its state.

3) Modify the page content while carefully manipulating flags to avoid triggering write-back.
4) Remove the page from LRU/MGLRU and the page-cache tree, and finally return it to the buddy allocator as a normal free page.

Externally, the page appears to have completed a normal cache lifecycle; no suspicious system calls or inode changes are observed.

**We divide the attack into two variants:**

1) Phases 1–4: file-system-based Zero-I/O (in-memory only).
2) Phase 5: memory-page-based Zero-I/O with disk persistence.

## 7 Phases 1–4: File-System-Based Zero-I/O

### 7.1 Phase 1 – Locating the Target Page

Objective: identify, purely via in-memory traversal, the `struct page` (folio) that caches a specific file, for example `/etc/passwd`.
Traversal path: starting from NUMA and memory-cgroup structures, we traverse `pg_data_t → mem_cgroup → lruvec → lru_gen_folio → struct page`.

For each candidate page we:

1) Check whether it is file-backed and resident in the page cache.
2) Follow `page->mapping->host` to obtain the owning inode.
3) Use the dentry cache to resolve the full pathname.
4) Compare the resolved path (string) to the target path.

Concurrency challenges include:

1) AB-BA deadlock risk. Scanning MGLRU requires locking candidate pages (`page_lock`). Path resolution requires the dentry lock (`dentry_lock`). The normal VFS lock order is “take `dentry_lock` first, then `page_lock`”. A naïve scan that locks pages first and then acquires dentry locks reverses this order, creating a classic AB-BA deadlock scenario across CPUs.
2) Soft-lockup risk. `lru_lock` is a global spinlock shared by all CPUs. If we hold `lru_lock` while performing expensive path resolution, other CPUs busy-wait, triggering watchdog warnings and apparent system freezes.

To avoid these pitfalls, Phase 1 is split into three stages.

Stage 1 – Collect candidates: acquire `lru_lock` only briefly, quickly traverse MGLRU lists and record references to candidate pages, release `lru_lock` immediately, and periodically yield the CPU to avoid long critical sections.

Stage 2 – Resolve paths: with no global locks held, iterate over the collected pages, perform path resolution via the dentry cache (calling potentially blocking functions as needed), and compare each resolved path against the target pathname.

Stage 3 – Isolate target page: once a candidate is confirmed as the target page, lock that page and mark it for isolation. Subsequent phases operate solely on this isolated page.

This design avoids AB-BA deadlocks and spinlock-induced soft-lockups, while still providing a deterministic way to identify the target page.

### 7.2 Phase 2 – Isolating the Target Page from Reclaim

The second phase removes the target page from normal reclaim and I/O paths. The key steps are:

1) Call `lock_page(page)` to freeze the page's state.
2) Call `get_page(page)` to increment its reference count and prevent freeing.
3) Call `SetPageReserved(page)` (or an equivalent helper) to hide the page from the buddy allocator and remove it from reclaim candidates.

After this, the page is effectively captured by the attacker. Reclaim threads will no longer select it, and to the buddy allocator it resembles a special reserved page (similar to pages pinned for DMA).

### 7.3 Phase 3 – Modifying the Page

Before modifying the page, we must ensure that no process holds a vulnerable mapping that might cause crashes.
Preparation: use routines such as `try_to_unmap()` to remove existing page-table mappings. Subsequent accesses will fault and reload from disk instead of using the old page.
Two modification options are available.

**Option A – Remap into user space:**

1) Use `remap_pfn_range` to map the physical frame into a chosen process's virtual address space.
2) Modify contents from user mode.
3) Multiple virtual mappings to the same physical frame are common in Linux (for example, DMA zero-copy).
4) If direct use of `remap_pfn_range` is considered too visible, the mapping can be built manually using lower-level page-table operations.

**Option B – Pure kernel-space modification:**

1) Do not create any user-space mapping.
2) Modify the page directly from kernel code.
3) This option is used, for example, when sanitizing login logs entirely within the kernel.

Both options bypass the entire storage stack; no monitored I/O APIs are invoked.

### 7.4 Phase 4 – Clean-Up and Logical Rollback

After tampering, we must erase all traces of the page from LRU/MGLRU, accounting, and the page cache, and then hand it back to the buddy allocator.
Step 1 – Remove from LRU/MGLRU:
1) `list_del(&folio->lru)` – unlink the folio from the global MGLRU list to hide it from reclaim scanners.
2) Adjust generation and zone counters, for example `GEN_INDEX/ZONE -= folio_nr_pages(folio)`.
3) Update per-CPU VM statistics (`NR_FILE_PAGES`, `NR_INACTIVE_FILE`) to reflect the reduced number of file pages.
4) Call `__ClearPageLRU(folio)` to clear the LRU flag and officially remove the page from LRU/MGLRU management.

Step 2 – Detach from the page cache:
1) `xa_erase(mapping->i_pages, index)` – remove the folio from the file's XArray tree and logically disconnect file and physical page.
2) `inode_add_lru(inode)` – return now-redundant metadata slab objects to the slab shrinker for later reclamation.
3) `ClearPageDirty(folio)` – ensure the page is no longer considered dirty, preventing any write-back of tampered contents.

Step 3 – Return to the buddy allocator:
1) `ClearPageReserved(folio)` – clear the reserved flag so the page becomes visible again to the buddy allocator.
2) `put_page(folio)` – drop the final reference, allowing the allocator to reclaim and reuse the frame.

After these steps, LRU/MGLRU and all VM statistics are internally consistent; the page is no longer associated with any file or write-back; and the tampered data existed only transiently in memory and has been absorbed back into the normal pool of free pages. Phases 1–4 therefore implement a complete file-system-based Zero-I/O attack: all operations occur within the page cache and MGLRU, without traversing the traditional storage stack.

## 8 Phase 5: Memory-Page-Based Zero-I/O with Disk Persistence

In many scenarios, in-memory tampering is insufficient. Attackers often want to modify the actual on-disk content—for example, to erase or forge login records, audit logs, or configuration files.

The core difficulty is:
1) Persisting data requires disk I/O.
2) The normal page-cache write-back path depends on well-known APIs that are heavily monitored.

### 8.1 Normal Page-Cache Write-Back

Under standard operation, flushing a dirty page from the page cache to disk involves three mandatory steps:

1) `SetPageDirty(page)` – mark the data page as dirty.
2) `mark_inode_dirty(inode)` – synchronize inode metadata with the page's state.
3) `filemap_write_and_wait_range(mapping, ...)` – invoke the filesystem to write the page to disk.

These steps cannot be omitted without violating filesystem invariants. Any attempt to persist attacker-controlled data via this route necessarily uses these APIs and is therefore observable.

### 8.2 XArray Entry Replacement

To evade such monitoring, Page Phantoms manipulates the page-cache XArray directly.

High-level procedure:

1) Allocate a new physical page (`new_folio`) from the buddy allocator.
2) Forge it into a legitimate file-backed cache page by:
   a) Copying the `mapping` and `index` fields from an existing page;
   b) Marking it uptodate and, later, dirty;
   c) Attaching appropriate block-device metadata.
3) Use XArray operations to replace the entry at the chosen index in `mapping->i_pages` so that it points to `new_folio` instead of the old page.
4) Let the kernel's existing reclaim and write-back threads treat this page as a normal dirty file page and flush it to disk.

From the kernel's perspective, this is indistinguishable from a regular dirty cache page that happened to appear in memory; all disk I/O is generated by pre-existing kernel threads.

### 8.3 Technical Challenges

#### 8.3.1 Issue 1 – Identifying the Replacement Target

For many files, particularly logs, we are interested in the last page, which typically contains the most recent and security-relevant entries.

1) The `address_space`'s `->i_pages` XArray holds all resident pages for the file.
2) Using XArray iteration primitives such as `xas_for_each()`, we traverse from index 0 to $ULONG\_MAX$, track the maximum index observed, and treat the folio at that index as the tail page.
3) Replacing this tail page allows us to surgically overwrite the most recent segment of the log.

#### 8.3.2 Issue 2 – Making the New Page Look Like a Real File-Backed Page

A newly allocated page must satisfy several conditions to be accepted as a legitimate cache page:

1) Copy the `mapping` pointer and `index` from the old folio to `new_folio`, so it appears to belong to the same file and offset.

2) Call a helper such as `folio_mark_uptodate(new_folio)` to set the uptodate flag.
3) Filesystems (for example, ext4) expect pages to be marked uptodate before being dirtied; otherwise they emit warnings and may refuse write-back.

8.3.3 Issue 3 – Attaching `struct buffer_head` Metadata

For block-based filesystems, dirty pages are expected to have associated `struct buffer_head` descriptors:

1) If a dirty page has no attached buffer heads, filesystems may log warnings such as `mpage_prepare_extent_to_map`.
2) To avoid this, we compute the block size via `i_blocksize(mapping->host)`, call `create_empty_buffers(new_folio, blocksize, 0)` to attach empty buffer heads to `new_folio`, and finally call `folio_mark_dirty(new_folio)` (or an equivalent helper) to mark the page dirty.

Once these steps are complete, `new_folio` is a fully valid file-backed cache page from both filesystem and block-layer perspectives.

8.4 Executing the Replacement

The actual replacement proceeds as follows

1) Initialize an XArray operation: set `xas.xa = &mapping->i_pages` and `xas.xa_index = offset`.
2) Acquire the appropriate lock via `xas_lock_irq(&xas)`.
3) Call `xas_store(&xas, new_folio)` to replace the XArray entry.
4) Set `old_folio->mapping = NULL` to sever the link between the old page and the file's address space.
5) Adjust VM statistics (for example, `NR_FILE_PAGES`) for both old and new pages.
6) Release the lock with `xas_unlock_irq(&xas)`.
7) Drop the old page's reference with `folio_put(old_folio)`, allowing it to be reclaimed as a normal free page.

Afterwards, the file's page-cache tree maps the selected index to `new_folio`, and the old page is a regular, unreferenced page in the buddy allocator.

8.5 Resulting Behavior

From the page-cache perspective:

1) It appears as though a new page has been paged in at the chosen offset, containing slightly different data.
2) The old page has been safely reclaimed.

From the write-back subsystem:

1) `new_folio` is an ordinary dirty file page.
2) Background or direct reclaim threads will eventually pick it up and flush it to disk as part of normal operation.

Crucially:

1) We never modified the original physical page in place; we simulated a page-in and let the kernel's own mechanisms handle persistence.
2) We did not invoke monitored page-cache write-back APIs such as `filemap_write_and_wait_range`.
3) We did not directly tamper with inodes or other structures under host-side hashing.

This constitutes a memory-page-based Zero-I/O attack with disk persistence: externally indistinguishable from standard cache behavior, internally performing a controlled, covert disk modification.

## 9 Implementation Overview and Evaluation Context

To validate Page Phantoms, we implemented a prototype in a tightly controlled dual-layer defense environment.

On the host side:

1) A QEMU/KVM-based VMM extension monitors guest inode structures at GPA granularity.
2) It periodically hashes these regions and raises alarms on deviations.

On the guest side:

1) A custom EDR kernel module hooks VFS entry points and page-cache APIs.
2) Any use of traditional I/O paths or standard page-cache write-back routines is logged and can be reported to the host.

We use `/etc/passwd`, `/etc/shadow`, and various login logs as target files to demonstrate both in-memory and on-disk tampering. In this environment, Page Phantoms is able to modify these files' contents in memory and on disk; avoid triggering monitored system calls and page-cache APIs; keep kernel statistics and reclaim behavior internally consistent; and operate reliably under realistic server workloads using only exported, stable kernel symbols. Even under this stricter-than-typical setup, both the host VMM and guest EDR fail to observe direct signs of the tampering.

## 10 Discussion and Limitations

### 10.1 Dependence on Cache Residency

The techniques described require that the target file have at least one page resident in the page cache. In practice, even lightly loaded systems touch their own configuration and state files more often than intuition suggests, so this condition often holds naturally. When it does not, the attacker can simply wait and poll: periodically scanning MGLRU lists at a reasonable interval is frequently more effective and less invasive than trying to hook every possible trigger event.

### 10.2 Concurrency and Stability

Operating on live kernels requires careful handling of locks, reference counts, and accounting. Our three-stage, lock-aware design for Phase 1 avoids AB-BA deadlocks and spinlock-induced soft-lockups, but it adds implementation complexity. Incorrect sequencing could easily lead to deadlocks, data-structure corruption, or watchdog timeouts. Despite this, a well-engineered implementation can remain stable on high-load systems.

### 10.3 Scope of Applicability

The current techniques target Linux 6.x kernels with MGLRU enabled and conventional block-based filesystems. Other operating systems have analogous page-cache and reclaim mechanisms, but porting Page Phantoms would require substantial reverse-engineering to identify comparable structures and invariants.

## 11 Defensive Considerations and Future Work

### 11.1 Potential Defense Directions

A natural defensive idea is to monitor transitions between address spaces and physical pages, for example by:

1) Tracking changes to each page's `mapping` and `index` fields;
2) Correlating XArray membership changes with LRU/MGLRU state transitions;
3) Flagging suspicious patterns of XArray replacement or anomalous "forged" pages.

However, such fine-grained monitoring is extremely expensive. Applying it to all pages, or even a large subset, in a production environment with high memory churn would incur prohibitive overhead. At present, this level of instrumentation is difficult to justify outside highly constrained, high-security deployments.

### 11.2 Future Research Directions

Future work can explore several avenues:

1) Device passthrough and IOMMU. Investigate how device passthrough, NUMA layouts, and IOMMU mappings interact with page-lifecycle attacks and what additional attack or defense surfaces they introduce.
2) Confidential computing and protected memory. Study how hardware-based confidential-computing technologies (for example, AMD SEV, Intel TDX) affect both feasibility and detectability of Page-Phantoms-style attacks, and what defensive mechanisms are realistic in such environments.
3) Non-Linux platforms. Adapt the fundamental idea—attacking the lifecycle of reclaimable pages rather than the I/O path—to non-Linux operating systems, after reverse-engineering their page-cache and reclaim subsystems.

## 12 Conclusion

Page Phantoms shows that even in environments with “god-mode” VMI and rigorous in-guest monitoring, attackers can still bypass defenses by shifting focus from the I/O path to the lifecycle of reclaimable physical pages.

By analyzing the interaction between the Linux page cache, XArray, and MGLRU, we demonstrate how to use MGLRU as a high-precision hunting ground to locate file-backed pages of interest; implement file-system-based Zero-I/O attacks that perform complete locate–isolate–modify–cleanup cycles entirely within page-cache and MGLRU data structures, without invoking traditional I/O paths; and achieve memory-page-based Zero-I/O persistence by replacing XArray entries and leveraging the kernel’s own write-back machinery to commit forged pages to disk, all while sidestepping the APIs most commonly monitored by host VMMs and guest EDRs.

These results highlight a structural blind spot in current defenses: the management of reclaimable pages. Robust hardening in virtualized environments must consider this “silent path” and find ways—within acceptable performance budgets—to constrain or observe the lifecycle of file-backed pages that are otherwise invisible to traditional I/O-centric monitoring.