# A brief primer on Persistent Memory Objects


Derrick Greenspan Naveed Ul. Mustafa Zoran Kolega Mark Heinrich Yan Solihin

University of Central Florida, Orlando, Florida

{derrick.greenspan, unknown.naveedulmustafa, heinrich, yan.solihin}@ucf.edu

kolegazoran@knights.ucf.edu


## Introduction to Persistent Memory (PM)

- **Byte Addressable**
- **Non-Volatile/Persistent**
- **High density/cheaper per byte vs. volatile memory (i.e., DRAM)**
- **Performance much closer to volatile memory vs. block storage (i.e., SSD)**
- **Could augment or replace volatile memory as main memory**

## PM Management

### Memory-Mapped Files

- Organize PM as a file system
- Limits the use of system calls
- Keeps data as an array of bytes rather than pointers
- Requires keeping two systems (file system and virtual memory) consistent despite distinct semantics

### Persistent Memory Objects (PMOs)

- Organize PM as a collection of objects (PMOs) holding pointer-rich data structures
- No file backing
- More intuitive design

## PMO System Calls

| Primitive | Description |
|---|---|
| attach(name,perm,key) | Render accessible the PMO `name`, given a valid `key` with permissions `perm`, and return a pointer to the start of the PMO. |
| detach(addr) | Render inaccessible the PMO `addr` points to. |
| psync(addr) | Force modifications to the PMO associated with `addr` to be durable. |
| pcreate(name,size,key) | Create a PMO `name` of `size` and `key`. |
| pdestroy(name,key) | Given a valid `key`, delete PMO `name` and reclaim its space. |

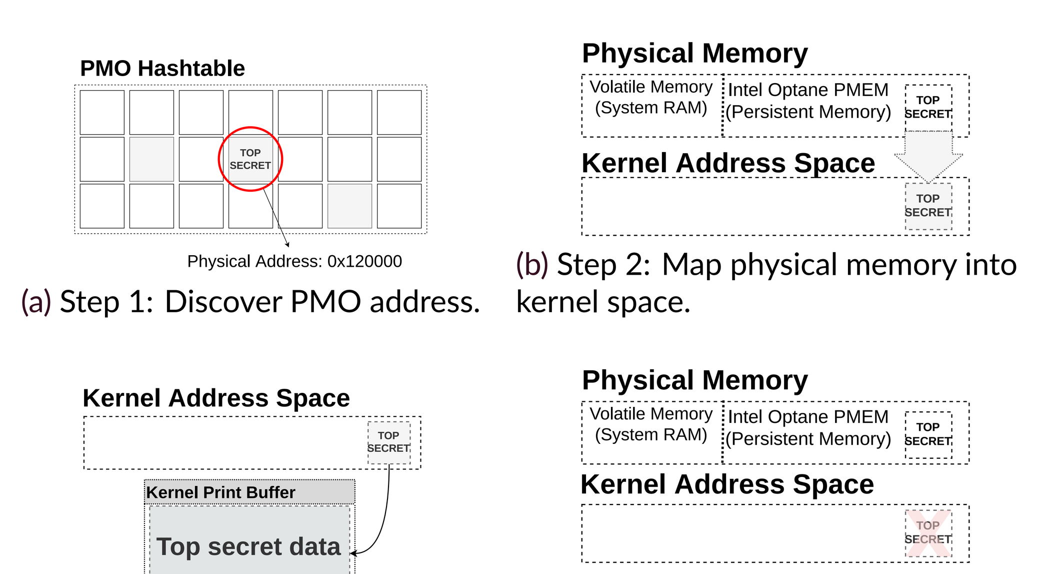
## Threat Model

### Assumptions

- PMO is not attached to any user process (i.e., "at-rest")
- PMO-resident data structures may contain buffers/pointers
- Only the Kernel Crypto API, `memcpy`/`memset`, and PMO subsystem are assumed free of vulnerabilities
- Attacker knows location of PMO in system

### Goal of Attacker

- Disclose private data held in at-rest PMO (Figure 1)
- Overwrite data held in at-rest PMO with malicious data




(a) Step 1: Discover PMO address. (b) Step 2: Map physical memory into kernel space. (c) Step 3: Read mapped data. (d) Step 4: Unmap kernel space memory.

Figure 1. Steps of PMO example attack.

## Basic PMO Design Principles

### Fast Access

- Simple PMO System layout
  - PMO is contiguous region of memory
  - Metadata entries located at start of PM and store state information
  - Data can be accessed by adding given offset to base-address of PMO
  - Obviates the need for pointer chasing
- Low-latency attach/detach calls
  - Use *demand paging* to only map required pages at fault time
  - Only change permission of faulted pages at detach time (rather than unmap them entirely)
- Low-latency Pointer Dereference
  - Use *static pointers* that point directly to unique PM addresses
  - Split address space in half so that virtual addresses with MSB of $1$ reference PMOs

### Crash Consistent

- Data are consistent even after crash
- Create shadow copy at attach
- Render changes crash consistent via `psync`
- Copy modified shadow pages $\rightarrow$primary pages

## PMO state transitions

- PMO recovery depends on the PMO state, as illustrated in Figure 2
- Invariant: At least one of the primary or shadow copy are always valid
  - Recovery restores from that copy based on state

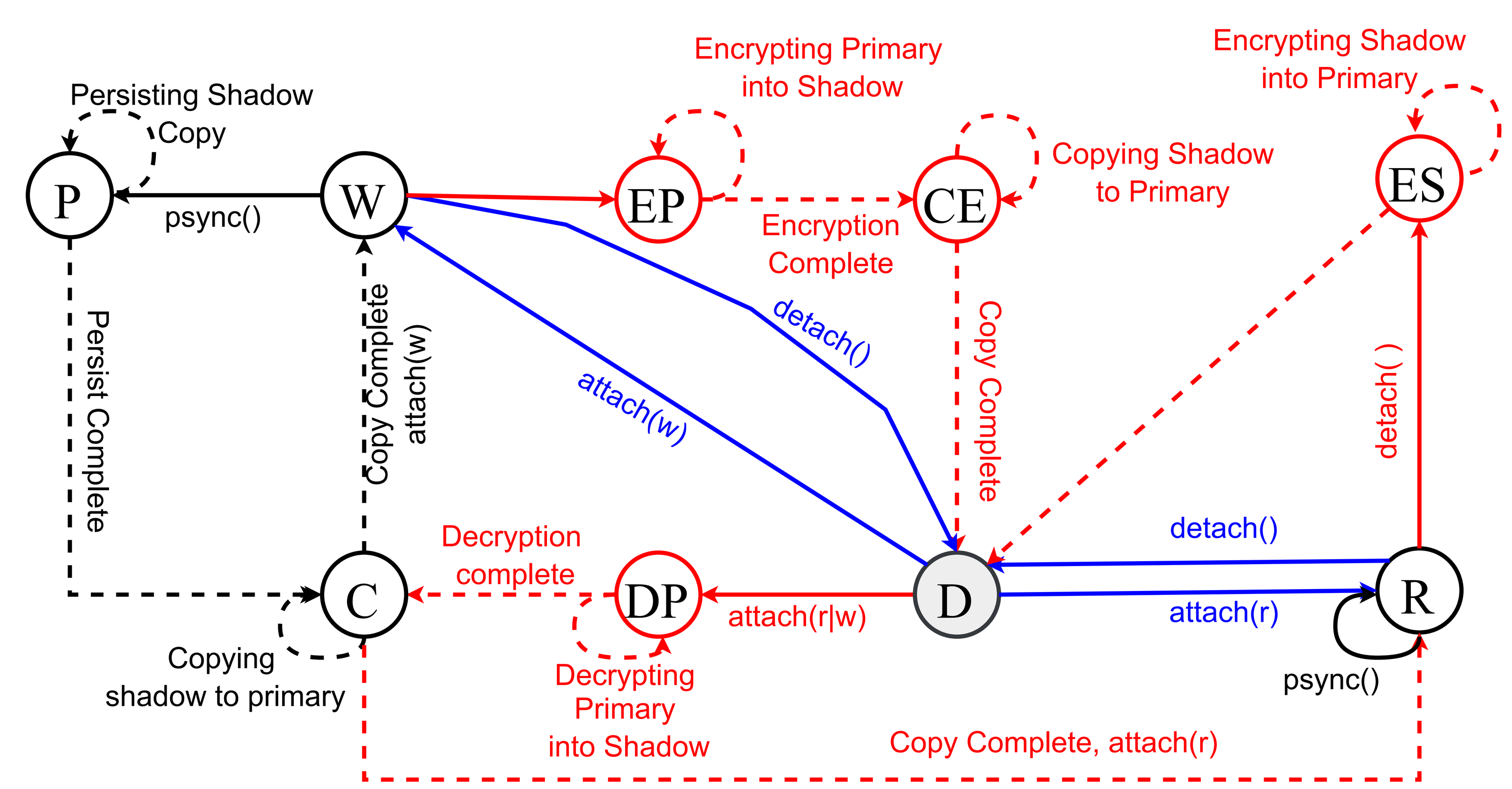



Figure 2. PMO state transitions. Dashed are for the crash consistent design without encryption. Dotted are for the crash consistent design with encryption. Solid are for both.

## Security Protection For At-Rest PMOs

### Protection from Corruption

- Compute checksum over PMO at detach time
- Store checksum in associated PMO metadata entry
- Compare computed checksum at attach with stored checksum, attach fails if different

### Protection from Disclosure

- Use Kernel Crypto API to decrypt PMO when in use (i.e., attached)
- Use kernel Crypto API to encrypt PMO when at rest (i.e., detached)
- Do not store encryption key alongside PMO; key is provided by user at attach
- Encryption/Decryption not atomic, so must add new states (see 2)
- Never perform encryption in place

## Evaluation Methodology

### Evaluated Designs

- No Crash Consistency (ext4-dax)
- State-of-the-art crash consistent filesystem (Nova-Fortis)
- Persistent Memory Object System (PMO System)

### Evaluated Benchmarks

- Microbenchmarks
  - LU Decomposition (LU) - $3584 \times 3584$ doubles
  - 2D-Convolution (2DConv) - $4096 \times 128$ integers
  - Tiled Matrix Multiplication (TMM) - $3072 \times 3072$ integers
- FileBench benchmarks
  - Representations of real-world applications
  - File Server (FS), Web Server (WS), Web Proxy (WP), Var Mail (VM)

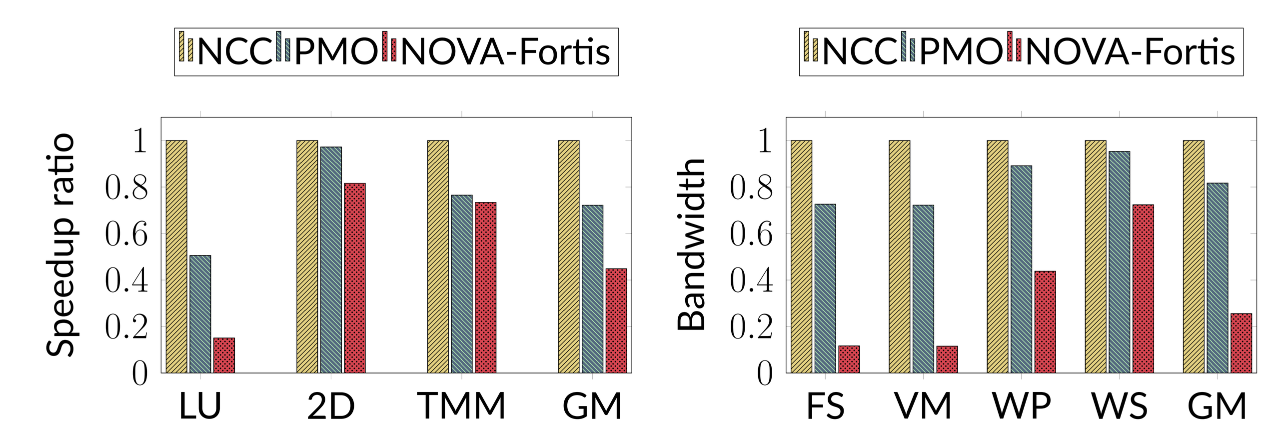
## Results

### Microbenchmarks

- PMOs are only $27.8\%$ slower than a system without crash consistency
- PMOs are $1.61\times$ faster than Nova-Fortis

### Filebench

- PMOs have $18.3\%$ lower bandwidth than a system without crash consistency
- PMOs have $3.2\times$ higher bandwidth over NOVA-Fortis




(a) Microbenchmark performance results (b) Filebench bandwidth results

### Integrity and Encryption

- Encryption lowers bandwidth by $41\%$ on average
- Integrity Checking alone lowers bandwidth by $3\%$ on average
- Both Encryption and Integrity Checking lowers bandwidth by $46\%$ on average

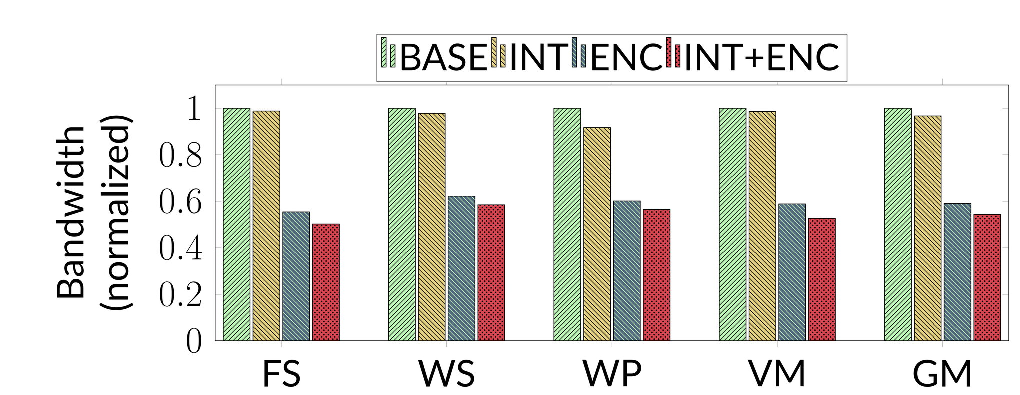



Figure 4. Bandwidth comparison of attach/detach PMO, with different modes: baseline (BASE), Integrity (INT), Encryption (ENC), and both (ENC+INT).

## Acknowledgements


This work is supported in part by the Office of Naval Research (ONR) under grant N00014-20-1-2750, and the National Science Foundation (NSF) under grant 1900724.