Recovering Write-Protected NVMe SSDs Through USB Bridge XRAM Injection: Bypassing the ASMedia ASM2362 Firmware Opcode Whitelist

J. Sullivan
2026

***Abstract*—Consumer NVMe solid-state drives can enter a permanent firmware-level read-only mode after power loss during Windows hibernation, silently discarding all write operations while reporting success. Recovery is complicated when the drive is connected through a USB-to-NVMe bridge such as the ASMedia ASM2362, whose firmware implements an opcode whitelist that blocks the NVMe administrative commands required for recovery—Format NVM, Sanitize, Security Send, and Set Features—while allowing only Identify and Get Log Page to pass through. We present a novel technique that bypasses this whitelist by writing NVMe Submission Queue entries directly into the bridge controller's internal XRAM via vendor SCSI commands (0xE4/0xE5), then ringing the NVMe Admin Submission Queue doorbell through the bridge's PCIe Transaction Layer Packet (TLP) engine. Using this method, we successfully recovered a Phison PS5012-E12 based Silicon Power 256 GB SSD from firmware write protection by injecting a Sanitize Block Erase command. Our open-source tool, implemented in approximately 5,400 lines of Zig, demonstrates that USB bridge XRAM constitutes a viable attack surface for NVMe drive recovery without professional tools or native M.2 PCIe connection.**

***Index Terms*—NVMe, SSD recovery, USB bridge, ASMedia ASM2362, XRAM injection, firmware write protection, SCSI passthrough, Phison PS5012-E12**

I. Introduction

Solid-state drives based on the NVMe (Non-Volatile Memory Express) protocol have become the dominant storage medium in modern computers. Their Flash Translation Layer (FTL) firmware manages the mapping between logical block addresses and physical NAND flash pages, maintaining this mapping in volatile SRAM backed by a capacitor-protected write to NAND on power loss. When this protective mechanism fails—as can occur during an abrupt power loss while Windows hibernation is writing its memory image—the FTL mapping tables may become corrupted, causing the controller firmware to place the drive into a permanent read-only mode as a data-protection measure [1].

This paper documents the investigation and successful recovery of a Silicon Power 256 GB NVMe SSD (Phison PS5012-E12 controller) that entered this failure mode. The drive exhibited a subtle and pernicious symptom: *silent write failure*. Commands such as `dd` and `wipefs` reported success, but subsequent reads revealed that no data had actually been written. Standard Linux write-protection checks (`blockdev -getro`, SCSI mode page WP bit) all reported the drive as writable.

The drive was connected to a Linux host through an ASMedia ASM2362 USB-to-NVMe bridge, which presents the NVMe device as a SCSI block device (`/dev/sdX`). The bridge provides a vendor-specific SCSI Command Descriptor Block (CDB) with opcode `0xe6` for tunneling NVMe admin commands over USB. However, as we discovered through firmware reverse engineering by cyrozap [2] and confirmed through empirical testing, the bridge firmware maintains an internal opcode whitelist that permits only two NVMe admin commands—Identify (`0x06`) and Get Log Page (`0x02`)—to pass through. All recovery-relevant commands (Format NVM, Sanitize, Security Send/Receive, Set/Get Features) are silently dropped: the SCSI layer returns success, but the commands never reach the NVMe controller.

Our contribution is a novel XRAM injection technique that bypasses this whitelist entirely. The ASM2362 exposes three additional vendor SCSI commands (`0xE4`, `0xE5`, `0xE8`) that provide direct read/write access to the bridge chip's 64 KB internal XRAM. The NVMe Admin Submission Queue resides within this XRAM at addresses `0xB000`–`0xB1FF`. By writing a 64-byte NVMe Submission Queue entry directly into this memory and ringing the doorbell via the bridge's PCIe TLP engine, we can submit arbitrary NVMe admin commands without firmware intervention. Using this method, we successfully executed a Sanitize Block Erase that restored full write capability to the drive.

The tool is open source, implemented in Zig [3], and requires only a Linux host with root privileges and a USB connection to an ASM236x-based NVMe enclosure. No professional recovery hardware (PC-3000 [4]), native M.2 PCIe slot, or Windows operating system is needed.

II. Background

A. NVMe Architecture

The NVMe specification [5] defines a register-level interface between host software and NVMe controllers over PCIe. Commands are submitted through *Submission Queues* (SQs) and completions returned through *Completion Queues* (CQs). Each queue pair is managed through *doorbell registers* mapped into the controller's PCIe Base Address Register 0 (BAR0) memory space.

A Submission Queue Entry (SQE) is a 64-byte structure organized as sixteen 32-bit dwords (all little-endian):

- **CDW0** (bytes 0–3): Opcode (bits 7:0), fused operation (bits 9:8), PRP/SGL descriptor type (bits 15:14), Command ID (bits 31:16)
- **NSID** (bytes 4–7): Namespace Identifier
- **Reserved/MPTR** (bytes 8–23): Metadata pointer
- **PRP1/PRP2** (bytes 24–39): Physical Region Page pointers for data transfer
- **CDW10–CDW15** (bytes 40–63): Command-specific dwords

The Admin Submission Queue (ASQ) is queue ID 0 and handles administrative commands including Identify, Format NVM, Sanitize, and Set/Get Features. Its base address and size are configured through the Admin Queue Attributes (AQA) and Admin Submission Queue Base Address (ASQ) registers in BAR0. The host submits commands by writing SQEs into the SQ and then writing the updated tail index to the corresponding doorbell register at BAR0 + 0x1000.

A Completion Queue Entry (CQE) is a 16-byte structure containing a command-specific result (DW0–DW1), the SQ Head Pointer (SQHD), SQ Identifier (SQID), Command Identifier (CID), and a status field encoding Phase bit, Status Code (SC), Status Code Type (SCT), and Do Not Retry (DNR) flag.

The NVMe SMART/Health Information log page (Log Page ID `0x02`) [5] includes a Critical Warning byte. Bit 3 of this field, when set, indicates that the media has been placed in read-only mode due to internal data reliability concerns. This bit reflects autonomous firmware state and cannot be cleared by standard host commands—it is not a writable register.

Two administrative commands are relevant to recovery from write protection:

- **Format NVM** (opcode `0x80`): Reformats a namespace, optionally performing a secure erase. Operates *per-namespace* and relies on the existing FTL for logical-to-physical mapping. CDW10 encodes LBA Format (LBAF, bits 3:0) and Secure Erase Setting (SES, bits 11:9).
- **Sanitize** (opcode `0x84`): Erases all user data from the controller. Operates at the *controller level* across all namespaces. CDW10 encodes Sanitize Action (SANACT, bits 2:0): 1 = Exit Failure Mode, 2 = Block Erase, 3 = Overwrite, 4 = Crypto Erase. The Sanitize Capabilities (SANICAP) field in the Identify Controller data reports which actions are supported.

The Optional Admin Command Support (OACS) field in the Identify Controller data structure indicates which optional admin commands the controller supports: bit 0 for Security Send/Receive, bit 1 for Format NVM, bit 2 for Firmware Commit/Download.

B. USB-NVMe Bridge Architecture

The ASMedia ASM2362 is a USB 3.1 Gen 2 (10 Gbps) to PCIe Gen3 x2 bridge controller commonly used in consumer NVMe enclosures. It is built around an 8051-compatible CPU core running at approximately 114.3 MHz with 64 KB of internal XRAM (external data memory in 8051 terminology) and an SPI flash interface for firmware storage [2]. The bridge firmware has no signature verification, meaning arbitrary code can theoretically be loaded [2].

The bridge presents the NVMe device to the USB host as a USB Mass Storage device using the SCSI protocol. It translates standard SCSI commands (READ(10), WRITE(10), INQUIRY, TEST UNIT READY) into NVMe I/O commands transparently. For NVMe administrative commands, it provides a vendor-specific 16-byte CDB with opcode `0xe6`:

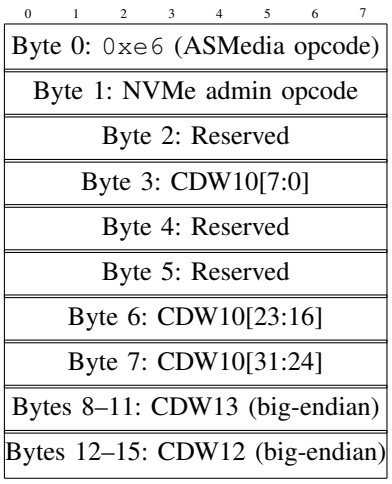

Fig. 1. ASM2362 `0xe6` passthrough CDB format (16 bytes). Note that CDW10 bits [15:8] are not mapped—they fall in reserved bytes 4–5. This causes some NVMe command fields (e.g., the SEL field in Get Features, which occupies CDW10[10:8]) to be silently lost through the bridge.

Beyond the `0xe6` passthrough, the bridge exposes several other vendor SCSI commands, documented through firmware reverse engineering [2]:

TABLE I
ASM2362 VENDOR SCSI COMMANDS

Opcode	Command	Direction	Purpose
`0xE0`	Config Read	From Dev	Bridge config
`0xE1`	Config Write	To Dev	Bridge config
`0xE2`	Flash Read	From Dev	SPI flash access
`0xE3`	Firmware Write	To Dev	Flash firmware
`0xE4`	XDATA Read	From Dev	Read XRAM
`0xE5`	XDATA Write	None*	Write XRAM byte
`0xE6`	NVMe Admin	Varies	NVMe passthrough
`0xE8`	Reset	None	CPU/PCIe reset

*Value embedded in CDB byte 1; no SCSI data transfer phase.

The `0xE4` and `0xE5` commands are of particular interest for this work: they provide direct access to the bridge controller's XRAM address space, bypassing all firmware-level command filtering. The `0xE4` command reads 1–255 bytes from a 16-bit XRAM address; the `0xE5` command writes a single byte (carried in the CDB itself, not in a data phase) to a 16-bit XRAM address.

C. Related Work

cyrozap/usb-to-pcie-re [2] provides the foundational reverse engineering of ASMedia USB bridge firmware, documenting the XRAM memory map, vendor SCSI commands,

PCIe TLP engine registers, and the 8051 CPU architecture. This work was essential to our approach but does not address NVMe drive recovery use cases.

smartmontools [6] implements an ASMedia bridge driver (`sntasmedia_device` class in `scsinvme.cpp`) that uses the `0xe6` CDB for reading SMART data and Identify information. It does not attempt recovery commands and implicitly acknowledges the opcode whitelist by implementing only the two permitted opcodes.

ASMTool [7] by smx-smx is a firmware dumping utility for ASMedia bridges that uses vendor SCSI commands for flash access but does not implement XRAM-based NVMe command injection.

Phison PS5012 Reinitial Tool [8] is a Windows-only vendor utility designed for recovery of Phison-based SSDs in read-only mode. It supports ASM2362 bridges but requires Windows with specific firmware files matched to the NAND configuration. Our approach achieves similar results from Linux without vendor tools.

PC-3000 SSD [4] by ACE Lab is a professional data recovery platform that supports Phison controllers including the PS5012-E12. It can inject firmware loaders to SRAM, bypass corrupted FTL mappings, and rebuild translation tables [9]. At $300–1500 per recovery, our technique provides a free alternative when data preservation is not required.

sg3_utils [10] provides userspace SCSI command utilities including `sg_raw` for sending arbitrary CDBs, but does not include ASMedia-specific functionality.

Zheng et al. [1] demonstrated that 13 of 15 consumer SSDs tested lost data under power fault conditions, establishing that FTL corruption from power loss is a widespread problem. Cai et al. [11] provide a comprehensive analysis of flash memory error mechanisms that can compound FTL corruption, including program disturb and read disturb effects.

DiskTuna [12] documents the "write-protected SSD" failure mode from a data recovery perspective, noting that SMART Critical Warning bit 3 and the controller's internal state are distinct mechanisms—a drive can be functionally read-only without bit 3 being set.

III. Problem Analysis

A. Symptom Characterization

The drive exhibited *silent write failure*—write commands completed without error but had no effect on stored data:

```
$ sudo dd if=/dev/zero of=/dev/sdb bs=512 count=1
    \
   conv=fsync oflag=direct
512 bytes copied, 0.000314 s, 1.6 MB/s

$ sudo xxd -l 16 /dev/sdb
00000000: 33c0 8ed0 bc00 7c8e ... # MBR unchanged!
```

Listing 1. Silent write failure: dd reports success but data is unchanged.

The `dd` command reported success and the `fsync` completed without error, yet the MBR boot code remained intact rather than being overwritten with zeros. The `wipefs` utility exhibited identical behavior—reporting erasure of GPT and PMBR signatures at three separate offsets while leaving all signatures byte-for-byte untouched:

```
$ sudo wipefs --all --force /dev/sdb
/dev/sdb: 8 bytes erased at offset 0x200 (gpt)
/dev/sdb: 8 bytes erased at offset 0x3b9e655e00 (
    gpt)
/dev/sdb: 2 bytes erased at offset 0x1fe (PMBR)

$ sudo xxd -s 0x1fe -l 2 /dev/sdb
000001fe: 55aa # MBR signature still present!

$ sudo xxd -s 512 -l 8 /dev/sdb
00000200: 4546 4920 5041 5254 # "EFI PART" still
    there
```

Listing 2. wipefs reports success but signatures remain intact.

Critically, standard write-protection detection mechanisms all reported the drive as writable:

TABLE II
WRITE PROTECTION DETECTION RESULTS

Check	Result	Expected for RO
`blockdev -getro`	0 (writable)	1 (read-only)
SCSI Mode Page WP bit	OFF	ON
`/sys/.../ro` sysfs	0	1
`dd` exit code	0 (success)	Non-zero
`wipefs` exit code	0 (success)	Non-zero

This made the failure mode particularly insidious: without explicitly verifying write operations via readback, there was no indication that the drive was rejecting writes. The NVMe controller accepted write commands for protocol compliance but did not commit them to NAND flash.

B. The Opcode Whitelist

The ASM2362 `0xe6` passthrough mechanism selectively forwards NVMe admin commands based on an internal firmware whitelist. Through a combination of firmware reverse engineering [2], analysis of the smartmontools ASMedia driver [6], and empirical testing with our tool, we determined that only two of eight observed NVMe admin opcodes are actually forwarded to the NVMe controller:

TABLE III
ASM2362 `0xE6` OPCODE WHITELIST TEST RESULTS

Opcode	Command	Captures*	Result
`0x02`	Get Log Page	17	**Forwarded**
`0x06`	Identify	8	**Forwarded**
`0x09`	Set Features	6	Silently dropped
`0x0A`	Get Features	9	Silently dropped
`0x80`	Format NVM	5	Silently dropped
`0x81`	Security Receive	4	Silently dropped
`0x82`	Security Send	14	Silently dropped
`0x84`	Sanitize	6	Silently dropped

*Occurrence count from SP Toolbox USB traffic captures.

The term "silently dropped" is precise: the bridge firmware returns SCSI Good status (`0x00`) for blocked commands. The SCSI layer indicates success, the host driver reports no error, and the application (whether our tool, `nvme-cli` [13],

or SP Toolbox) believes the command was accepted. The NVMe controller never receives the command. This misdirects debugging effort toward the NVMe controller when the bridge itself is the barrier.

For NVMe admin commands that the bridge does forward (Identify, Get Log Page), a separate error arises: the bridge returns SCSI sense data with `NOT READY / Medium not present` (Sense Key `0x02`, ASC `0x3A`). This is a SCSI concept that does not exist in the NVMe specification—the bridge generates it when the NVMe controller fails to respond as expected during the bridge's initialization sequence.

C. UAS vs. BOT Mode

An additional complication arose with the USB protocol layer. The Linux `uas` (USB Attached SCSI) driver, which is the default for USB 3.0+ mass storage devices supporting the UAS protocol, returned `DID_ERROR` (host_status `0x07`, errno `-75 EOVERFLOW`) for all vendor SCSI commands (`0xE4`/`0xE5`/`0xE8`). Standard commands (`0x12` INQUIRY, `0x00` TEST UNIT READY) functioned correctly under UAS.

The solution was to force the device into BOT (Bulk-Only Transport) mode using the `usb-storage` driver with a quirks flag:

```
sudo bash -c '
  echo "1-3" > /sys/bus/usb/drivers/usb/unbind
  rmmod uas
  rmmod usb_storage
  modprobe usb-storage quirks=174c:2362:u
  echo "1-3" > /sys/bus/usb/drivers/usb/bind
'
# Device re-enumerates as /dev/sdc
# Verify: readlink /sys/.../1-3:1.0/driver
#   -> usb-storage (not uas)
```

Listing 3. Switching from UAS to BOT mode for vendor command support.

BOT mode passes raw CDB bytes in a Command Block Wrapper (CBW) without reinterpretation, whereas UAS wraps commands in Command Information Units (IUs) using a stream-based protocol that does not accommodate the non-standard 6-byte CDB format used by `0xE4` and `0xE5`. The USB VID:PID for the ASM2362 is `174c:2362`.

IV. XRAM Injection Method

A. CDB Format for Vendor Commands

The three vendor SCSI commands used for XRAM injection each have distinct CDB formats, documented through firmware reverse engineering [2]. We initially had the byte positions wrong (placing the address at incorrect offsets), and correcting this from the cyrozap Python source code was essential to achieving working operations.

The `0xE4` command reads 1–255 bytes (specified in CDB byte 1) starting from the 16-bit XRAM address in bytes 3–4 (big-endian). The data is returned via the standard SCSI data-in phase. The `0xE5` command writes a single byte (CDB byte 1) to the address in bytes 3–4—there is no data transfer phase; the value is carried entirely within the 6-byte CDB. The `0xE8` command triggers a reset: type `0x00` for a full 8051 CPU restart (reinitializes all firmware state), type `0x01` for a PCIe soft reset (link re-initialization only).

TABLE IV
Vendor SCSI CDB Byte Layouts

Command	Size	CDB Layout
`0xE4` Read	6B	`E4 [len] 00 [addr_hi] [addr_lo] 00`
`0xE5` Write	6B	`E5 [val] 00 [addr_hi] [addr_lo] 00`
`0xE8` Reset	12B	`E8 [type] 00 00 00 00 00 00 00 00 00 00`

```
pub fn buildXdataReadCdb(
    address: u16,
    length: u8,
) [6]u8 {
    return .{
        0xE4,                   // opcode
        length,                 // 1-255 bytes
        0x00,                   // padding
        @truncate(address >> 8), // addr high
        @truncate(address & 0xFF), // addr low
        0x00,                   // padding
    };
}

pub fn buildXdataWriteCdb(
    address: u16,
    value: u8,
) [6]u8 {
    return .{
        0xE5,                   // opcode
        value,                  // byte to write
        0x00,                   // padding
        @truncate(address >> 8), // addr high
        @truncate(address & 0xFF), // addr low
        0x00,                   // padding
    };
}
```

Listing 4. XDATA Read and Write CDB builders.

B. NVMe Admin Submission Queue Layout

The NVMe Admin Submission Queue occupies XRAM addresses `0xB000`–`0xB1FF`, providing 512 bytes organized as 8 physical slots of 64 bytes each, though only slots 0–3 are within the firmware-configured queue depth of 4 (see §VII-B). Each slot holds one NVMe Submission Queue Entry in little-endian byte order.

The SQE structure is serialized in little-endian format matching the NVMe specification [5]:

```
pub const NvmeSqEntry = struct {
    cdw0: u32, // Opcode[7:0], CID[31:16]
    nsid: u32, // Namespace ID
    reserved8: u32 = 0,
    reserved12: u32 = 0,
    mptr_lo: u32 = 0, mptr_hi: u32 = 0,
    prp1_lo: u32 = 0, prp1_hi: u32 = 0,
    prp2_lo: u32 = 0, prp2_hi: u32 = 0,
    cdw10: u32 = 0, cdw11: u32 = 0,
    cdw12: u32 = 0, cdw13: u32 = 0,
    cdw14: u32 = 0, cdw15: u32 = 0,

    /// Serialize to 64 LE bytes for XRAM write
    pub fn toBytes(self: NvmeSqEntry) [64]u8 {
        var out: [64]u8 = undefined;
        writeU32Le(out[0..4], self.cdw0);
        writeU32Le(out[4..8], self.nsid);
```

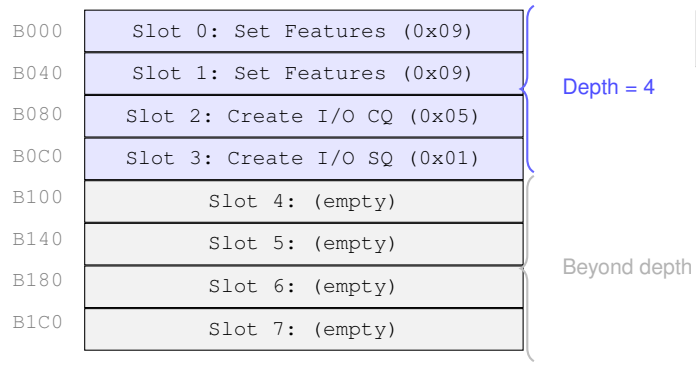

Fig. 2. Admin Submission Queue layout in XRAM. Slots 0–3 contain the bridge firmware's initialization commands (two Set Features, one Create I/O CQ, one Create I/O SQ). The Admin SQ depth is 4, not 8—a critical discovery that invalidated early injection attempts at slots 4–7.

```
        // ... remaining 14 dwords ...
        return out;
    }

    pub fn fromBytes(b: []const u8) NvmeSqEntry {
        return .{
            .cdw0 = readU32Le(b[0..4]),
            .nsid = readU32Le(b[4..8]),
            // ...
        };
    }
};
```

Listing 5. NVMe Submission Queue Entry structure with serialization.

A critical discovery was that the **Admin SQ depth is 4, not 8**. Although the XRAM provides physical space for 8 entries, the bridge firmware configures the Admin SQ with a depth of only 4 during initialization. This was determined empirically by observing the SQ Head Pointer (SQHD) field in Completion Queue entries: SQHD values wrapped at 4 rather than 8, and commands written to slots 4–7 were never consumed. There is no XRAM register that directly exposes the configured queue depth; the AQA register in BAR0 would contain this information, but PCIe memory reads through the TLP engine time out (see Section IV-D).

C. Admin Completion Queue Discovery

Locating the Admin Completion Queue (ACQ) required systematic scanning of XRAM. The CQ was not at the initially expected address `0xB800`, which turned out to be the *I/O* Completion Queue (identifiable by SQID=1 in its entries). By scanning XRAM for 16-byte structures with valid NVMe CQE fields—specifically looking for entries with SQID=0 (Admin queue)—we located the Admin CQ at `0xBC00`.

Each 16-byte CQE is parsed as follows:

```
Bytes 0- 3: DW0 (command-specific result)
Bytes 4- 7: DW1 (reserved)
Bytes 8- 9: SQHD (SQ Head Pointer)
Bytes 10-11: SQID (SQ Identifier, 0=Admin)
Bytes 12-13: CID (Command Identifier)
Bytes 14-15: Status word:
  Bit 0:  Phase (P)
  Bits 8:1: Status Code (SC)
  Bits 11:9: Status Code Type (SCT)
  Bit 14: Do Not Retry (DNR)
```

Listing 6. Admin CQ entry structure (16 bytes, little-endian).

D. PCIe TLP Doorbell Mechanism

After writing an SQE to XRAM, the NVMe controller must be notified of the new command by writing the updated tail index to the Admin SQ Tail Doorbell register at BAR0 + `0x1000` in the controller's PCIe memory space.

The ASM2362's PCIe TLP (Transaction Layer Packet) engine, documented by cyrozap [2], enables generating PCIe transactions from XRAM register writes. The engine's control registers reside in XRAM:

TABLE V
PCIe TLP Engine Registers in XRAM

Address	Size	Function
`0xB210`	12B	TLP request header (3×u32 BE)
`0xB220`	4B	TLP write data (u32 big-endian)
`0xB224`	12B	TLP completion header
`0xB254`	1B	Operation trigger
`0xB284`	1B	Config request status
`0xB296`	1B	Control/status register (CSR)

The CSR at `0xB296` has three significant bits: bit 0 (timeout occurred), bit 1 (completion received), and bit 2 (ready to send).

To determine the doorbell address, we first read BAR0 from the NVMe controller's PCIe configuration space using a Type 1 configuration read TLP (`fmt_type = 0x05`) targeting bus=1, device=0, function=0, offset=`0x10`. On our hardware, BAR0 = `0x00D00000`, placing the Admin SQ Tail Doorbell at `0x00D01000`.

The complete doorbell sequence uses a PCIe *posted* memory write TLP (`fmt_type = 0x40`). Posted writes require no completion from the target, which is critical: non-posted transactions (memory reads) through this TLP engine consistently time out, likely because the bridge firmware does not implement the completion routing required for split transactions. The posted nature of the doorbell write means the USB connection remains alive—unlike a PCIe or CPU reset, which disconnects the USB device.

```
pub fn ringDoorbell(
    device_path: []const u8,
    new_tail: u32,
) XramError!void {
    // Step 1: Read BAR0 via config space
    const bar0 = try readBar0(device_path);
    const doorbell_addr = bar0 + 0x1000;

    // Step 2: Posted memory write (0x40)
    _ = try pcieGenReq(
        device_path,
        0x40,        // fmt_type: MWr (posted)
        doorbell_addr, // BAR0 + 0x1000
        new_tail,    // new SQ tail index
        4,           // 4-byte write
    );
    // USB stays alive -- no completion needed
}
```

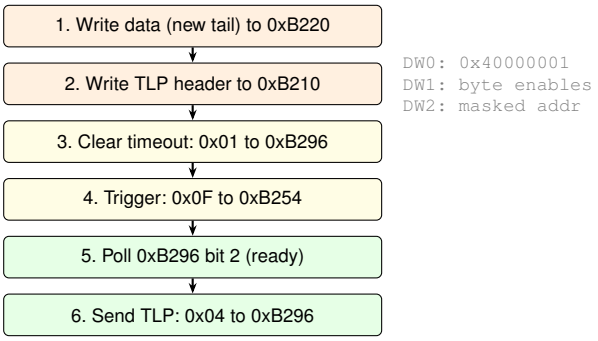

Fig. 3. PCIe TLP doorbell sequence. Each step requires one or more `0xE5` XRAM writes. The TLP header encodes a 32-bit posted memory write to the doorbell address.

Listing 7. Doorbell ring via PCIe TLP engine (simplified from xram.zig).

E. Injection Workflow

The complete injection process comprises six phases, designed with safety as the default mode of operation:

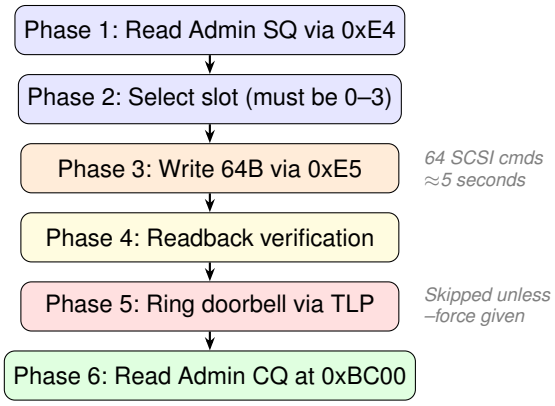

Fig. 4. XRAM injection workflow. Phases 1–4 are safe read/write operations that do not trigger command execution. Phase 5 commits the command by ringing the NVMe doorbell. Phase 6 checks the outcome.

Phase 1: Read current state. Read all 512 bytes of the Admin SQ via `0xE4`. The 255-byte-per-read limit requires three sequential reads (255 + 255 + 2 bytes). Parse into 8 `NvmeSqEntry` structures to understand the current queue contents.

Phase 2: Select slot. Choose a target slot within the active queue depth (0–3). The tool can auto-select the first empty slot or accept an explicit `-slot=N` parameter. For recovery, we overwrite slot 0 (the first Set Features command from initialization).

Phase 3: Write command. Serialize the NVMe SQE to 64 little-endian bytes via `toBytes()` and write them to XRAM one byte at a time via `0xE5`. This requires 64 individual SCSI commands, each carrying one byte in CDB byte 1, and takes approximately 5 seconds due to USB round-trip latency.

Phase 4: Verification. Read back all 64 bytes via `0xE4` and compare byte-by-byte against the intended command. If any byte mismatches, the injection is aborted before the doorbell is rung, preventing submission of a corrupted command.

Phase 5: Doorbell. Write the new tail index to the Admin SQ Tail Doorbell register via the PCIe TLP engine (Section IV-D). This phase is **skipped by default**—the `-force` flag is required to execute it. Without `-force`, the tool performs a "dry-run injection": the command is written to XRAM and verified, but never submitted to the NVMe controller.

Phase 6: Completion check. Read the Admin CQ at `0xBC00` via `0xE4` and search for a CQE matching the injected Command ID. Check the status field for success (SCT=0, SC=0x00) or specific error codes.

V. Implementation

A. Tool Architecture

The recovery tool is implemented in Zig 0.13.0+ [3], comprising approximately 5,400 lines of source code across 10 modules with 26 unit tests. Zig was chosen for its low-level control over memory layout (essential for constructing NVMe data structures with precise byte ordering), zero-cost abstractions, and direct `ioctl` system call access without FFI overhead.

TABLE VI
Source Module Organization

Module	LOC	Responsibility
`main.zig`	652	CLI entry, argument parsing, command dispatch
`scsi/sg_io.zig`	343	Linux SG_IO ioctl wrapper (`sg_io_hdr` struct)
`scsi/sense.zig`	398	SCSI sense data parsing (fixed + descriptor format)
`asm2362/passthrough.zig`	705	`0xe6` CDB protocol, NVMe opcode/feature enums
`asm2362/xram.zig`	1097	XRAM R/W, SQE inject, TLP doorbell, CQ read
`asm2362/commands.zig`	—	NVMe SMART log helpers
`nvme/identify.zig`	—	Identify Controller/NS parsing
`analysis/probe.zig`	—	Bridge/device capability detection
`frida/replay.zig`	—	USB capture replay engine
`log.zig`	—	JSON command audit logging

The `xram.zig` module is architecturally independent from the `passthrough.zig` module. It calls `sg_io.execute()` directly, constructing its own 6-byte CDBs for `0xE4`/`0xE5`/`0xE8` commands. This separation reflects the fundamental insight that XRAM injection and `0xe6` passthrough are orthogonal mechanisms operating at different levels of the bridge architecture.

The Linux SG_IO interface [14] provides the foundation for all SCSI command execution. The tool opens the block device (`/dev/sdX`) with `O_RDWR`, populates an `sg_io_hdr` structure with the CDB, data buffer, sense buffer, and timeout, then issues `ioctl(fd, SG_IO, &hdr)`:

```
pub fn execute(
    device_path: []const u8,
    cdb: []const u8,
    data_buffer: ?[]u8,
```

```
    direction: Direction,
    timeout_ms: u32,
) SgError!SgResult {
    const fd = try std.posix.open(
        device_path, .{ .ACCMODE = .RDWR }, 0);
    defer std.posix.close(fd);

    var sense_buffer: [64]u8 = .{0} ** 64;
    var hdr = SgIoHdr{
        .interface_id = 'S', // 0x53
        .dxfer_direction = @intFromEnum(direction),
        .cmd_len = @intCast(cdb.len),
        .mx_sb_len = 64,
        .dxfer_len = if (data_buffer) |b|
            @intCast(b.len) else 0,
        .dxferp = if (data_buffer) |b|
            @ptrCast(b.ptr) else null,
        .cmdp = cdb.ptr,
        .sbp = &sense_buffer,
        .timeout = timeout_ms,
    };

    const rc = std.os.linux.ioctl(
        fd, 0x2285, @intFromPtr(&hdr));
    // ... parse result ...
}
```

Listing 8. SG_IO ioctl execution (simplified from sg_io.zig).

B. Safety Measures

Given that incorrect XRAM writes could corrupt the bridge firmware's operating state or submit malformed NVMe commands, the tool implements multiple safety layers:

1) **Dry-run by default**: The `inject` command requires `-force` to ring the doorbell. Without it, phases 1–4 execute (write + verify) but phase 5 (doorbell) is skipped.
2) **Per-byte write verification**: Every `0xE5` write is followed by an `0xE4` readback to confirm the byte was stored correctly. Any mismatch returns `VerificationFailed`.
3) **Address safety checks**: Only the Admin SQ (`0xB000`–`0xB1FF`) and data buffer (`0xF000`–`0xFFFF`) regions are writable without `-force`. Writes to PCIe MMIO registers (`0xB200`+) or other sensitive regions require explicit override.
4) **Full readback after injection**: After writing all 64 bytes, a contiguous readback compares the entire SQE against the intended command. The injection is aborted before doorbell if verification fails.
5) **Explicit queue parameters**: Slot index (`-slot=N`), doorbell tail value (`-tail=N`), and Command ID (`-cid=0xNNNN`) can all be specified explicitly to prevent queue state confusion.

C. Key Code Patterns

NVMe command construction uses dedicated crafting functions that produce correctly-formed `NvmeSqEntry` structures. The Sanitize command is controller-scoped (NSID=0) with SANACT in CDW10:

```
/// sanact: 1=Exit Failure, 2=Block Erase,
///         3=Overwrite, 4=Crypto Erase
pub fn craftSanitizeEntry(
    sanact: u3,
    command_id: u16,
) NvmeSqEntry {
    return .{
        .cdw0 = @as(u32, 0x84)
            | (@as(u32, command_id) << 16),
        .nsid = 0, // Controller-level operation
        .cdw10 = @as(u32, sanact),
    };
}
```

Listing 9. Sanitize Block Erase SQE construction.

The PCIe TLP generation matches the cyrozap Python implementation [2], writing big-endian values to XRAM registers via sequences of `0xE5` commands:

```
fn writeBe32(
    device_path: []const u8,
    address: u16,
    value: u32,
) XramError!void {
    const bytes = [4]u8{
        @truncate((value >> 24) & 0xFF),
        @truncate((value >> 16) & 0xFF),
        @truncate((value >> 8) & 0xFF),
        @truncate(value & 0xFF),
    };
    for (bytes, 0..) |b, i| {
        try xdataWriteRaw(
            device_path,
            address + @as(u16, @intCast(i)),
            b,
        );
    }
}
```

Listing 10. Big-endian u32 write to XRAM via four 0xE5 commands.

VI. Recovery Process

A. Initial Controller Analysis

Using the `0xe6` passthrough (which successfully forwards Identify and Get Log Page commands), we obtained the controller's identity and health status:

TABLE VII
NVMe Controller Identity (via Identify Controller)

Field	Value
Vendor ID (VID)	`0x1E4B` (Phison Electronics)
Model Number	SPCC M.2 PCIe SSD
Firmware Revision	H211011a
Serial Number	00000000000000001387
Controller	Phison PS5012-E12
OACS	`0x0006` (Format NVM + FW Download)
SANICAP	`0x00000002` (Block Erase only)
Total NVM Capacity	256 GB

Two findings contradicted our initial hypotheses:

1) **SMART Critical Warning bit 3 was NOT set.** The controller was not reporting read-only mode through the standard NVMe mechanism. The write protection was enforced at a firmware level below the SMART reporting layer—consistent with DiskTuna's observation that functional read-only and SMART bit 3 are independent mechanisms [12].

TABLE VIII
SMART/HEALTH INFORMATION LOG PAGE

Field	Value
Critical Warning	0x00 (all bits clear)
Available Spare	1%
Available Spare Threshold	10%
Media and Data Integrity Errors	292
Unsafe Shutdowns	270
Power Cycles	297

2) **OACS reported 0x0006**, indicating the controller supports Format NVM (bit 1) and Firmware Download (bit 2). However, Format NVM proved ineffective for recovery—it operates within the corrupted FTL mapping rather than below it. Only Sanitize Block Erase successfully restored write capability (see §VI-B).

The SANICAP value of 0x00000002 indicated that only Block Erase sanitize was supported (bit 1). Crypto Erase (bit 0) was *not* available—a fact that would invalidate one of our early recovery attempts.

B. Failed Attempts

The path to successful recovery involved several informative failures, each revealing important constraints of the system:

1) Injection to Slots 4–7: Commands were written to XRAM slots 4–7, verified via readback, and the doorbell was rung. No CQ entries were generated. This led to the discovery that the Admin SQ depth is 4 (see Section IV-B): slots beyond the configured depth are simply never consumed by the NVMe controller, regardless of the doorbell tail value.

2) Sanitize Crypto Erase (SANACT=4): Attempted before consulting SANICAP. The CQ showed no response because Crypto Erase is not supported (SANICAP bit 0 = 0). The NVMe specification requires controllers to ignore unsupported sanitize actions.

3) Format NVM via Injection: A Format NVM command (opcode 0x80, SES=1, NSID=0xFFFFFFFF) was successfully injected and *accepted* by the controller—a CQE appeared in the Admin CQ with success status (SCT=0, SC=0x00). However, the FTL write protection persisted: writes continued to fail silently after the format completed. This suggested that Format NVM, operating per-namespace, could not overcome controller-level firmware protection.

4) PCIe Memory Read TLPs: We attempted to read NVMe controller registers (including AQA and controller status) by issuing PCIe memory read TLPs through the TLP engine (fmt_type 0x00 for Type 0, 0x04 for Type 1). These consistently timed out at the CSR polling stage. Only PCIe configuration space reads (0x05) and posted memory writes (0x40) function through the TLP engine. The bridge firmware apparently does not implement split transaction completion routing for memory reads.

C. Successful Recovery

The successful recovery used **Sanitize Block Erase** (SANACT=2), injected to slot 0 with explicit parameters:

```
sudo ./zig-out/bin/asm2362-tool inject \
   --inject-cmd=sanitize-block \
   --slot=0 --tail=1 --cid=0x4242 \
   --force /dev/sdc
```

Listing 11. Successful Sanitize Block Erase injection command.

The tool output confirmed each phase:

1) **Phase 1**: Admin SQ read—4 entries in slots 0–3 (firmware initialization commands).
2) **Phase 2**: Slot 0 selected (explicit override of existing Set Features command from bridge init).
3) **Phase 3**: 64 bytes written to XRAM 0xB000–0xB03F, all individually verified.
4) **Phase 4**: Full 64-byte readback matched expected Sanitize SQE.
5) **Phase 5**: BAR0 = 0x00D00000, doorbell at 0x00D01000, tail written as 1. USB connection remained live.
6) **Phase 6**: Post-injection state read.

The Admin CQ at 0xBC00 showed:

```
[0] SQHD=1 SQID=0 CID=0x4242 P=1 SCT=0 SC=0x00
   DNR=0 (Success)
```

Listing 12. Admin CQ entry confirming Sanitize acceptance.

The CID 0x4242 matched our injected command, confirming that the NVMe controller received and accepted the Sanitize Block Erase. The controller entered sanitize mode, which required approximately 90 seconds for the full 256 GB Block Erase.

Following the sanitize operation, a CPU reset (0xE8 type=0x00) forced a full NVMe controller re-initialization, causing USB re-enumeration:

```
# Force controller re-init
sudo ./zig-out/bin/asm2362-tool reset \
   --reset-type=0 /dev/sdc

# After USB re-enumeration (~3 seconds):
sudo ./zig-out/bin/asm2362-tool identify /dev/sdc
# Full Identify data returned -- controller alive

# Verify writes work:
sudo dd if=/dev/urandom of=/dev/sdc bs=1M count=16
     \
   oflag=direct
# 16+0 records out, 17.7 MB/s

sudo dd if=/dev/sdc of=/tmp/verify bs=1M count=16
# Readback matches -- writes are persistent!

# Format and use:
sudo mkfs.vfat -F 32 /dev/sdc1
# 239 GB available, formatted successfully
```

Listing 13. Post-sanitize reset and write verification.

VII. DISCUSSION

A. Why Sanitize Worked but Format Did Not

Format NVM and Sanitize represent fundamentally different levels of operation in the NVMe architecture [5]:

- **Format NVM** operates *per-namespace*. It re-creates the logical block layout and optionally performs a secure erase of user data within a single namespace. Critically, it relies on the existing FTL to map logical blocks to physical NAND pages during the erasure process.
- **Sanitize Block Erase** operates at the *controller level*, affecting all namespaces simultaneously. It issues erase commands directly to all NAND flash blocks, resetting them to a known electrical state. This forces the controller to rebuild the entire FTL mapping table from scratch during subsequent initialization.

The key distinction is that Format NVM attempts to work *within* the corrupted FTL, while Sanitize Block Erase works *below* it, resetting the physical NAND state. When the FTL itself is the source of the write protection—corrupted mapping tables cause the firmware to enter a protective read-only mode to prevent further data loss—only an operation that destroys and forces a complete FTL rebuild can restore write capability.

This finding has practical implications: for firmware write-protected NVMe drives, Sanitize Block Erase should be attempted before Format NVM, despite Format being the more commonly suggested recovery command.

B. The Admin SQ Depth Problem

The NVMe specification allows variable Submission Queue depths (configurable via the AQA register), and the ASM2362 firmware creates the Admin SQ with a depth of only 4 entries despite allocating XRAM space for 8. This is not documented in any public ASMedia documentation.

The depth was determined empirically by two methods:

1) **CQ SQHD observation**: The SQHD field in CQ entries wraps at 4 (values 0, 1, 2, 3, 0, 1, ...) rather than at 8.
2) **Slot injection testing**: Commands written to slots 4–7 and submitted via doorbell were never processed—no corresponding CQE appeared.

The AQA register in BAR0 would definitively reveal the configured queue depth, but PCIe memory reads through the TLP engine time out, preventing direct verification.

This finding corrects an assumption in previous ASM236x work that equates physical XRAM allocation with logical queue depth. Anyone implementing XRAM injection on these bridges must account for the possibility that the firmware-configured depth is smaller than the physical slot count.

C. UAS Incompatibility

USB Attached SCSI (UAS) uses stream-based Command Information Units (IUs) that expect standard SCSI CDB lengths (6, 10, 12, or 16 bytes per SPC/SBC specifications). The vendor commands `0xE4` and `0xE5` use 6-byte CDBs with non-standard opcodes in the vendor-specific range (`0xC0`–`0xFF`). While 6-byte CDBs are valid in SCSI, the UAS transport layer may interpret or validate them differently than BOT.

BOT (Bulk-Only Transport) wraps the CDB in a 31-byte Command Block Wrapper (CBW) that includes a `bCBWCBLength` field specifying the exact CDB length, passing the bytes directly to the device without reinterpretation. This makes BOT the correct transport for non-standard vendor commands.

The Linux kernel's `usb-storage` module provides a quirks mechanism: the `u` flag in `quirks=VID:PID:u` forces a device to use `usb-storage` (BOT) instead of `uas`, overriding the USB interface descriptor negotiation.

This UAS limitation is poorly documented outside of kernel source code and scattered forum posts. We recommend that any tool using ASMedia vendor SCSI commands include a UAS detection and BOT-switching routine.

D. XRAM Memory Map

Through systematic scanning of the full 64 KB XRAM address space using `0xE4` reads, we constructed a detailed memory map:

TABLE IX
ASM2362 XRAM MEMORY MAP (EMPIRICALLY VERIFIED)

Address Range	Size	Contents
`0x0000–0x07FF`	2 KB	8051 CPU workspace
`0x07F0–0x07F5`	6B	Firmware version string
`0x9000+`	var.	USB/SCSI control regs
`0xA000–0xAFFF`	4 KB	NVMe I/O Submission Queue
`0xB000–0xB1FF`	512B	Admin Submission Queue (depth=4)*
`0xB200–0xB296`	var.	PCIe TLP engine regs
`0xB800–0xB8FF`	256B	I/O Completion Queue (SQID=1)
`0xBC00–0xBC7F`	128B	Admin Completion Queue (SQID=0)
`0xF000–0xFFFF`	4 KB	NVMe data buffer (Identify cache)

*Physical allocation is 512 B (8 slots), but firmware-configured queue depth is 4; see §VII-B.

The firmware version at `0x07F0` on our device read `19 05 02 81 16 00`, which does not correspond to any human-readable version string but serves as a firmware fingerprint.

The data buffer at `0xF000` caches the most recent NVMe data transfer—after an Identify Controller command, this region contains the 4 KB Identify Controller data structure. This buffer is readable and writable, though writing to it has no direct effect on NVMe controller state.

E. Limitations

Several limitations constrain the practicality and generality of this approach:

1) **Single-byte writes**: The `0xE5` command writes one byte per SCSI transaction. Injecting a single 64-byte SQE requires 64 individual SCSI commands, taking approximately 5 seconds. This is inherent to the vendor command protocol and cannot be optimized.
2) **No PCIe memory reads**: Memory read TLPs through the TLP engine time out. This prevents reading NVMe

controller registers (AQA, CSTS, CC) and limits observability to what can be inferred from XRAM contents and CQ entries.

3) **BAR0 variability**: The doorbell address depends on BAR0, which may vary across firmware versions or different NVMe controllers. Our tool reads BAR0 dynamically, but the PCIe bus/device/function topology (bus=1, dev=0, fn=0) is hardcoded and may differ on other ASM236x variants.
4) **No CQ interrupt**: There is no interrupt mechanism for CQ updates through XRAM. Completion must be detected by polling, which introduces a timing window between command completion and detection.
5) **Destructive recovery only**: The technique supports drive recovery (where data loss is acceptable) but not data-preserving recovery. Sanitize destroys all user data by design.
6) **Bridge-specific**: The XRAM memory map, vendor command formats, and TLP engine registers are specific to the ASM236x family. Other bridge chipsets (JMicron JMS583, Realtek RTL9210, VIA VL716) may have different architectures.

F. Security Implications

The XRAM injection technique has security implications beyond recovery. The bridge's `0xe6` opcode whitelist creates an appearance of access control—suggesting that Format, Sanitize, and Security commands cannot be issued over USB. In reality, the `0xE4`/`0xE5` vendor commands provide unrestricted access to the bridge's XRAM, enabling:

- Submission of *any* NVMe admin command, including data destruction (Sanitize), namespace erasure (Format), and security credential changes (Security Send).
- Reading of NVMe data cached in the XRAM data buffer (`0xF000`+), potentially including sensitive information from recent Identify or log page queries.
- Modification of bridge firmware state, as the XRAM contains the 8051 CPU's working memory.

The bridge firmware does not perform any authentication for `0xE4`/`0xE5` access, and the firmware itself is not signature-verified [2]. This means that USB-connected NVMe drives behind ASM236x bridges should not be considered protected by the bridge's command filtering from a security perspective.

The TCG Opal specification [15] addresses storage security through authenticated command sets, but its effectiveness depends on commands reaching the NVMe controller—which our technique can ensure even when the bridge would normally block them.

VIII. Comparison with Existing Approaches

Table X compares our XRAM injection approach to existing NVMe SSD recovery methods across several practical dimensions.

Our approach uniquely enables NVMe drive recovery from Linux over a USB connection, without proprietary software, native M.2 PCIe access, or Windows. This is particularly valuable in scenarios where:

- The drive cannot be physically removed from its USB enclosure (e.g., soldered connections in portable SSDs).
- No M.2 slot is available on the recovery system.
- Windows is not available or preferred.
- The cost of professional recovery is not justified for a drive where data preservation is unnecessary.

IX. Conclusion

We have presented a novel technique for recovering NVMe SSDs from firmware-level write protection using only a USB connection through an ASMedia ASM2362 bridge. The key insight is that while the bridge's `0xe6` passthrough CDB implements a restrictive opcode whitelist that blocks all recovery commands, the bridge's internal XRAM—directly accessible through vendor SCSI commands `0xE4` and `0xE5`—contains the NVMe Admin Submission Queue and PCIe TLP engine registers that enable arbitrary command injection and doorbell signaling.

The technique successfully recovered a Phison PS5012-E12 based 256 GB SSD from permanent silent-write-failure mode caused by Windows hibernate corruption. After multiple failed approaches—Format NVM (accepted but ineffective), Crypto Erase (unsupported), out-of-depth slot injection (outside queue boundary)—Sanitize Block Erase restored full write capability by operating below the corrupted FTL and forcing a complete mapping table rebuild.

Key technical findings include:

1) The ASM2362 Admin SQ depth is 4, not 8, despite physical XRAM space for 8 entries.
2) UAS (USB Attached SCSI) mode is incompatible with ASMedia vendor SCSI commands; BOT (Bulk-Only Transport) mode is required.
3) Sanitize Block Erase succeeds where Format NVM fails because it operates below the corrupted FTL, forcing a complete rebuild.
4) PCIe posted memory writes through the TLP engine maintain USB connectivity, enabling doorbell writes without disconnecting the device.
5) The Admin CQ is at XRAM `0xBC00`, not `0xB800` (which is the I/O CQ).
6) SMART Critical Warning bit 3 and firmware-level write protection are independent mechanisms—a drive can be functionally read-only without bit 3 being set.

The recovery tool is implemented in approximately 5,400 lines of Zig with 26 unit tests and is available as open source. The methodology is applicable to any NVMe drive connected through an ASM236x-series USB bridge, and the general approach of XRAM-based command injection may extend to other bridge chipsets with similar vendor SCSI command interfaces.

USB-to-NVMe bridges are not transparent tunnels—their internal XRAM constitutes both an attack surface and, as demonstrated here, a recovery surface.

TABLE X
COMPARISON OF NVME SSD RECOVERY APPROACHES

Method	Connection	Cost	Data	Linux	Windows	Key Limitation
XRAM Injection (this work)	USB bridge	Free	Destroyed	Yes	No	ASM236x bridges only
Phison Reinitial [8]	USB bridge	Free	Destroyed	No	Yes	Requires matching FW files
`nvme-cli` [13]	Native M.2	Free	Destroyed	Yes	No	Needs physical M.2 slot
PC-3000 SSD [4]	Proprietary	$300–1500	Preserved	No	Yes	Professional equipment
Manufacturer RMA	N/A	Warranty	Destroyed	N/A	N/A	Weeks turnaround

ACKNOWLEDGMENTS

The author thanks cyrozap for the foundational ASMedia firmware reverse engineering work [2] without which this technique would not have been possible, smx-smx for the ASMTool firmware dumper [7], the smartmontools developers for the ASMedia bridge driver reference implementation [6], and the Linux kernel SCSI subsystem developers for the SG_IO interface [14].

REFERENCES

[1] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Understanding the robustness of SSDs under power fault," *Proceedings of USENIX FAST*, 2013, 13/15 consumer SSDs lost data in power fault testing.

[2] cyrozap, "usb-to-pcie-re: USB-to-PCIe bridge reverse engineering," 2023, aSM236x XRAM memory map, vendor SCSI commands, PCIe TLP engine documentation. [Online]. Available: https://github.com/cyrozap/usb-to-pcie-re

[3] Zig Software Foundation, "Zig programming language," 2024, version 0.13.0+. [Online]. Available: https://ziglang.org/

[4] ACE Lab, "PC-3000 SSD phison utility," 2024. [Online]. Available: https://blog.acelab.eu.com/pc-3000-ssd-phison-utility.html

[5] NVM Express, Inc., "NVM Express Base Specification, Revision 2.0e," 2024, sections 5.14.1.2 (SMART Critical Warning), 5.24 (Sanitize), 5.15 (Format NVM). [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0e-2024.07.29-Ratified.pdf

[6] smartmontools developers, "smartmontools: S.M.A.R.T. monitoring tools," 2024, sntasmedia_device class in scsinvme.cpp. [Online]. Available: https://github.com/smartmontools/smartmontools

[7] smx smx, "ASMTool: ASMedia firmware dumper," 2023. [Online]. Available: https://github.com/smx-smx/ASMTool

[8] usbdev.ru, "Phison PS5012 reinitial tool (ECFM22.6)," 2024, recovery tool for PS5012-E12 controllers in read-only mode. [Online]. Available: https://www.usbdev.ru/files/phison/ps5012reinitialtool/

[9] Rossmann Group, "SSD data recovery services," 2025, pC-3000 SSD with PS5012-E12 support. [Online]. Available: https://rossmanngroup.com/services/ssd-data-recovery

[10] D. Gilbert, "sg3_utils: Utilities for SCSI devices," 2024. [Online]. Available: https://sg.danny.cz/sg/sg3_utils.html

[11] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery," in *arXiv preprint arXiv:1711.11427*, 2017.

[12] DiskTuna, "A write protected SSD/NVMe read only," 2023. [Online]. Available: https://www.disktuna.com/a-write-protected-ssd-nvme-read-only/

[13] linux-nvme, "nvme-cli: NVMe management command line interface," 2024. [Online]. Available: https://github.com/linux-nvme/nvme-cli

[14] Linux Kernel Documentation, "SCSI Generic (sg) driver interface," 2024, sG_IO ioctl for SCSI passthrough. [Online]. Available: https://www.kernel.org/doc/html/latest/scsi/scsi-generic.html

[15] NVM Express, Inc. and Trusted Computing Group, "TCG storage Opal and NVMe joint white paper," 2022. [Online]. Available: https://nvmexpress.org/wp-content/uploads/TCGandNVMe_Joint_White_Paper-TCG_Storage_Opal_and_NVMe_FINAL.pdf