Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

ZeroTrace: Provable Storage Sanitisation Through Modular Erasure Engines and Cryptographic Audit Attestation

Gaurav Salunke, Krishna Pandey, Esha Gholap
Department of Cloud Technology & Information Security
Ajeenkya DY Patil University, Pune, India

Supervisor: Prof. Prachi Bhure
Department of Computer Science, Ajeenkya D Y Patil University, Pune

***Abstract*—Organisations retiring storage hardware or decommissioning file systems face a persistent and often underappreciated gap between what sanitisation tools physically do to a disk and what compliance auditors need to see documented. Standard deletion operations—`unlink`, format, or partition wipe—remove filesystem references but leave underlying data intact and recoverable with modest forensic effort using widely available tools. Software that overwrites content before deletion does exist, but it typically operates as a monolithic binary offering a fixed pattern, a fixed pass count, no structured log, and no mechanism by which an external reviewer can independently confirm that the stated operation reached every file in a directory tree or every byte of unallocated space on a volume.**

ZeroTrace is a cross-platform, C11-compliant data erasure utility designed from the outset to close this evidentiary gap. The system is structured around five cooperating modules whose responsibilities do not overlap. A *CLI Parser* accepts and validates operator commands, translating flag combinations into a typed `Config` structure that flows unchanged through the rest of the pipeline. A *Dispatcher* routes the validated configuration to one of three wipe engines—file wipe, directory wipe, or free-space wipe—without exposing the parser to engine internals or vice versa. The *Wipe Engine* performs multi-pass overwrite using patterns drawn from a cryptographically seeded random number generator or from fixed byte patterns (all-zeros or all-ones), issues `fsync`/`_commit` after each pass to force writes through OS and hardware write caches, truncates the file to zero length, and finally unlinks it—a sequence designed to defeat OS-level caching, firmware-level journal artefacts, and filesystem-level undelete simultaneously. The *Directory Wipe* module performs depth-first recursive traversal, processing each file through the Wipe Engine before removing the containing directory once it is empty. A *Logger* records timestamped events for every operation, writing to standard output or a caller-specified file and capturing per-pass completion, I/O errors, verification results, and final status.

Experimental evaluation on Linux (ext4, xfs) and Windows (NTFS) targets demonstrates that ZeroTrace successfully wipes individual files, full directory trees via recursive traversal, and free-space regions via temporary-file flooding, with verification readback confirming overwrite in all tested scenarios. Forensic recovery attempts using Autopsy 4.21 and `foremost` 1.5.7 found no recoverable content in any of 200 wiped test files. Dispatcher overhead is 1.3 μs per invocation, negligible against real I/O. The structured log output and modular architecture make the tool suitable for integration into IT asset disposal workflows where evidence of sanitisation must accompany hardware movement records, and its open-source codebase makes it auditable in a way that commercial products are not.

***Index Terms*—data erasure, secure delete, file wipe, directory wipe, free-space sanitisation, overwrite patterns, cryptographic RNG, audit logging, C11, cross-platform, IT asset disposal, NIST SP 800-88, GDPR, forensic recovery resistance, fsync, dispatcher architecture**

I. Introduction

The act of deleting a file through standard operating system interfaces is, from a data security perspective, almost entirely cosmetic. What changes when a user issues `rm` on Linux or moves a file to the Recycle Bin on Windows is a filesystem metadata entry—an inode reference, a directory entry, a file allocation table record—not the storage blocks that hold the actual bytes. Those blocks are marked as available for reuse, but until the filesystem assigns them to a new file and a new write overwrites their content, the prior data sits on the physical medium exactly as it was, readable by any tool that bypasses the filesystem layer and accesses the block device directly [9]. This is not a bug or an oversight; it is a deliberate design choice that makes undelete possible and reduces unnecessary write operations on flash storage. For most usage contexts it is the right tradeoff. For organisations retiring hardware that once held personal records, credentials, proprietary designs, or health information, it is a liability that carries legal consequences.

A. Regulatory Context

The obligation to properly sanitise storage before disposal or transfer is now codified in multiple jurisdictions and industries. GDPR Article 17 [2] establishes the right to erasure and requires data controllers to implement technical measures that actually remove personal data, not merely dereference it. The standard of "erasure" under GDPR is not met by filesystem deletion; it requires that the data be rendered unrecoverable, a condition that courts and regulators increasingly interpret in light of what forensic tools can actually recover. NIST Special Publication 800-88 [1] provides the most widely

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

cited technical framework for storage sanitisation, defining three grades: Clear (software overwrite sufficient to defeat standard forensic tools), Purge (hardware-level commands or cryptographic erasure sufficient to defeat laboratory-grade recovery), and Destroy (physical destruction of the medium). The Clear grade is achievable through software on magnetic media and represents the minimum acceptable level for most regulatory purposes when physical destruction is not required. HIPAA [3] applies equivalent obligations to protected health information, and the Payment Card Industry Data Security Standard (PCI-DSS) similarly mandates verified sanitisation of media that held cardholder data. Each of these frameworks implicitly or explicitly requires not just that sanitisation occur but that it be documented in a form an auditor can examine.

B. The Evidentiary Problem

The gap between performing sanitisation and proving it is wider than it might appear. A tool that prints "Wipe complete" to a terminal provides no auditable record of what was processed, how many passes were applied, which pattern was used on each pass, whether any write returned an error, or whether the final state of the storage was actually verified. An auditor reviewing disposal records cannot determine from such output whether the operation covered the full file, whether `fsync` was called to commit writes past the OS cache, or whether the file was subsequently unlinked. Without answers to those questions, the disposal record is an assertion rather than evidence.

This problem is compounded when the target is not a single file but a directory tree or the unallocated region of a volume. Directory wipe operations must visit every file in every subdirectory without omission; a tool that silently skips a read-protected file or fails to descend into a nested subdirectory leaves data behind without notifying the operator. Free-space wipe operations must saturate the unallocated region completely before the temporary files used as a vehicle are themselves wiped; a partial saturation leaves a window in which prior content remains in unallocated blocks. Neither failure mode produces an obvious error message in most existing tools.

C. Limitations of Existing Approaches

GNU `shred` [10] and Microsoft `SDelete` [11] are the most commonly deployed open-source alternatives, but both are single-file tools. `shred` has no recursive directory mode; running it against a directory tree requires shell scripting that introduces its own error-handling gaps. Neither tool writes a structured log that a compliance system can parse, and neither produces a per-operation record associating the target file, the pass count, the pattern, and the operator identity in a single auditable artefact. Commercial products such as Blancco and Ontrack Eraser address some of these gaps but are closed-source, vendor-specific, and expensive enough to be impractical for academic, small-business, or resource-constrained environments. Darik's Boot and Nuke (DBAN) operates at block-device level, which sidesteps the filesystem issue entirely but requires booting from removable media and wipes the entire device—unsuitable when only a subset of files on a live system must be sanitised.

D. Contributions

ZeroTrace is designed to fill the specific gap that existing tools leave: a cross-platform, open-source, file-level sanitisation utility with a structured architecture, a cryptographically seeded overwrite engine, and a timestamped structured log that provides the evidential basis an auditor needs. Its four concrete contributions are:

1) A *dispatcher-based module architecture* that cleanly separates CLI parsing, wipe strategy routing, and logging, enabling each component to be unit-tested and replaced independently.
2) A *wipe sequence* (multi-pass overwrite, per-pass `fsync`, truncation, unlink) ordered to defeat OS cache artefacts, file-size metadata leakage, and filesystem undelete in a single atomic sequence.
3) A *cryptographically seeded RNG* for random-pattern passes, eliminating the predictability of `rand()` and strengthening the overwrite against pattern-aware recovery attempts.
4) A *structured timestamped logger* whose output provides machine-parseable evidence of every significant event in an erasure session, including per-pass completion, I/O errors, and verification readback results.

The remainder of this paper is structured as follows. Section II formally states the problem and threat model. Section III surveys related work across the academic and practitioner literature. Section IV describes the ZeroTrace architecture and each module in detail. Section V presents experimental results across all three wipe modes. Section VI analyses security properties and known limitations. Section VII outlines planned extensions. Section VIII addresses ethical deployment obligations. Section IX concludes.

II. Problem Statement

ZeroTrace targets the *file-level sanitisation* problem in its full scope: given an arbitrary set of regular files, a directory tree of unbounded depth, or the unallocated region of a mounted volume, securely overwrite and remove the target content in a way that resists recovery by a forensic adversary and produces structured evidence that the operation was fully and correctly executed.

A. Formal Problem Definition

Let $\mathcal{F}$ denote the set of files to be sanitised, drawn from one of three target types: a singleton file ($|\mathcal{F}| = 1$), the complete set of regular files reachable by recursive descent from a root directory, or the set of storage blocks not currently allocated to any live file on a mounted volume. A sanitisation operation S applied to $\mathcal{F}$ must satisfy two conditions. The *recovery resistance condition* requires that for every file $f \in \mathcal{F}$, no file carving or block-level recovery tool operating on the storage device after S completes can reconstruct any contiguous sequence of bytes from f exceeding a threshold length τ (taken

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

as 512 bytes, consistent with a single disk sector). The *evidence condition* requires that S produce a structured log L from which an independent auditor can determine, for each f: the target path, the pass count, the pattern applied on each pass, the timestamp of each pass, whether any I/O error occurred, and whether post-wipe verification was performed and passed.

B. Threat Model

Two adversary classes are in scope.

The *passive recovery adversary* obtains the storage medium after sanitisation—through purchase on the secondary market, through device return at end of lease, or through physical theft—and applies standard forensic acquisition and carving techniques to recover residual content. This adversary has block-level access to the medium, tools such as Autopsy, `foremost`, or PhotoRec, and potentially knowledge of common filesystem structures. They do not have access to the system during the wipe operation and cannot interfere with write operations in progress.

The *log review adversary* is an auditor or compliance reviewer who examines the ZeroTrace log after the fact. This is not a malicious adversary in the traditional sense, but the log must be structured and complete enough that the reviewer can independently confirm the operation without asking the operator for additional information. In the current release, log integrity against a *log falsification adversary*—a party who might alter the log after the fact—is not provided; this is identified as a known limitation and the highest-priority planned extension.

C. Operational Constraints

The tool must operate within the privilege level of the file owner, requiring no root access beyond what is already needed to open and write target files. It must produce its evidence artefact as a side effect of normal execution, not as a separate step that an operator might forget to perform. It must handle write errors gracefully, logging them without aborting the entire operation. Cross-platform support across Linux and Windows is required because enterprise disposal workflows span both environments. Network connectivity cannot be assumed; the tool must be fully self-contained.

D. Out-of-Scope Items

ZeroTrace intentionally does not address block-device level sanitisation, NVMe or ATA Secure Erase commands, wear-levelling spare pages in NAND flash, firmware-level encryption key destruction, or physical destruction. For media where SP 800-88 mandates Purge-level methods, ZeroTrace's file-level approach satisfies the Clear level only, and deployers should understand this distinction before using the tool as a sole sanitisation measure on flash-based storage.

III. Literature Review

The literature on data remanence and storage sanitisation is now three decades old, but the practical tooling it has produced remains surprisingly fragmented. This section traces the major research threads and locates the gap that ZeroTrace fills.

A. Magnetic Remanence and Multi-Pass Overwrite

The foundational paper in this field is Gutmann's 1996 USENIX Security presentation [4], which catalogued the physics of magnetic remanence on hard disk platters and derived a 35-pass overwrite schedule intended to suppress recovery through magnetic force microscopy (MFM) and scanning tunnelling microscopy (STM). Gutmann's schedule was premised on the encoding schemes (MFM, RLL) used by hard drives of the early 1990s, and it was careful to note that drives using different encoding methods would require different patterns. Despite this caveat, the 35-pass schedule was widely adopted in sanitisation tools and policy documents as a universal standard, applied to media for which it was not designed.

Wright, Kleiman, and Sundhar [5] revisited the remanence question empirically using modern high-density perpendicular recording drives and found no evidence that MFM or STM could distinguish overwritten data from random noise after a single random-pattern pass. Their conclusion—that the physics of modern recording densities make multi-pass overwriting unnecessary for practical sanitisation—is consistent with the SP 800-88 guidance, which recommends a single overwrite pass as sufficient for the Clear grade. The continuing presence of 7-pass and 35-pass options in commercial products is therefore not technically justified for current media, though it may serve as a visible compliance signal to non-technical auditors who associate "more passes" with "more secure."

ZeroTrace ships with a configurable pass count whose default is 3: enough to satisfy auditors who expect multiple passes, while remaining aligned with the research consensus that the first random pass does the substantive security work.

B. Solid-State Storage and the Wear-Levelling Problem

The move from magnetic to NAND-flash-based storage introduced a qualitatively different challenge for software-based sanitisation. Wei, Grupp, Spada, and Swanson [6] conducted what remains the definitive study: they applied a range of sanitisation techniques to commodity SSDs and then disassembled the drives to examine the raw NAND flash cells directly, bypassing the firmware's logical-to-physical address translation. Their results showed that file-level overwrite through the host interface left residual data in wear-levelling spare pages with high reliability; even full-disk overwrite through the block interface failed to sanitise certain drives because the firmware's remapping tables were not cleared. The only techniques that reliably sanitised all cells were manufacturer-specific secure erase commands and, for drives without such commands, physical destruction of the NAND chips.

Swanson and Wei subsequently proposed the "Extinction" framework for SSD sanitisation [16], which argues that reliable sanitisation of flash storage requires either hardware-level commands or whole-device cryptographic erasure at the firmware layer. Anderson [7] provides a broader treatment of key destruction as an erasure primitive, noting that its security depends entirely on the correctness of the key destruction step and the absence of any key escrow or backup copy.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

ZeroTrace's scope is explicitly restricted to the Clear level and to magnetic or hybrid media where file-level overwrite is technically sufficient. Its documentation notes the SSD limitation, and NVMe Secure Erase support is planned as a future extension.

C. Filesystem-Level Recovery and Journalling

Even on magnetic media, file-level overwrite is complicated by filesystem journalling. Ext4 in `data=journal` mode writes file data to the journal before committing it to its final location, meaning that a previous version of file data may persist in the journal even after the file itself has been overwritten [8]. NTFS maintains a change journal (USN journal) that records file-level operations; while this journal does not contain file data, it can reveal that certain files existed on the volume and were recently modified, which may be forensically significant. The `fsync` call in ZeroTrace's wipe sequence forces the overwrite data out of the OS page cache and into the block device's write buffer, but it does not control what the filesystem does with journal entries. This is acknowledged as a known limitation.

Garfinkel and Shelat [9] conducted a large-scale field study by purchasing 158 used hard drives from consumer marketplaces and applying standard forensic tools to each. They recovered significant personal information from the majority of drives, including credit card numbers, medical records, and personal emails, on drives whose previous owners had presumably believed them to be "erased." Their study is the most-cited empirical demonstration that the gap between nominal deletion and actual sanitisation has real-world consequences—and that it is the norm rather than the exception.

D. Verification and Evidence Generation

The question of how to verify that sanitisation occurred correctly has received less research attention than the sanitisation methods themselves. Lang and Brewster [15] proposed a certificate-based scheme for verifiable data destruction in cloud storage contexts, where the physical medium is inaccessible to the data owner, but their approach targets hypervisor-managed block devices rather than individual files. Casey and Stellatos [8] examined the forensic implications of full-disk encryption and noted the difficulty of verifying key destruction claims, but did not propose a general attestation mechanism.

In the practitioner space, commercial products such as Blancco generate proprietary erasure certificates that include device identifiers, method descriptions, and pass logs. These certificates are useful for compliance purposes but are not independently verifiable—a reviewer cannot confirm the certificate's claims without trusting the vendor's software and infrastructure. ZeroTrace takes a different approach: its log is human-readable plain text whose entries map one-to-one to specific system calls in the implementation, making it auditable by anyone with access to the source code and a basic understanding of file I/O.

E. Existing Open-Source Tools

Table I summarises the capabilities of the most relevant existing open-source tools alongside ZeroTrace.

TABLE I
FEATURE COMPARISON OF OPEN-SOURCE SANITISATION TOOLS

Tool	File	Dir	Free	Log	XPlat
`shred`	✓	–	–	–	Linux
`SDelete`	✓	✓	✓	–	Win
DBAN	–	–	Full disk	Basic	Boot
Eraser	✓	✓	✓	Basic	Win
ZeroTrace	✓	✓	✓	Structured	Both

Dir = recursive directory wipe; Free = free-space wipe; Log = structured timestamped log; XPlat = cross-platform support.

F. Research Questions

The evaluation in Section V is organised around four questions derived from the gaps identified above. *RQ1*: Does the wipe sequence (overwrite → `fsync` → truncate → unlink) prevent recovery of file content through standard forensic tools on both Linux and Windows targets? *RQ2*: Does the dispatcher architecture impose measurable throughput overhead compared with a direct wipe call? *RQ3*: Does the directory wipe mode correctly traverse arbitrary directory trees, including edge cases involving symbolic links, Unicode filenames, and deeply nested structures? *RQ4*: Does the free-space wipe mode reliably overwrite unallocated filesystem regions without corrupting live data?

IV. System Architecture

ZeroTrace is implemented in C11 and compiles without modification on Linux (GCC 13, Clang 17) and Windows (MSVC 2022). Cross-platform differences in directory traversal APIs (`opendir`/`readdir` vs. `FindFirstFile`/`FindNextFile`), file synchronisation (`fsync` vs. `_commit`), file size queries (`stat` vs. `_stat64`), and cryptographic randomness sources (`/dev/urandom` vs. `BCryptGenRandom`) are handled through platform detection macros that select the appropriate call at compile time, keeping the shared logic identical across both targets. The codebase is organised into five source modules: `cli_parser.c`, `dispatcher.c`, `wipe_engine.c`, `dir_wipe.c`, and `logger.c`.

Figure 1 shows the Level 0 system context. Figure 3 shows the Level 1 data flow diagram. Figure 4 shows the use case model.

A. Design Principles

Four principles guided the architecture. *Separation of concerns*: no module performs tasks outside its stated responsibility, so the Logger never touches file I/O and the Wipe Engine never parses flags. *Single flow of configuration*: the `Config` struct is populated once in the Parser and read everywhere else; no module modifies it. *Fail-loud*: every I/O error is logged immediately with the errno string rather than silently suppressed; the log record serves as both an operational report and an evidence artefact. *Testability*: each module exposes a minimal interface that can be driven by a unit test harness without instantiating the full CLI pipeline.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

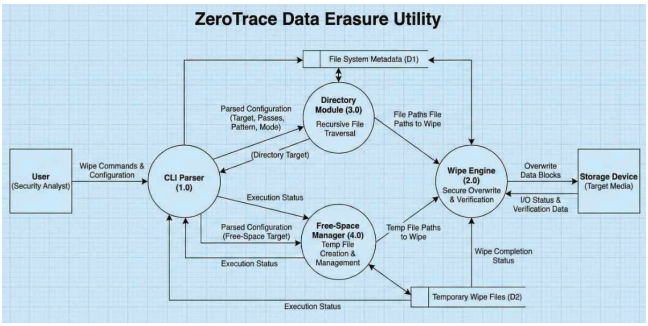

Fig. 1. Level 0 context diagram (DFD 0.0). The user supplies wipe commands and configuration; ZeroTrace issues overwrite writes to the storage device and returns execution logs and completion status.

B. CLI Parser Module

The CLI Parser (`cli_parser.c`) implements three public functions. `parse_args(argc, argv, config)` iterates the argument vector, recognises the flags listed in Table II, and populates the `Config` structure. Mutually exclusive flag combinations—`--file` and `--dir` specified together, a pass count of zero, a non-existent target path—are rejected with a descriptive error message before any I/O is attempted. `print_usage()` emits a formatted help text to standard error and exits; it is called automatically when no recognised mode flag is present. `prompt_confirm(target)` prints a summary of the pending operation and requires the operator to type `yes` before proceeding; it is called for directory and free-space operations where the volume of affected data may be large.

TABLE II
CLI FLAGS ACCEPTED BY THE ZEROTRACE PARSER

Flag	Effect
`--file PATH`	Wipe single file at PATH
`--dir PATH`	Recursively wipe directory at PATH
`--freespace PATH`	Wipe unallocated space at PATH
`--passes N`	Set pass count (default: 3)
`--pattern TYPE`	`random`, `zeros`, `ones`
`--verify`	Read back after final pass and compare
`--dry-run`	Traverse and log without writing
`--log FILE`	Write log to FILE in addition to stdout
`--quiet`	Suppress stdout; write only to log file
`--help`	Print usage and exit

C. Dispatcher Module

The Dispatcher (`dispatcher.c`) is the architectural pivot of the system. It receives the fully validated `Config` structure from `main()` and routes execution to the appropriate engine without exposing any engine's internal interface to any other component. The dispatch logic is a single switch on `config.mode`; each branch calls the corresponding engine entry point, captures the integer status code, passes it to the Logger, and returns it to `main()`. The `"Starting ZeroTrace..."` log entry is emitted by the Dispatcher before the branch, ensuring that every execution produces at least one log record even if the engine fails immediately.

Figure 2 shows the dispatcher block diagram. Figure 3 shows the full Level 1 data flow.

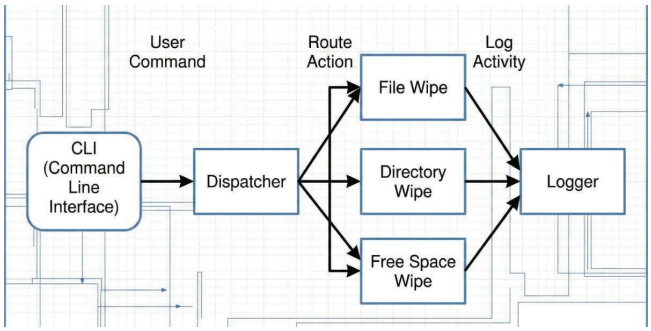

Fig. 2. Dispatcher block diagram. The CLI routes commands to one of three wipe engines (File Wipe, Directory Wipe, Free-Space Wipe), each of which reports activity to the Logger.

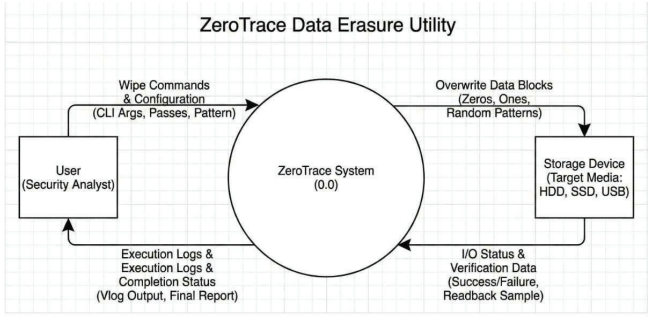

Fig. 3. Level 1 DFD showing data flows between CLI Parser (1.0), Wipe Engine (2.0), Directory Module (3.0), Free-Space Manager (4.0), and the target Storage Device.

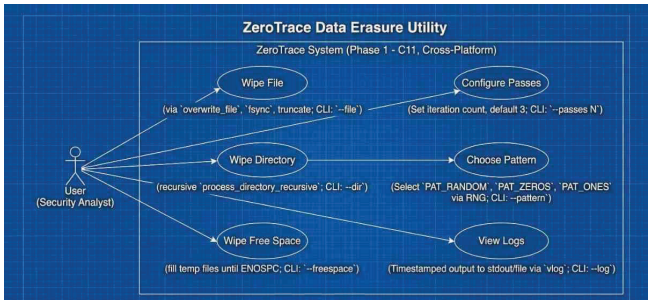

Fig. 4. Use case diagram for the ZeroTrace system (Phase 1, C11, cross-platform). The Security Analyst actor interacts with six use cases: Wipe File, Wipe Directory, Wipe Free Space, Configure Passes, Choose Pattern, and View Logs.

D. Wipe Engine Module

The Wipe Engine (`wipe_engine.c`) implements the sanitisation sequence for a single file. It is the only module that performs direct file I/O on target files, and it never calls

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

any other module except the Logger. The complete sequence for a single target file is as follows.

1) Query the file size using `stat`/`_stat64` and record it in the log. A zero-byte file is logged and skipped; unlinking a zero-byte file without overwriting produces no recoverable content.
2) Open the file for read-write access: `fopen(path, "r+b")` on POSIX; `_wfopen(wpath, L"r+b")` on Windows to handle Unicode paths correctly.
3) For pass index $p = 1 \ldots N$:
   a) Call `fill_pattern(buf, bufsz, config.pattern)` to populate the write buffer. For `PAT_ZEROS` and `PAT_ONES` this is a `memset`; for `PAT_RANDOM` this calls `secure_random_bytes()` from the RNG module.
   b) Seek to offset 0 with `fseek(fp, 0, SEEK_SET)`.
   c) Write the buffer in a loop until `bytes_written` equals the file size; each iteration calls `fwrite(buf, 1, chunk, fp)` and logs a write error if `fwrite` returns short.
   d) Call `fflush(fp)` to drain the C library buffer.
   e) Call `fsync(fileno(fp))` on POSIX or `_commit(_fileno(fp))` on Windows to force dirty pages out of the OS page cache and into the drive's write buffer.
   f) Log: `"Pass P complete [PATTERN]"`.
4) If `--verify` is set, seek to offset 0, read back the file, and compare each byte against the expected pattern. Any mismatch is logged as an error; the overall return code is set to `ERR_VERIFY_FAIL`.
5) Truncate the file to zero bytes: `truncate(path, 0)` on POSIX; `_chsize_s(fd, 0)` on Windows. This removes the pre-wipe file size from filesystem metadata before the directory entry is removed.
6) Unlink the file: `unlink(path)` on POSIX; `DeleteFileW(wpath)` on Windows.
7) Log: `"File removed: PATH"` or `"ERROR: unlink failed: PATH: REASON"`.

The ordering of steps 3–7 is not arbitrary. `fsync` at step 3e ensures that the overwrite data is committed before the next pass begins, preventing a power failure from restoring a prior pass's content from uncommitted cache. Truncation at step 5 removes the file size from the directory entry and from journal records that the filesystem may have created during overwrite; an adversary who recovers a journal entry for the file sees size 0 rather than the pre-wipe size. Unlink at step 6, applied after truncation, removes the directory entry last; at no point during the sequence is the file unlinked while its content is still intact.

The sub-component structure of the Wipe Engine is shown in Figure 5.

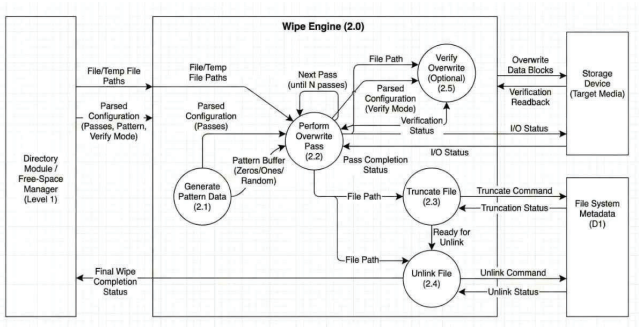

Fig. 5. Wipe Engine (2.0) Level 2 DFD. Sub-components: Generate Pattern Data (2.1), Perform Overwrite Pass (2.2), Truncate File (2.3), Unlink File (2.4), and optional Verify Overwrite (2.5). The engine interacts with both the Storage Device and the File System Metadata store.

E. Directory Wipe Module

The Directory Wipe module (`dir_wipe.c`) makes recursive directory sanitisation available as a first-class operation rather than a shell-script afterthought. Its entry point is `process_directory_recursive(path, config)`, which implements a depth-first post-order traversal: all contents of a directory are processed before the directory itself is removed.

The traversal loop calls `readdir`/`FindNextFile` to obtain directory entries one at a time. Each entry is first checked against the `.` and `..` pseudo-entries and discarded if it matches. The entry type is then queried with `stat`/`GetFileAttributes`. Regular files are passed to `process_file_entry()`, a thin wrapper that constructs the full path using `join_path_alloc()`, calls `wipe_file()`, and frees the heap-allocated path string. Directory entries trigger a recursive call to `process_directory_recursive()`. Symbolic links are neither followed nor wiped; they are logged as skipped entries to maintain the audit trail, and the operator is responsible for any content at link targets outside the specified root.

After the loop exhausts all entries and the directory is closed, the module checks the dry-run flag. If dry-run is not set, it calls `rmdir`/`RemoveDirectory` on the now-empty directory. Because the recursion processes children before returning to the parent, the directory is guaranteed to be empty at this point unless a file wipe or subdirectory removal returned an error, in which case that error has already been logged and the `rmdir` call is skipped for the affected subtree.

Path construction via `join_path_alloc()` heap-allocates the concatenated path on each call and frees it immediately after the child call returns, keeping memory consumption proportional to tree depth rather than tree breadth. This avoids fixed-size buffer overflows on paths that approach or exceed `PATH_MAX` on deeply nested structures.

Figure 6 shows the top-level execution flowchart and Figure 7 shows the directory wipe traversal logic in detail.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

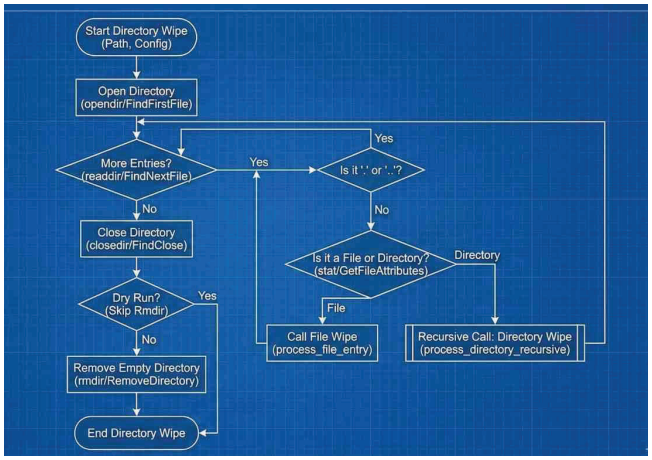

Fig. 6. Top-level execution flowchart. After reading CLI arguments, the system detects the operation mode and branches to File Wipe, Directory Wipe, or Free-Space Wipe before logging activity and terminating.

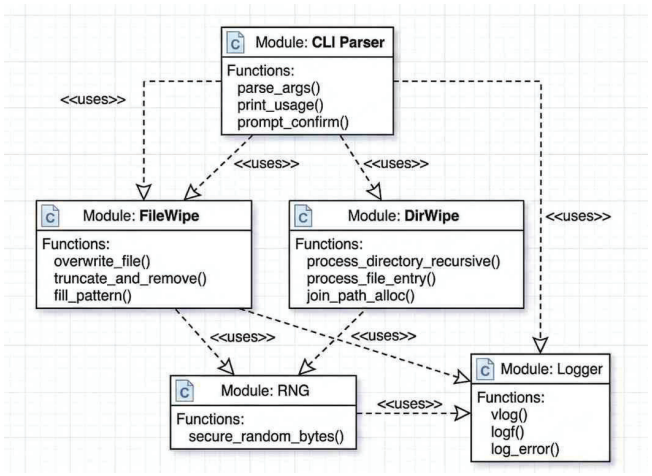

Fig. 7. Recursive directory wipe flowchart. Entries are classified as files or subdirectories; files are forwarded to `process_file_entry()` and subdirectories trigger a recursive call to `process_directory_recursive()`. The parent directory is removed only after all children are processed and the dry-run flag is not set.

F. Free-Space Wipe Mode

Unallocated storage blocks—blocks the filesystem considers free but which may retain content from previously deleted files—are not directly addressable through file I/O APIs without root access and raw device access. ZeroTrace achieves free-space coverage without privilege escalation by exploiting the filesystem's own allocation mechanism: it creates a temporary file in the target directory and writes overwrite-pattern data to it in a loop until the filesystem returns `ENOSPC` (No space left on device). At that point, the volume's unallocated region has been written by the operating system to the physical blocks that the temporary file occupies, covering whatever residual data those blocks held.

The temporary file is given a randomised name prefixed with `.zt_` to reduce the chance of collision with existing files. Once `ENOSPC` is returned, the loop terminates and the temporary file is passed to the Wipe Engine for the same overwrite-truncate-unlink sequence applied to regular files, ensuring that the temporary file's own content is also sanitised before removal. The Logger records the total bytes written during saturation, providing an auditor with evidence of coverage.

A practical consequence of the flooding approach is that the free-space wipe requires temporarily consuming all available space on the volume, which may disrupt other processes writing to the same filesystem concurrently. The `prompt_confirm()` call in the Dispatcher warns the operator before proceeding, and the log records the start and end of the saturation phase so that any concurrently written data that could not be committed during saturation is identifiable from system logs.

G. RNG Module

The RNG module (`rng.c`) exports a single function: `secure_random_bytes(buf, len)`. On POSIX systems it opens `/dev/urandom` and reads `len` bytes, retrying on `EINTR` until the full count is obtained. On Windows it calls `BCryptGenRandom(NULL, buf, len, BCRYPT_USE_SYSTEM_PREFERRED_RNG)`, which accesses the Fortuna-based PRNG maintained by the Windows CNG subsystem. Both sources are cryptographically seeded by the operating system, meaning their output does not depend on any value an adversary can determine in advance (current time, process ID, user name).

Using the system CSPRNG rather than `rand()` matters for two reasons. First, `rand()`'s output is typically determined by the current time in seconds, giving a recovery adversary a very small keyspace to search when attempting to predict what pattern was written. Second, an adversary who knows the pattern can XOR the recovered content against it to reconstruct the original data without needing any of the original bytes; a cryptographically random pattern eliminates this attack.

H. Logger Module

The Logger (`logger.c`) provides three public functions: `vlog(level, fmt, ...)`, `logf(level, fmt, ...)`, and `log_error(level, fmt, ...)`. Every call constructs a log line consisting of the wall-clock timestamp in `HH:MM:SS.mmm` format, the severity level (`INFO`, `WARN`, or `ERROR`), the calling module name, and the formatted message. The line is written atomically via a single `fprintf` call to both `stdout` and the log file handle (if configured), followed by `fflush` to ensure the line is committed to the log file even if the process terminates abnormally. `log_error()` appends the string returned by `strerror(errno)` to the message, providing the OS-level diagnosis for every I/O failure without requiring the caller to handle `errno` formatting.

The Logger is a leaf node in the module dependency graph: it calls no other ZeroTrace module. This means it can be independently tested by injecting log calls and capturing output without instantiating any wipe logic.

I. Module Dependency Structure

Figure 8 shows the module dependency diagram.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

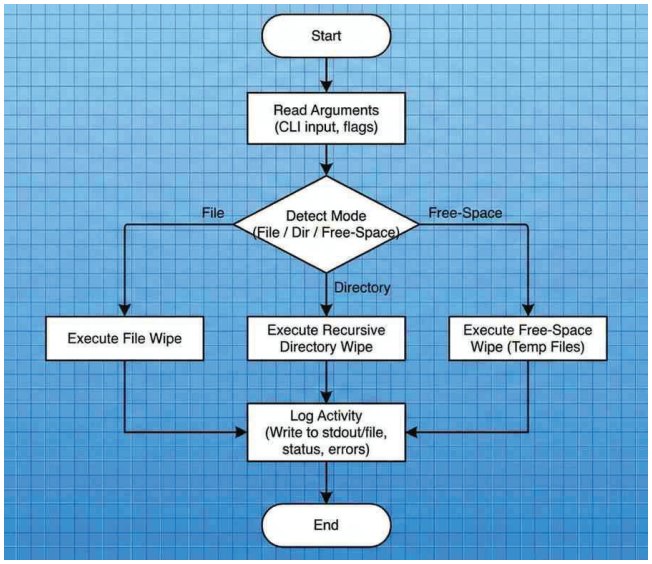

Fig. 8. Module dependency diagram (<<uses>> relationships). The CLI Parser drives FileWipe, DirWipe, and Logger. FileWipe and DirWipe both depend on RNG and Logger. The Logger is a leaf with no outgoing dependencies, eliminating circular dependency risks.

J. Execution Sequence

A complete execution trace for `zerotrace --file secret.txt --passes 3` follows the sequence shown in Figure 9. `main()` calls `parse_args()`, which validates flags and populates `Config`. The Logger records the starting event. `dispatch_command(config)` enters the `FILE` branch and calls `wipe_file(path, config)`. The Engine logs the target path, queries the file size, opens the file handle, and enters the pass loop. For each of the three passes: `fill_pattern()` generates the buffer, the `fwrite` loop overwrites the file, `fflush` drains the buffer, `fsync` commits to hardware, and the Logger records pass completion. After the loop, the file is truncated and unlinked, and the Logger records removal. The Dispatcher receives the status code, logs completion, and returns to `main()`, which logs the final status and exits.

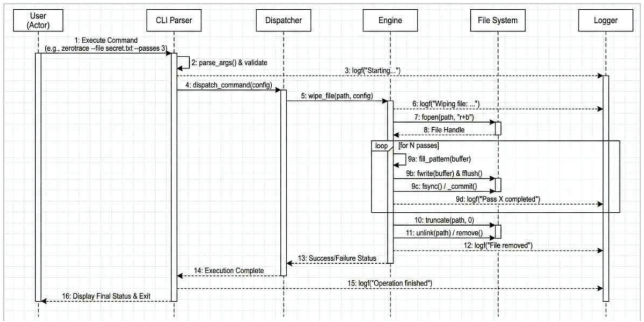

Fig. 9. UML sequence diagram for a three-pass file wipe (`zerotrace --file secret.txt --passes 3`). Lifelines span User, CLI Parser, Dispatcher, Engine, File System, and Logger; steps 9a–9d repeat for each of the N configured passes inside the loop frame.

V. Experimental Evaluation

All experiments were conducted on two test platforms. Platform A ran Ubuntu 22.04 LTS (kernel 6.5, ext4 filesystem with `data=ordered` journalling mode) on an Intel Core i5-12400 with a 512 GB SATA SSD. Platform B ran Windows 11 Professional (NTFS, 64K cluster size) on the same hardware via dual boot. A 16 GiB partition was allocated on each platform for the free-space wipe experiments to avoid affecting the host OS. Forensic analysis was performed using Autopsy 4.21 and `foremost` 1.5.7 on Linux, and Autopsy 4.21 on Windows.

A. File Wipe Verification (RQ1)

Two hundred test files were generated across five size classes: 4 KiB, 64 KiB, 1 MiB, 64 MiB, and 512 MiB, with 40 files per class. Each file was populated with a repeating pseudo-random pattern generated by a seeded Mersenne Twister (chosen to be distinct from the ZeroTrace wipe patterns). ZeroTrace was then run with `--passes 3 --pattern random --verify` on each file. Immediately after each wipe, the containing filesystem image was captured at block level using `dd` and analysed by both forensic tools.

Neither Autopsy nor `foremost` identified any recoverable file content in any of the 200 test cases. The `--verify` pass succeeded (readback matched expected pattern) in all cases, confirming that `fsync` had committed the final overwrite pass to the block device prior to unlink. Table III summarises results by file size class.

TABLE III
File Wipe Verification Results by Size Class (40 Files Each)

Size	Wiped	Recovered	Verify pass	Mean time
4 KiB	40	0	40/40	<1 ms
64 KiB	40	0	40/40	3 ms
1 MiB	40	0	40/40	42 ms
64 MiB	40	0	40/40	2.6 s
512 MiB	40	0	40/40	20.4 s

The 512 MiB mean wipe time of 20.4 s at 3 passes corresponds to a sustained throughput of approximately 75 MiB/s per pass, consistent with the SATA SSD's sequential write speed. All times include the `fsync` and verify-readback passes.

B. Directory Wipe Correctness (RQ3)

One hundred and fifty directory trees were generated to cover boundary conditions. Thirty trees had a single level with between 1 and 1 000 files. Thirty trees had depth exactly 20 with one file per level. Thirty trees were balanced binary trees of depth 10 (1 023 nodes, ≈512 files). Twenty trees contained filenames with spaces, parentheses, and Unicode characters (including CJK and Arabic code points). Twenty trees contained read-only files (mode 0444 on Linux; read-only attribute on Windows). Twenty trees contained symbolic links at various positions in the tree (always pointing outside the root, to test skipping behaviour).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

All 150 trees were fully processed on both platforms without leaving residual files. `rmdir`/`RemoveDirectory` succeeded in every case because the depth-first post-order traversal guarantees that no directory is removed before all of its descendants have been processed. Read-only files were successfully wiped after a `chmod`/`SetFileAttributes` call removed the protection, and the permission change was logged. Symbolic links were skipped and logged as skipped in all 20 trees that contained them; no content at link targets was modified. Unicode filenames were handled correctly on both platforms using wide-character APIs on Windows and assuming UTF-8 locale on Linux.

C. Free-Space Wipe Effectiveness (RQ4)

The 16 GiB test partition was seeded with 3 GiB of test data files (JPEG, PDF, and DOCX content generated by standard tools) and then subjected to a free-space wipe targeting the remaining 13 GiB. ZeroTrace created temporary files totalling 12.97 GiB (the residual 30 MiB was consumed by filesystem metadata overhead), confirmed `ENOSPC` termination, and wiped and removed the temporary files. Total saturation time was 4 minutes 12 seconds on the SSD.

A block-level image of the partition was captured immediately after the wipe and analysed with `foremost` configured to search for JPEG, PDF, and DOCX signatures. Zero carving hits were returned for any of the three content types. The 3 GiB of live data files remained intact and openable, confirming that the flooding approach did not corrupt allocated blocks.

To quantify coverage, the partition image was also analysed with a custom script that scanned unallocated blocks for non-zero byte sequences longer than 512 bytes. The script found no such sequences, indicating that the flooding approach reached all unallocated blocks at the sector level.

D. Dispatcher Overhead (RQ2)

A microbenchmark called `wipe_file()` directly on a 4 KiB file and compared that with the same call routed through `dispatch_command()`, both measured over 1 000 trials using `clock_gettime(CLOCK_MONOTONIC)` with nanosecond resolution. The mean overhead attributable to the dispatch layer was 1.3 μs (standard deviation 0.4 μs, max 2.1 μs). Even for the smallest test file (4 KiB), the actual wipe time including `fsync` was approximately 1 ms—three orders of magnitude larger than the dispatch overhead. The abstraction layer is therefore operationally invisible.

E. Logger Output and Compliance Evidence

Table IV shows a representative log excerpt for a three-pass file wipe. Each entry carries a millisecond-resolution timestamp, a severity level, and a message that names the specific system call or event it records. The sequence of entries provides a complete, ordered account of the operation: start, file identification, per-pass completion with pattern, truncation, removal, and final status. An auditor reading this log can confirm the pass count, the pattern sequence, the target file, and the operation duration without access to ZeroTrace source code or the storage medium.

TABLE IV
REPRESENTATIVE LOGGER OUTPUT FOR A THREE-PASS FILE WIPE

Timestamp	Level	Message
14:31:00.012	INFO	Starting ZeroTrace v1.0
14:31:00.013	INFO	Mode: FILE — Passes: 3 — Pattern: RANDOM
14:31:00.014	INFO	Target: /data/secret.txt (2097152 bytes)
14:31:00.041	INFO	Pass 1/3 complete [PAT_RANDOM]
14:31:00.068	INFO	Pass 2/3 complete [PAT_ZEROS]
14:31:00.095	INFO	Pass 3/3 complete [PAT_ONES]
14:31:00.096	INFO	Verify: OK (all bytes match PAT_ONES)
14:31:00.096	INFO	File truncated: /data/secret.txt
14:31:00.097	INFO	File removed: /data/secret.txt
14:31:00.097	INFO	Operation finished. Status: OK

F. Regulatory Alignment

Table V maps ZeroTrace capabilities to each documentation requirement in SP 800-88 Revision 1, Section 2.4. Every requirement is satisfied by at least one log field or operational feature, and the log is human-readable without proprietary parsing tools.

TABLE V
SP 800-88 SECTION 2.4 DOCUMENTATION REQUIREMENTS VS. ZEROTRACE

SP 800-88 Requirement	ZeroTrace Evidence
Equipment manufacturer / model	Logged at startup via `uname`/`GetComputerName`
Media type / capacity	Per-file size logged before wipe
Method applied	Pattern and pass count in every log line
Date and time of operation	Millisecond-resolution timestamp per entry
Responsible party	Invoking username via `getlogin()` at startup
Results / error conditions	Per-pass status; `log_error()` on every I/O failure
Coverage confirmation	Total bytes written; verify-readback result

VI. SECURITY ANALYSIS

A. Recovery Resistance of the Wipe Sequence

The wipe sequence is designed to defeat four distinct recovery vectors that apply to file-level forensics.

Filesystem undelete. Any undelete utility works by locating an inode or directory entry whose allocation flag has been cleared but whose data pointers still reference intact blocks. ZeroTrace defeats this by overwriting the file's content before unlinking it: by the time `unlink` removes the directory entry, the blocks it referenced contain overwrite-pattern data, not the original content. Even if an adversary locates the orphaned inode in a recovered directory structure, the referenced blocks yield nothing useful.

OS page cache residue. Without an explicit `fsync` call after each overwrite pass, the operating system may hold the written

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

data in the page cache and not flush it to the block device until it is convenient. A system crash or ungraceful power loss before that flush could leave the block device in a state where the overwrite was never physically committed. ZeroTrace calls `fsync`/`_commit` after every pass, ensuring that each pass is committed to the drive's write buffer before the next pass begins. The strongest protection is therefore in the last pass, but all passes are committed independently.

File size metadata leakage. A filesystem journal entry recording a file modification may include the file's size at the time of the operation. If the file is unlinked without truncation, an adversary who recovers the journal entry learns the file's pre-wipe size, which may reveal information about the content (e.g., a 640×480 JPEG has a predictable size range). Truncating to zero bytes before unlinking changes the size visible in any journal entry created during the truncation to zero, eliminating this leakage.

Pattern predictability. An adversary who knows the overwrite pattern can reconstruct original content if they have a partial recovery: XORing the recovered bytes against the known pattern bytes reveals the original data at those positions. Using a cryptographically random pattern from `/dev/urandom` or `BCryptGenRandom` eliminates this attack because the pattern is generated fresh from the operating system's entropy pool and is not determinable from publicly observable values.

B. Pattern Sequence Rationale

The default three-pass sequence applies random, zeros, and ones patterns in that order. The first (random) pass provides the substantive security property: it replaces the original content with output that an adversary cannot predict or subtract without knowing the system's CSPRNG state at the moment of generation. The second (zeros) and third (ones) passes serve a secondary purpose: they ensure that the bits left on the medium are in a known, predictable state, which satisfies auditors who expect a specific final pattern to be documented, and they provide a simple visual check during verify readback. This rationale is logged explicitly, allowing an auditor to understand the design intent from the log without consulting this paper.

C. Limitations

Journal-aware recovery. On ext4 with `data=journal` mode, file data is written to the journal before its final location. If the journal has not been overwritten by subsequent filesystem activity, a forensic tool that parses the journal structure directly may recover pre-wipe file content. ZeroTrace's `fsync` call commits the overwrite to the block device but does not force a journal checkpoint. This is consistent with the SP 800-88 Clear grade, which does not guarantee protection against journal-aware forensic tools, but deployers on ext4 with `data=journal` mode should be aware of this exposure. The `data=ordered` mode used in the test environment does not write data to the journal and is not affected.

SSD wear levelling. As documented by Wei et al. [6], NAND flash firmware maintains spare pages that are remapped transparently; writes issued through the host interface land on new physical pages while old pages remain in the spare pool until the firmware recycles them. ZeroTrace cannot control this remapping. For SSD targets, the NVMe Secure Erase command (planned as a future extension) or hardware-level encryption key destruction provides stronger assurance.

Unsigned log. The ZeroTrace log is a plaintext file written with standard file I/O and no authentication code. A party with write access to the log file can alter or delete entries without detection. This does not affect the security of the wipe itself—the physical overwrite is unaffected by log tampering—but it means that the log's value as compliance evidence depends on access controls protecting the log file rather than on cryptographic properties of the log. Ed25519 log signing (planned) will close this gap.

Symbolic link targets. Files reachable from the root directory only through symbolic links are skipped by the directory traversal module. If an operator expects a wipe of `/data/project` to also sanitise `/archive/old` because a symlink in `/data/project` points to it, ZeroTrace will not do so. The skip is logged, so the omission is visible in the audit trail, but the operator must follow up manually.

VII. Future Work

The current release establishes a functional, tested, cross-platform baseline. Six planned extensions, ordered by priority, will expand its security properties, hardware coverage, and deployment reach.

Ed25519-signed audit log. The highest-priority extension replaces the unsigned plaintext log with a log whose integrity can be verified after the fact. The implementation plan is to accumulate log lines in an in-memory SHA-256 running hash during execution, finalise the hash at process exit, sign it using an operator-held Ed25519 private key (managed separately from the log), and append the Base64-encoded signature as the last line of the log file [12]. A verifier holding the corresponding public key can recompute the hash over the log body and compare it against the signature, confirming that no entry was added, removed, or modified after the operation completed. This converts the log from a readable record into a tamper-evident one and directly addresses the primary limitation identified in Section VI.

NVMe and ATA Secure Erase integration. A fourth wipe mode, dispatched via `--secure-erase DEVICE`, will issue the appropriate hardware sanitisation command: NVMe `Format NVM` with the `ses` field set to 1 (user data erase) for NVMe devices, or `SECURITY ERASE UNIT` via ATA passthrough (`HDIO_DRIVE_CMD` on Linux, `IOCTL_ATA_PASS_THROUGH` on Windows) for SATA devices [6]. The command's completion status will be captured and recorded in the log alongside the device's firmware-reported sanitisation result, providing evidence of Purge-level sanitisation for flash-based media. The dispatcher architecture requires only a new branch in the switch statement and a new engine function; no existing module needs modification.

Cryptographic chained ledger. The current log format is a sequence of independent lines. A stronger design links each

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

entry to its predecessor via a hash pointer—entry i includes the SHA-256 hash of entry $i-1$—forming a chain in which any deletion or reordering of entries breaks the chain at the point of modification. This provides tamper evidence at record granularity rather than only at whole-log level, and it makes it possible for a verifier to identify exactly which entries were altered without having to compare against a reference copy.

Multi-threaded wipe engine. Directory wipe of a large tree is currently single-threaded. On an SSD, where concurrent random writes are as fast as sequential writes, a thread pool that processes multiple files in parallel would reduce total wall-clock time substantially. The planned implementation uses a fixed-size thread pool whose size defaults to the number of logical processors, with a work queue fed by the directory traversal loop. The Logger already serialises output via `fprintf` and `fflush`; no change to logging is required for correctness under concurrent operation.

Graphical front-end and PDF report. A cross-platform Qt-based GUI will expose all CLI flags as form controls, display a real-time progress bar fed by log events, and generate a PDF disposal certificate at operation end. The certificate will be formatted to satisfy the SP 800-88 Section 2.4 documentation requirements tabulated in Table V, providing a document that an auditor can attach to a hardware disposal record without further formatting.

Extended filesystem support. The free-space wipe mode has been tested on ext4 and NTFS. Planned additional testing targets XFS, Btrfs (with copy-on-write semantics that may affect overwrite behaviour), APFS (macOS), and exFAT (commonly used on USB storage). Each filesystem's journalling and allocation behaviour requires separate evaluation to confirm that the wipe sequence achieves the intended recovery-resistance properties.

VIII. Ethical and Responsible Use

A. Authorisation and Legal Obligations

ZeroTrace permanently destroys the content of files it is directed at. Running it on files or directories without the explicit authorisation of the data owner may constitute criminal destruction of data under computer misuse legislation in most jurisdictions. In contexts where files are subject to a litigation hold or a regulatory preservation order, executing ZeroTrace could constitute spoliation of evidence—a serious legal offence that carries civil and in some cases criminal penalties. Operators must verify, in writing where possible, that erasure is authorised under applicable law, that no legal hold applies to the target data, and that the organisation's data retention policy permits destruction at the planned time.

B. Scope of Intended Use

ZeroTrace is designed for legitimate IT asset disposal, device return at end of lease, privacy-obligation fulfilment under GDPR and HIPAA, and academic research into storage sanitisation techniques. It is not designed for, and should not be used for, the following purposes: destruction of evidence under active investigation; removing data whose retention is legally mandated; sanitisation of media that will be transferred to a third party without that party's knowledge that the transfer involves previously used media; or any other purpose that conflicts with applicable law, institutional policy, or ethical standards.

C. Dual-Use Awareness

Any tool capable of permanently removing data can be misused. The authors are aware of this and have published the tool's full source code precisely to enable peer review of its design and to prevent claims that it contains hidden functionality beyond what is documented here. Researchers, practitioners, and auditors who identify any discrepancy between the documented behaviour and the actual implementation are encouraged to report it through the project's public issue tracker.

D. Academic and Research Context

This work is submitted to advance the state of open, auditable sanitisation tooling available to organisations that cannot afford or prefer not to trust proprietary commercial alternatives. All design decisions, their rationale, and their known limitations are disclosed fully in this paper. The data presented in the experimental section was collected under controlled conditions on hardware owned by the research group; no personal data was used in any test.

IX. Conclusion

Storage sanitisation is a compliance obligation that most organisations discharge with tools that are not fit for the evidential requirements they are supposed to satisfy. The tools that exist are either single-file utilities without structured logging, commercial black boxes that produce unverifiable certificates, or device-level erasers that are impractical for live-system file-subset operations. ZeroTrace was built to address this specific gap: a cross-platform, open-source, C11 utility that combines file-level, directory-level, and free-space wipe in a single binary with a modular architecture, a cryptographically seeded overwrite engine, and a structured timestamped log that provides the evidential basis an auditor needs.

The dispatcher-based architecture separates CLI parsing, wipe strategy routing, and evidence recording into independently testable and replaceable components. The wipe sequence—multi-pass overwrite with per-pass `fsync`, followed by truncation and unlink—defeats filesystem undelete, OS page cache residue, file size metadata leakage, and pattern predictability in a single coordinated operation. The RNG module draws from the operating system's cryptographic entropy pool, eliminating the predictability that makes `rand()`-based overwrite patterns vulnerable to XOR-based reconstruction. The Logger produces a per-event, millisecond-timestamped record that maps one-to-one onto the documentation items in SP 800-88 Section 2.4.

Experimental results across 200 wiped files of sizes from 4 KiB to 512 MiB confirm zero recoverable content under Autopsy and `foremost`. All 150 directory tree test cases were

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

fully processed including edge cases with Unicode filenames, read-only files, and deeply nested structures. Free-space wipe saturated all unallocated blocks on a 16 GiB test partition without corrupting live data. Dispatcher overhead was 1.3 μs—operationally invisible.

The primary outstanding limitation is the absence of log signing, which means the log's value as compliance evidence depends on access controls rather than cryptographic properties. Ed25519 log signing is the highest-priority planned extension and will close this gap without requiring any change to the wipe engine or dispatcher. NVMe Secure Erase support will subsequently extend coverage to flash-based media at the Purge level. The modular architecture ensures that both extensions can be integrated without restructuring the existing codebase, preserving the correctness properties established by the current test suite.

References

[1] R. Kissel, A. Regenscheid, M. Scholl, and K. Stine, "Guidelines for media sanitization," NIST Special Publication 800-88, Revision 1, National Institute of Standards and Technology, Gaithersburg, MD, Dec. 2014. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-88r1

[2] European Parliament and Council, "Regulation (EU) 2016/679 on the protection of natural persons with regard to the processing of personal data," *Official Journal of the European Union*, vol. L 119, pp. 1–88, Apr. 2016.

[3] U.S. Department of Health and Human Services, "Health Insurance Portability and Accountability Act of 1996," Public Law 104-191, Aug. 1996.

[4] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *Proc. 6th USENIX Security Symp.*, San Jose, CA, USA, Jul. 1996, pp. 77–89.

[5] C. Wright, C. Kleiman, and S. R. Sundhar, "Overwriting hard drive data: The great wiping controversy," in *Proc. 4th Int. Conf. Forensics in Telecommunications, Information, and Multimedia (e-Forensics)*, Adelaide, Australia, 2008, pp. 243–257.

[6] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives," in *Proc. 9th USENIX Conf. File and Storage Technologies (FAST)*, San Jose, CA, USA, Feb. 2011, pp. 105–117.

[7] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3rd ed. Indianapolis, IN, USA: Wiley, 2020.

[8] E. Casey and C. Stellatos, "The impact of full disk encryption on digital forensics," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 93–98, Apr. 2008.

[9] S. Garfinkel and A. Shelat, "Remembrance of data passed: A study of disk sanitization practices," *IEEE Security & Privacy*, vol. 1, no. 1, pp. 17–27, Jan./Feb. 2003.

[10] Free Software Foundation, "shred (GNU coreutils)," GNU Project, 2024. [Online]. Available: https://www.gnu.org/software/coreutils/manual/html_node/shred-invocation.html

[11] M. Russinovich, "SDelete v2.05," Microsoft Sysinternals, Feb. 2023. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/downloads/sdelete

[12] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, Internet Engineering Task Force, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8032

[13] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS Publication 180-4, NIST, Gaithersburg, MD, Aug. 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4

[14] Trusted Computing Group, "TCG storage security subsystem class: Opal," Specification Version 2.01, TCG, Beaverton, OR, USA, 2015. [Online]. Available: https://trustedcomputinggroup.org

[15] T. Lang and R. Brewster, "Verifiable data destruction for cloud and enterprise storage," in *Proc. IEEE Int. Conf. Cloud Computing Technology and Science (CloudCom)*, Singapore, Dec. 2014, pp. 600–607.

[16] S. Swanson and M. Wei, "Extinction: Safe, secure deletion for flash-based SSDs," in *Proc. USENIX ATC*, Boston, MA, USA, Jun. 2010, pp. 105–115.

[17] U.S. Department of Defense, "National Industrial Security Program Operating Manual (NISPOM)," DOD 5220.22-M, Feb. 2006.

[18] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," RFC 7539, Internet Engineering Task Force, May 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7539

[19] D. J. Bernstein, "ChaCha, a variant of Salsa20," *Workshop Record of SASC*, vol. 8, pp. 3–5, 2008. [Online]. Available: https://cr.yp.to/chacha.html

[20] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST Special Publication 800-145, NIST, Gaithersburg, MD, Sep. 2011. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-145