Shred Tool in Go

Technical Implementation Report

Yusuf Yıldız

January 4, 2026

Contents

Abstract

This report documents the design and implementation of a Go helper function `Shred(path)` that overwrites a file **three times** with random bytes and then deletes it. The implementation treats the input as raw bytes (supporting any file contents), overwrites in fixed-size chunks for memory efficiency, and uses `fsync` after each pass to reduce the chance of buffered data remaining unwritten. A unit test suite covers both successful and failure scenarios (regular files, empty files, missing paths, directories, permission errors, symlinks, deterministic overwrite via a fake random source, and a 10 MiB file). Finally, `strace` is used as an external validation method to show that the intended system calls (`getrandom`, `write`, `fsync`, `unlinkat`, etc.) are invoked in the expected order and frequency.

1 Introduction

Securely deleting files is a recurring requirement in systems programming, incident response, and privacy-focused workflows. Simply removing a file (e.g., `rm`) typically only unlinks its directory entry; the underlying data blocks may remain recoverable until overwritten by the file system. A common mitigation is to overwrite the file contents with random data multiple times before deleting it.

This project implements `Shred(path)` in Go. The function overwrites the target file three times with random data and deletes it afterwards, regardless of the file type of its contents (text, binary, images, etc.). The report also discusses test cases and provides `strace`-based evidence of correctness. You can reach the project via this LINK.

2 Assignment Requirements

The assignment requirements are:

- Implement `Shred(path)` that overwrites the given file 3 times with random data.
- Delete the file afterwards.
- Support any file contents (treat it as bytes).
- Provide possible test cases (full coverage is a bonus).
- Briefly discuss use cases, advantages, and drawbacks of this approach.

3 Design and Implementation

3.1 High-level Algorithm

Given a file path, the function performs:

1. Validate that the path exists and refers to a regular file.
2. Open the file for writing (`O_WRONLY`).
3. Repeat 3 times:

   (a) Seek to offset 0.
   (b) Overwrite the entire file with random bytes in chunks.
   (c) `fsync` the file descriptor.

4. Close the file descriptor.

5. Remove the file (`unlink`).

3.2 Chunked Overwrite and Randomness Source

Overwriting is done in `chunkSize = 64 KiB` blocks to keep memory usage constant even for large files. Random bytes are read from `crypto/rand.Reader` through a package-level variable (`randomSource`) so that tests can inject a deterministic reader.

3.3 Durability Considerations

After each overwrite pass, `file.Sync()` is called (translated to `fsync(2)`), which asks the kernel to flush modified file data and metadata to the storage device. This improves durability but does not guarantee irrecoverability on all storage backends (see Section 6.3).

4 Testing

The test suite is implemented in `shred_test.go`. It combines table-driven tests for common error handling with targeted tests for special cases.

4.1 Implemented Unit Tests

Table 1 summarizes the implemented tests and their intent.

Test	**What it validates**
TestShred/RegularFileRemoved	File is overwritten (3 passes) and removed.
TestShred/EmptyFileRemoved	Empty file still results in 3 passes (seek+sync) and removal.
TestShred/NonExistingFile	Missing path returns an error.
TestShred/DirectoryIsNotRegularFile	Directories are rejected as non-regular files.
TestShred/ReadOnlyFile	Lack of write permission causes an error (no overwrite).
TestOverwriteDeterministicRandom	overwrite uses injected fake random bytes deterministically.
TestShredSymlink	Shredding a symlink path overwrites the target and removes the link.
TestShredBigFile	A 10 MiB file is overwritten in chunks and removed.

Table 1: Implemented test cases in `shred_test.go`.

4.2 Additional Test Cases (Not Implemented)

The following cases are relevant for fuller coverage:

- **Concurrent access:** another process holding the file open for reading/writing during shredding.
- **Hard links:** shredding one link does not remove other directory entries pointing to the same inode.
- **Special files:** FIFOs, sockets, device nodes, and memory-mapped files.
- **Sparse files:** ensuring holes are overwritten (may require explicit allocation).
- **Filesystem-specific behavior:** copy-on-write filesystems (e.g., btrfs, ZFS), journaling, snapshots, and SSD wear-leveling.
- **Interrupted execution:** power loss or process crash mid-pass (partial overwrite).
- **Very large files:** multi-gigabyte files to stress chunking and runtime behavior.

5 System Call Validation with `strace`

To externally validate behavior, `go test` was executed under `strace`. The goal is to confirm that the implementation actually triggers:

- random byte generation (`getrandom`)
- overwriting (`write`)
- durability hints (`fsync`)
- deletion (`unlinkat`)

5.1 Note on `getrandom` vs `write` Byte Formatting

In `strace`, buffers are printed with escaping. The same byte sequence can appear different: `getrandom` may show `\x..` escapes, while `write` may show octal escapes (e.g., `\271`) and printable characters. This is only a formatting difference; the underlying bytes are identical.

5.2 Regular File: 3 Passes + Remove

For a regular file of size 11 bytes, the overwrite loop runs 3 times, each time issuing one `getrandom(11)` and one `write(11)` followed by `fsync`, and finally `unlinkat`. Selected excerpt:

Listing 1: strace excerpt: RegularFileRemoved shows exactly 3 overwrite passes and deletion.

```
120146 openat(AT_FDCWD, "/tmp/TestShredRegularFileRemoved533947944/001/
   file.txt", O_WRONLY|O_CLOEXEC) = 6
120146 lseek(6, 0, SEEK_SET)          = 0
```

```
120146 getrandom("\x26\x41\xb9\xb6\x39\x9a\x4c\x12\x6d\xd8\xa0", 11, 0
    <unfinished ...>
120146 <... getrandom resumed>)        = 11
120146 write(6, "&A\271\2669\232L\22m\330\240", 11) = 11
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)            = 0
120146 lseek(6, 0, SEEK_SET)           = 0
120146 getrandom("\x06\xca\x55\xdb\xe8\xa2\x04\xa5\x69\xce\xb6", 11, 0)
     = 11
120146 write(6, "\6\312U\333\350\242\4\245i\316\266", 11) = 11
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)            = 0
120146 lseek(6, 0, SEEK_SET)           = 0
120146 getrandom("\x10\x22\xfe\xf8\xce\x53\x07\x15\x50\x09\x2a", 11, 0
    <unfinished ...>
120146 <... getrandom resumed>)        = 11
120146 write(6, "\20\"\376\370\316S\7\25P\t*", 11 <unfinished ...>
120146 <... write resumed>)            = 11
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)            = 0
120146 close(6)                        = 0
120146 unlinkat(AT_FDCWD, "/tmp/TestShredRegularFileRemoved533947944
    /001/file.txt", 0 <unfinished ...>
120146 <... unlinkat resumed>)         = 0
```

From this excerpt we observe:

- 3 calls to `getrandom` (each for 11 bytes)
- 3 calls to `write` on the file descriptor (each writing 11 bytes)
- 3 calls to `fsync`
- a final `unlinkat` removing the file

5.3 Empty File: No Data Writes, Still 3 Sync Passes + Remove

An empty file has size 0, so the data-overwrite inner loop performs no `getrandom`/`write`, but the code still does 3 passes of `seek`+`fsync`, and removes the file:

Listing 2: strace excerpt: EmptyFileRemoved performs 3 passes (seek+fsync) and then unlinks.

```
120146 openat(AT_FDCWD, "/tmp/TestShredEmptyFileRemoved1795746715/001/
    empty.txt", O_WRONLY|O_CLOEXEC) = 6
120146 lseek(6, 0, SEEK_SET)           = 0
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)            = 0
120146 lseek(6, 0, SEEK_SET)           = 0
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)            = 0
120146 lseek(6, 0, SEEK_SET)           = 0
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)            = 0
120146 close(6)                        = 0
120146 unlinkat(AT_FDCWD, "/tmp/TestShredEmptyFileRemoved1795746715
    /001/empty.txt", 0 <unfinished ...>
120146 <... unlinkat resumed>)         = 0
```

5.4 Non-existing Path and Directory Rejection

The following `newfstatat` calls show the failure modes:

Listing 3: strace excerpt: NonExistingFile returns ENOENT.

```
120146 newfstatat(AT_FDCWD, "/tmp/TestShredNonExistingFile2068087564
    /001/does-not-exist", 0xc00025ee28, 0) = -1 ENOENT (No such file or
    directory)
```

Listing 4: strace excerpt: DirectoryIsNotRegularFile detects S_IFDIR.

```
120146 newfstatat(AT_FDCWD, "/tmp/
    TestShredDirectoryIsNotRegularFile2728739365/001/subdir", {st_mode=
    S_IFDIR|0755, st_size=4096, ...}, 0) = 0
```

5.5 Read-only File: Open Fails with EACCES

Attempting to open a read-only file for writing fails (no overwrite happens):

Listing 5: strace excerpt: ReadOnlyFile shows EACCES on open for write.

```
120146 newfstatat(AT_FDCWD, "/tmp/TestShredReadOnlyFile1570036332/001/
    readonly.txt", {st_mode=S_IFREG|0400, st_size=9, ...}, 0) = 0
120146 openat(AT_FDCWD, "/tmp/TestShredReadOnlyFile1570036332/001/
    readonly.txt", O_WRONLY|O_CLOEXEC) = -1 EACCES (Permission denied)
```

5.6 Deterministic Overwrite via Fake Random Source

To test correctness of the overwrite logic independently from OS randomness, a fake reader is injected (0xAA = 252 octal). This eliminates `getrandom` syscalls and produces a predictable byte pattern. The excerpt below shows a single overwrite pass of 4096 bytes followed by `fsync`:

Listing 6: strace excerpt: Deterministic overwrite writes exactly 4096 bytes and fsyncs.

```
120146 openat(AT_FDCWD, "/tmp/
    TestOverwriteDeterministicRandom4112979404/001/data.bin", O_WRONLY|
    O_CLOEXEC) = 6
120146 lseek(6, 0, SEEK_SET)            = 0
120146 write(6, "
    \252\252\252\252\252\252\252\252\252\252\252\252\252\252\252\252"
    ..., 4096) = 4096
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)             = 0
120146 close(6)                         = 0
```

5.7 Symlink Shred: 3 Passes on Link Path + Unlink Link

In the symlink test, `Shred(linkPath)` overwrites through the symlink (affecting the target) and then unlinks the symlink itself. Because a fake random source is used here as well (0xBB = 273 octal), there is no `getrandom`; instead we observe 3 writes of 18 bytes and `unlinkat` on the link path:

Listing 7: strace excerpt: ShredSymlink shows 3 overwrite passes (18 bytes) and unlinkat on link.

```
120146 openat(AT_FDCWD, "/tmp/TestShredSymlink771919607/001/link.txt", O_WRONLY|O_CLOEXEC) = 6
120146 lseek(6, 0, SEEK_SET)            = 0
120146 write(6, "\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273", 18) = 18
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)             = 0
120146 lseek(6, 0, SEEK_SET)            = 0
120146 write(6, "\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273", 18) = 18
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)             = 0
120146 lseek(6, 0, SEEK_SET)            = 0
120146 write(6, "\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273\273", 18) = 18
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)             = 0
120146 close(6)                         = 0
120146 unlinkat(AT_FDCWD, "/tmp/TestShredSymlink771919607/001/link.txt", 0 <unfinished ...>
120146 <... unlinkat resumed>)          = 0
```

5.8 Big File: Chunked Overwrite (10 MiB)

The 10 MiB test file is overwritten in `64 KiB` chunks. From parsing the full `strace` output for this test run, the overwrite phase performs:

- **480** `getrandom` calls and **480** `write` calls (3 passes $\times$ 160 chunks/pass)
- **3** `fsync` calls (one after each pass)
- a final `unlinkat`

A short excerpt (showing the start of pass 1, pass boundaries via `fsync`+`lseek`, and final deletion) is:

Listing 8: strace excerpt: BigFile shows chunked overwrite pattern and 3-pass boundaries.

```
120146 openat(AT_FDCWD, "/tmp/TestShredBigFile2635864547/001/bigfile.bin", O_WRONLY|O_CLOEXEC <unfinished ...>
120146 <... openat resumed>)            = 6
120146 lseek(6, 0, SEEK_SET <unfinished ...>
120146 <... lseek resumed>)             = 0
120146 getrandom(0xc000280000, 65536, 0 <unfinished ...>
120146 <... getrandom resumed>)         = 4096
120146 write(6, "K\377\213Y\374e\244\347\332\257\331c\3200\252\27"..., 65536 <unfinished ...>
120146 <... write resumed>)             = 65536
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)             = 0
120146 lseek(6, 0, SEEK_SET <unfinished ...>
120146 <... lseek resumed>)           = 0
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)           = 0
120146 lseek(6, 0, SEEK_SET <unfinished ...>
120146 <... lseek resumed>)           = 0
120146 fsync(6 <unfinished ...>
120146 <... fsync resumed>)           = 0
120146 unlinkat(AT_FDCWD, "/tmp/TestShredBigFile2635864547/001/bigfile.
    bin", 0 <unfinished ...>
120146 <... unlinkat resumed>)        = 0
```

6 Use Cases, Advantages, and Limitations

6.1 Use Cases

Typical uses include:

- Removing sensitive temporary artifacts (API keys, credentials, decrypted documents).
- Cleaning build and CI caches that may contain secrets.
- Defensive deletion in tooling that handles personal data.

6.2 Advantages

- Simple API, easy to integrate in scripts and CLIs.
- Works for arbitrary file contents (operates on bytes).
- Chunked design is memory-efficient for large files.
- `fsync` reduces the risk of data staying only in page cache.

6.3 Limitations and Drawbacks

Overwriting a file is not a universal guarantee of secure deletion:

- **SSD wear leveling:** On SSD/flash devices the Flash Translation Layer (FTL) may redirect writes to new physical blocks and keep old blocks until garbage collection, so previous data can persist even after multiple overwrite passes.
- **Journaling / snapshots / CoW filesystems:** Journaled and copy-on-write filesystems may write new versions to new blocks (and/or temporarily duplicate data in a journal). Snapshots can still reference old blocks, preventing them from being reclaimed.
- **Hard links:** `unlink` removes only one directory entry; if other hard links exist, the inode remains reachable. Overwriting affects the inode's content, but "deletion" is not complete unless all links are removed.

- **Crash mid-pass:** If the program or system crashes during shredding, the file may be only partially overwritten, leaving recoverable fragments. `fsync` reduces this risk but cannot eliminate it.
- **Permissions:** Overwriting needs write permission on the file; deletion needs write permission on the parent directory. If either is missing (or blocked by ACL/MAC policies), shredding may fail or be incomplete.

For higher assurance, alternatives include full-disk encryption (so deletion reduces to key erasure), filesystem-specific secure-delete features (when available), or hardware-backed secure erase.

7 Conclusion

The implemented `Shred(path)` function satisfies the assignment requirements by overwriting a regular file three times with random data and deleting it afterwards. The unit tests validate both success and failure scenarios, and `strace` evidence confirms the expected system call behavior (`getrandom`+`write` in three passes, `fsync` after each pass, and `unlinkat` at the end). While the approach is useful as a best-effort mitigation, its security guarantees depend heavily on filesystem and storage characteristics.