ULocker: A Streaming, Re-keyable Encrypted Container for Removable Media, and What Its Framing Still Leaks

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com
https://github.com/yxpil/ULocker

Abstract

Full-disk encryption protects a USB drive only where a filesystem filter driver can be installed, and the ciphertext still exposes file names, directory structure and exact sizes to anyone who reads the raw device. ULocker is a container format that moves the unit of protection from the block device to a single file: the archive carries an encrypted index, so names and the directory tree are not readable without the password, and file content is written as independently authenticated 1 MiB chunks. A password-derived key unwraps an index key, which in turn unwraps a random data key, so a password change rewrites the header rather than the payload. We measure the format rather than assert its properties. The implementation passes its 212-test suite; a 24-case tamper matrix is detected in 19 cases, and the five misses are reported with their consequences (most are benign, one is a rollback with no freshness counter). A raw-byte audit finds 0 of 11 distinctive file-name, directory-name and content tokens in the container, but reconstructs the *number* of records and a size profile from the public 16-byte framing alone; eight files collapse into six inferred groups because two sizes are exact multiples of the chunk size. Password rotation takes 105–108 ms for containers from 8 MiB to 1 GiB, changes 388–399 bytes of header, rewrites 69,632 bytes, and leaves the data zone byte-identical. Create and extract throughput are 130–211 and 222–261 MiB/s on one host; the Python process's traced allocation stays at 6.2–7.2 MiB for both an 8 MiB and a 256 MiB container, which supports the streaming claim, while peak working set is dominated by the 64 MiB Argon2id cost. The contribution is the measured boundary: what the framing hides, what it cannot hide, and which attacks the format is not designed to stop.

Artifact. The container specification (`docs/FORMAT.md`), the Python implementation, the benchmark harness and every raw result file are open source at https://github.com/yxpil/ULocker. The repository is read-only input: the undetected mutations in Section 5.5 are findings, not patches. All numbers trace to `scripts/results/*.json` and `*.csv`, produced on the host recorded in Section 5.1.

1 Introduction

A USB drive is lost, stolen, or plugged into a computer that its owner does not control. Full-disk encryption answers the first two cases, but it has two structural costs. It requires a filesystem filter driver, which in practice means administrator rights and a driver installed on the machine that will read the drive; and even when it works, the ciphertext at the block layer reveals file names, directory structure, and exact file sizes to anyone who can read the raw device NIST (2007); Fruhwirth (2005). A container file makes the opposite trade: the user must explicitly unlock and extract, but the artifact is one opaque file that any machine can copy, and the metadata can be encrypted along with the payload.

ULocker is a Python implementation of that container design for removable media. Its stated goal is that "file names and directory structure are ciphertext too": a `strings` search over a locked container should return neither a file name nor a fragment of content. It also binds a container to a volume serial number, and it supports changing the password in constant time with respect to payload size. This paper asks which of those properties hold under measurement, and which do not.

The answer is more interesting than a feature list. The format does hide names and content: an audit of eleven distinctive tokens placed in file names, directory names and payloads finds none of them in the raw bytes, in a UTF-16LE interpretation, or in printable runs. But the format does not hide how many chunks it contains: the data zone is a sequence of records, each with a 12-byte nonce and a 4-byte length, so the record count and a size profile are recoverable without any key. Eight files were reported as six groups because two file sizes are exact multiples of the 1 MiB chunk size and therefore leave no short record at the boundary. This is a real, if bounded, leak, and we report it as such.

The tamper matrix is similarly mixed. Nineteen of twenty-four single-byte or structural mutations are detected by at least one public entry point. Four of the five misses are either harmless or require the key: flipping a byte of the zero padding, appending garbage after the last record, and reordering an index that the attacker has already re-sealed with the key do not affect extraction; selective extraction of one healthy file from a container with a corrupt sibling is a deliberate design

choice. The fifth miss matters: replacing a container with an older snapshot of itself is accepted, because the format has no monotonic counter or external freshness anchor. We state that plainly instead of claiming authenticity of the container as a whole.

Contributions.

1. *A container format with an authenticated, encrypted index.* Names, directory structure, modes, modification times, sizes, per-file SHA-256 digests and the chunk table live in a zlib-compressed JSON index encrypted with AES-256-GCM; the plaintext header before the index ciphertext is authenticated as associated data, so a modified header fails before any parse (Section 3).
2. *Envelope re-keying.* Argon2id derives a key-encryption key; HKDF derives an index key; the index carries a random 256-bit data key. A password change re-wraps the index only: 69,632 bytes are rewritten and the data zone is byte-identical, for containers from 8 MiB to 1 GiB (Section 5.7).
3. *Chunk binding.* Each 1 MiB chunk has a fresh 96-bit nonce and uses the file's 16-byte chunk identifier concatenated with the 64-bit chunk sequence as associated data, so a chunk cannot be moved between files, containers, or positions without an authentication failure. The matrix exercises exactly these attacks (Section 5.5).
4. *A measured boundary.* We report 212 passing tests, 19/24 detected mutations, the five misses, 0/11 metadata tokens found, the record-count and size-profile leak, constant-cost rotation, 130–261 MiB/s throughput, and 6.2–7.2 MiB of traced allocation independent of container size (Section 5).
5. *Explicit limits.* The drive binding is a policy control, not a cryptographic one; the format has no rollback protection; the number of entries and coarse sizes leak; and the default Argon2id parameters are below the RFC 9106 recommendation. Section 6 states each limit and its consequence.

2 Threat model and goals

Assets. File content, file names, directory structure, the number of files, their approximate sizes, and the password.

Adversary. We consider an adversary who obtains the container file and can read and modify it offline, but who does not know the password and does not observe the machine while it is unlocked. This covers a lost or stolen drive, a copy taken from a backup, and a cloud-storage provider. We explicitly exclude an adversary with code execution on the unlocked host, a keylogger, a cold-boot attacker who reads RAM after power loss Halderman et al. (2008), and a coercive adversary; Section 6 explains why.

Goals. (G1) Confidentiality of content, names and the directory tree against an offline adversary without the password.

Table 1: Header layout. *AAD* marks bytes covered as associated data by the index's AES-256-GCM invocation; the index ciphertext itself is the payload. The zero padding is not authenticated.

Field	Meaning	Bytes
`magic`	ASCII `ULOCKERV`	8
`version`	format version, currently 1	1
`kdf_id`	1 = Argon2id, 2 = scrypt	1
`flags`	bit 0 drive-bound, bit 1 zlib	2
`kdf_params_len`	length L of the JSON object	2
`kdf_params`	salt and KDF cost, UTF-8 JSON	L
`data_offset`	absolute start of data zone	8
`index_nonce`	GCM nonce for the index	12
`index_ct_len`	length M of index ciphertext	4
`index_ct`	index ciphertext + 16-byte tag	M
zero padding	to `data_offset`	variable

(G2) Integrity of each chunk and of the index: any modification is detected before the modified bytes are used, except for the cases enumerated in Section 5.5. (G3) Streaming: memory use must not grow with container size. (G4) A password change must not rewrite the payload. (G5) The container should be a single portable file that can be opened on a machine with no driver, using only a Python runtime or a packaged executable.

Non-goals. ULocker does not attempt to hide the fact that a container exists, the total ciphertext size, or the number of records. It does not provide plausible deniability, a hidden volume, multi-user access control, or remote revocation. It does not claim to make deleted plaintext unrecoverable on flash media. The container is a carrier, not a channel: no claim is made about its distinguishability from other files in the watermarking sense discussed for steganographic carriers Petitcolas (2000).

3 Container format

3.1 Layout

The file is a header followed by a data zone. All integers are big-endian. Table 1 lists the header fields in order. The header is variable-length because the KDF parameters are a JSON object and the encrypted index has variable length; the absolute start of the data zone is stored in the header and must be a multiple of 4096. The gap between the end of the index ciphertext and that offset is zero padding, reserved so that a growing index can be rewritten in place without moving the data zone.

The index is a JSON object with a chunk size (default 1 MiB) and a list of entries. Each entry has eight keys: `name`, `dir`, `mode`, `mtime`, `size`, `sha256`, `offset` and `chunk`. The last is a random 16-byte identifier, stored as hex, that is unique to the file and used as chunk associated data. The whole object is zlib-compressed before encryption. In the format probe of Section 5.3, four entries produce 627 bytes of

index ciphertext and a 750-byte header inside a 69,632-byte reservation.

3.2 Key hierarchy

The password is processed with Argon2id Biryukov et al. (2016); IETF (2021) to produce a 32-byte key-encryption key (KEK). HKDF-SHA256 IETF (2010) with the fixed info string `ulocker/v1/index` derives the index key. The index ciphertext is decrypted with AES-256-GCM NIST (2001); Dworkin (2007); IETF (2008) under that key and the nonce in the header. The decrypted index contains a random 32-byte data key, which is used with a fresh nonce per chunk. The data key never appears in plaintext outside the decrypted index.

This indirection is what makes goal (G4) achievable. A password change derives a new KEK from the new password and a new salt, re-encrypts the small index under a new index key, and writes a new header. The data key, and therefore every payload chunk, is unchanged. The alternative—deriving the data key directly from the password—would require re-encrypting the whole container. The construction is the classical split of a key-encryption key from a data key described in the applied-cryptography literature Menezes et al. (1996), with the index playing the role of the key-derivation material.

3.3 Chunk records

A record is a 12-byte nonce, a 4-byte ciphertext length, and the ciphertext including a 16-byte GCM tag. The plaintext is split at 1 MiB boundaries; a file whose size is an exact multiple of the chunk size has no trailing short record. The associated data for chunk i of file f is $chunk_id_f \,\|\, \mathrm{u64be}(i)$. This binds a ciphertext to both its file and its position. Without the file identifier, a chunk could be transplanted between two files with the same key; without the sequence number, chunks within a file could be reordered or duplicated. The tamper matrix in Section 5.5 tests both.

3.4 Header authentication and the reserve

The AAD for the index ciphertext is every byte from offset 0 through the end of `index_ct_len`. This covers the magic, version, KDF identifier, flags, the KDF parameters (including the salt), the data offset, the nonce, and the ciphertext length. A modified KDF parameter or a moved data offset therefore fails GCM verification before the index is parsed; in the implementation this surfaces as a wrong-password error, which we note as a usability defect in Section 5.5.

The padding is outside the AAD, because it is written after the index and is not part of the authenticated message. Flipping a padding byte is consequently undetected. The reserve is sized at container creation from the index length and grown by relocation: when an append would make the header exceed `data_offset`, the implementation rewrites the header at a larger aligned offset and copies the data zone. Section 5.4 measures the growth rate and the relocation point.

4 Implementation

The reference implementation is Python 3, with `cryptography` for AES-GCM and HKDF and `argon2-cffi` for Argon2id. It exposes a command-line interface, a library API, and an optional PyQt6 GUI; the measurements in this paper exercise the library and CLI, not the GUI. The command set covers create, open (extract), list, verify, change-password, and append. A container is created from a directory tree; extraction reconstructs directories, modes and modification times, and verifies each file's SHA-256.

The implementation makes three choices worth naming. First, it writes the header with a fixed-size reservation and relocates rather than re-writing the data zone on every append. Second, it uses a fresh random nonce per chunk rather than a counter; at 1 MiB per chunk, the probability of a repeated 96-bit nonce within one container is negligible, and a counter would require persisting state that a copied file does not have. Third, it maps a failed GCM tag check on the index to the same exception as a wrong password, which avoids an oracle but conflates tampering with a typo.

5 Evaluation

5.1 Environment and method

All measurements were taken on one host: Windows 11 (10.0.26300), Intel processor with 18 logical and 14 physical cores, 16 GB RAM, CPython 3.13.14, `cryptography` 50.0.2, `argon2-cffi` 25.1.0, OpenSSL 3.0.21. The storage device and filesystem were not recorded, so the throughput figures are single-host indicators rather than a comparison across media. Each timing script writes a JSON file with every run, and a CSV with the median and the min–max range; the paper reports medians and ranges, never a single favorable run. The scripts import the package from the repository under test and never reimplement its cryptography.

5.2 Test suite

Running `QT_QPA_PLATFORM=offscreen python -m pytest tests -o addopts= -q` collects 212 tests in eight files and reports 212 passed, 0 failed, 0 skipped, 0 errors in 8.5 s. The per-file counts are 57 for the vault, 33 for utilities, 29 for injection, 27 each for crypto and drive detection, 19 for the CLI, 13 for the GUI (offscreen), and 7 for hooks and progress. The suite is evidence that the implemented invariants hold at the measured commit; it is not independent evidence that the format is secure, and the paper does not treat it as such.

5.3 Format conformance

A format probe parses a freshly created container with an independent parser that uses only the public framing. It confirms that the magic, version, KDF identifier, flags and lengths are consistent; that the data offset is 4096-aligned, at least

as large as the header, and equal to the library's own view; that the record chain ends exactly at end-of-file; that all four records are complete; and that the 68,882-byte padding is zero. The probe also confirms that the index data key does not appear in plaintext and that the index is zlib compressed. For a 2,172,012-byte container holding 2,102,252 bytes of plaintext in four records, record framing overhead is 128 bytes, or 0.0061% of plaintext, and the 750-byte header is 0.0345% of the container. An empty file has zero records, and a file of exactly 2 MiB has two records, which confirms the boundary rule.

5.4 Header growth and relocation

With the index growing by appends, header size grows linearly at 33.42 bytes per entry with a 437-byte intercept: 1 entry gives 470 bytes, 4 give 588, 16 give 976, 64 give 2,549, and 256 give 8,998. The default reservation is 69,632 bytes, so the in-place growth capacity is roughly two thousand entries. The reserve experiment pushes past that point: at 1,966 entries the header reaches 69,633 bytes, the implementation relocates the data zone from offset 69,632 to 143,360, and the header becomes 69,633 bytes at the new offset. The 1,966 appends take 105.9 s with a deliberately cheap Argon2id (16 KiB, one pass) so that the cost of each append is dominated by I/O rather than the KDF. The reserve is therefore a real optimization for ordinary containers and a known cliff just below two thousand files; the paper does not claim that relocation is crash-safe, because it was not crash-tested.

5.5 Tamper matrix

The tamper experiment starts from a byte-identical copy of one pristine container and applies exactly one mutation. It then calls the three public entry points in order: `open_vault`, `Vault.verify`, and `Vault.extract`. Table 2 lists all 24 cases. Nineteen are detected by at least one entry point. Five are not.

The four benign misses have different causes. Padding is outside the AAD by construction, so a flipped pad byte is invisible; it is also ignored on read. Trailing bytes after the last record are never read, so appending garbage changes nothing. Reordering a re-sealed index requires the key, and the index is a map keyed by name, so order has no semantic effect. Extracting one healthy file while a sibling is corrupt is the intended behavior of selective extraction; the container-level `verify` would catch the corruption if it were called.

The rollback case is different. If an adversary replaces the container with an earlier snapshot, every authentication check passes, because the bytes are a genuine container. The format has no monotonic counter, no external witness, and no binding to a log; the implementation cannot distinguish an old valid container from the current one. This is a real limitation of the design, not an implementation bug. Section 6 discusses what would be needed to close it.

Two observations about the detection behavior are worth recording. First, mutations in the header and index fail at `open_vault` with `WrongPasswordError`, not with an integrity error; the exception class conflates a tampered header with a mistyped password. This is a UX and diagnostics defect, and it is reported rather than fixed. Second, data-zone mutations open successfully, because opening only decrypts the index; the failure appears at `verify` or `extract`. A user who only lists a container therefore does not learn that a payload chunk is corrupt. Listing is a metadata operation, and the format does not claim otherwise, but the asymmetry is easy to miss.

5.6 Metadata leakage

The metadata audit asks two questions: do distinctive strings survive in the raw container, and what can be inferred from framing without a key?

For the first, eleven tokens were placed in file names, directory names and content, including a marker inside a file body. The audit searches the raw bytes, a lower-cased copy, a UTF-16LE interpretation, and printable runs of at least four ASCII bytes. None of the eleven tokens appears in any of the four searches. The container has 105,228 printable runs totaling 483,069 bytes; exactly one run exceeds 16 characters, and the longest run is 86 bytes. The plaintext header contains only the magic and the KDF parameters: zero user-supplied strings, and the salt is the only high-entropy field. Names, directories and content are not recoverable by string search.

For the second, the data zone is walked with the public framing only: a 12-byte nonce, a 4-byte length, then that many bytes. A container with eight files of known sizes produces 13 records. Six records are shorter than 1 MiB. Two files have sizes that are exact multiples of 1 MiB and therefore leave no short record, so their records merge with the preceding file's records. The walker infers six groups from the short records, not eight files, and reports a descending size profile of 3,845,728, 2,101,152, 1,500,000, 1,048,577, 300,000 and 8,192 bytes. The true profile is 3,145,728, 2,097,152, 1,500,000, 1,048,577, 700,000, 300,000, 8,192 and 4,000 bytes. The totals agree exactly: 8,803,649 bytes in both cases. The leak is therefore of the record count and of sizes modulo the chunk size, not of names. An adversary learns roughly how many megabytes of data the container holds and how many chunk boundaries exist, but not which file is which, nor the directory tree.

Header growth is also observable without a key: 33.4 bytes per entry means an adversary can estimate the number of entries from the header size, to within the compression ratio of the index. The header probe reports 0 user strings but a header of 1,239 bytes for a container with a 1,116-byte compressed index; the KDF parameters alone account for the visible JSON. The format's own specification says the file names and tree are ciphertext, which is true; it does not say that the number of entries is hidden, and it is not.

Table 2: Single-mutation tamper matrix. *open* is `open_vault`; *verify* is the whole-container file digest check; *extract* is a full extraction. *Detected* means at least one entry point rejected the mutation. Four cases marked *no* are benign or require the key; the rollback case is the one security-relevant miss.

Mutation	open	verify	extract	Det.	Note
flip one bit of flags	error	n/a	n/a	yes	GCM AAD failure
change Argon2id $t = 3 \rightarrow 2$	error	n/a	n/a	yes	GCM AAD failure
flip one bit of data_offset	error	n/a	n/a	yes	GCM AAD failure
flip last byte of index ciphertext	error	n/a	n/a	yes	GCM tag failure
swap two 32-byte index blocks	error	n/a	n/a	yes	GCM tag failure
flip a padding byte	ok	ok	ok	no	padding not authenticated
flip one ciphertext byte of chunk 1	ok	failed	error	yes	chunk GCM
flip one nonce byte of chunk 1	ok	failed	error	yes	chunk GCM
set a record length to 0	ok	failed	error	yes	chunk GCM
inflate a record length by one	ok	failed	error	yes	chunk GCM
swap chunks 0 and 2 of one file	ok	failed	error	yes	sequence AAD
swap whole record runs of two files	ok	failed	error	yes	chunk-id AAD
paste a chunk from another container	ok	failed	error	yes	chunk-id AAD
overwrite a chunk with a copy of another	ok	failed	error	yes	chunk-id AAD
delete a chunk and shift the rest	ok	failed	error	yes	offset/digest
truncate the container by one byte	ok	failed	error	yes	framing / AES-GCM
truncate the final 0.5 MiB record	ok	failed	error	yes	record missing
remove every record of the last file	ok	failed	error	yes	SHA-256/size
append 9.6 KiB after the last record	ok	ok	ok	no	trailing bytes ignored
reverse entry order in a re-sealed index	ok	ok	ok	no	key required; order irrelevant
swap offsets of equal-size entries	ok	failed	error	yes	digest mismatch
swap SHA-256 fields in a re-sealed index	ok	failed	error	yes	digest mismatch
replace with a pre-append snapshot	ok	ok	ok	no	no freshness counter
extract only alpha while delta is corrupt	ok	n/a	ok	no	selective extraction by design

Table 3: Password rotation cost. *change* and *open* are medians of five runs. *changed* counts bytes that differ between the old and new header; *rewritten* is the reserved header region written by the commit. The data zone is unchanged in every case.

MiB	change (ms)	open (ms)	changed	rewritten	data same
8	105.4	110.2	388	69,632	yes
64	106.6	102.6	392	69,632	yes
256	106.7	108.5	397	69,632	yes
1,024	108.0	106.5	399	69,632	yes

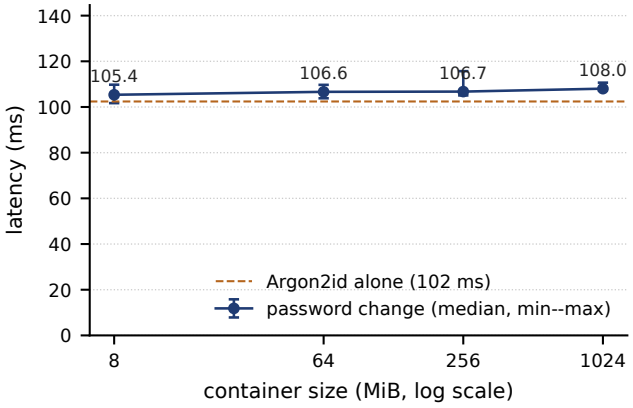


Figure 1: Password-change latency against container size, with the Argon2id cost alone (102.4 ms) as a reference. The flat line is the point: the payload is not re-encrypted.

5.7 Password rotation

Table 3 reports password changes for containers of 8, 64, 256 and 1,024 MiB. The median change latency is 105.4, 106.6, 106.7 and 108.0 ms, essentially flat; the Argon2id derivation alone is 102.4 ms, so the operation is dominated by the KDF and the payload is not touched. The implementation rewrites 69,632 bytes—the header region—of which only 388 to 399 bytes actually differ: the new salt, the new index nonce and ciphertext, and the checksum-like fields. The data zone is byte-identical, the container size is unchanged, and extraction after rotation returns the original bytes. The fraction of the container changed falls from 4.6×10^{-5} at 8 MiB to 3.7×10^{-7} at 1 GiB, which is the quantitative form of the constant-cost claim.

This property has a security caveat. Re-keying the envelope does not change the data key. If the data key was ever exposed—for example by a memory dump of an unlocked process or by a compromised host—changing the password does not revoke that exposure; an adversary who kept a copy of the old container and the data key can still decrypt the old payload, and can decrypt the new payload unless the data key is rotated and the payload re-encrypted. ULocker's rotation protects against password compromise, not data-key compromise. We did not measure a full re-encryption mode because the implementation does not expose one under that name.

5.8 Throughput and memory

Table 4 and Figure 2 report end-to-end create and extract throughput on 64 MiB and 256 MiB payloads, and the per-

Table 4: Throughput in MiB/s of plaintext, median of five runs (min–max in parentheses). *create* and *extract* are end-to-end and include `fsync` and, for extract, SHA-256 verification. *AEAD* and *SHA-256* are microbenchmarks with no file I/O.

Operation	64 MiB	256 MiB
create	130.5 (81.8–135.2)	210.8 (200.6–212.5)
extract	222.4 (209.6–271.3)	260.7 (248.1–268.3)
AEAD encrypt	2,106.8 (1,787.0–2,577.9)	
AEAD decrypt	1,871.3 (1,630.4–2,079.3)	
SHA-256	455.8 (454.0–468.3)	

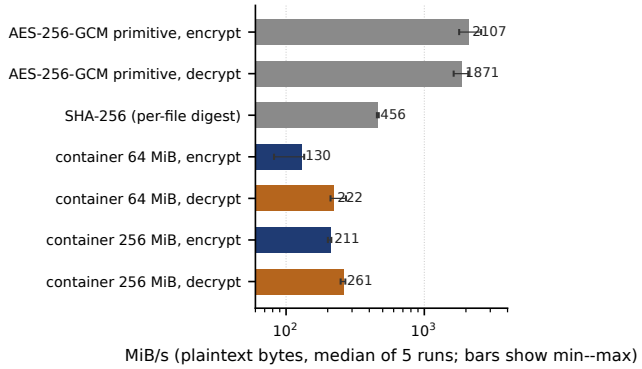


Figure 2: Throughput by layer. The gap between the AEAD microbenchmark and the end-to-end container rates is the cost of file I/O, `fsync`, chunk framing and SHA-256, not of AES.

layer rates for AES-256-GCM and SHA-256. Create is 130.5 MiB/s at 64 MiB and 210.8 MiB/s at 256 MiB; extract is 222.4 and 260.7 MiB/s. The smaller container pays a fixed cost: the KDF and the header reservation are amortized over less data. The AEAD layer alone runs at 2,106.8 MiB/s for encryption and 1,871.3 MiB/s for decryption; SHA-256 runs at 455.8 MiB/s. The end-to-end numbers are therefore dominated by file I/O, `fsync`, and the per-chunk Python call overhead, not by AES.

The memory experiment creates and extracts 8 MiB and 256 MiB containers while sampling the process's peak working set and tracing Python allocations. The traced peak is 6.18 MiB for create and 7.23 MiB for extract, and it is essentially identical at both sizes; this is the direct evidence for the streaming claim (G3). The working-set peak is about 106 MiB for create and 103 MiB for extract at both sizes, against a 21 MiB baseline. The difference is the 64 MiB Argon2id allocation plus the runtime; it does not grow with the container. A 1 GiB container would therefore not require 1 GiB of RAM.

Two caveats apply. The benchmark host has 16 GB of RAM and the page cache can absorb a 256 MiB file, so the throughput figures partly measure memory bandwidth; a cold-cache or removable-media run would be lower. And the memory figures are process-level on Windows, where working set includes pages shared with the runtime; the traced allocation is the cleaner signal.

5.9 Drive binding

A container can be bound to a volume serial number. In the binding experiment, a container bound to serial `DEADBEEF` refuses to open on a volume with serial `362ED8E0`: the error names both serials and the mount point and suggests `--ignore-drive-binding`. The only trace of the binding in the container is the `DRIVE_BOUND` flag in the header; neither the serial nor the volume label appears in the raw bytes. With the override, opening succeeds. This is the intended semantics, but it is important not to describe it as a cryptographic control: an adversary who can run the tool can pass the override, and a volume serial number can be cloned or spoofed. Binding raises the cost of casual misuse (mounting the container on the wrong stick) and does not stop a determined attacker who has the password. It is a policy hint, not a key derivation input.

6 Limitations

No rollback protection. A replaced container from an earlier snapshot verifies and extracts normally. Closing this requires state outside the container (a monotonic counter in trusted storage, a transparency log, or a remote anchor), and the design has none. The paper reports the miss rather than implying that authentication equals freshness.

Metadata is partially visible. The number of records and a size profile are inferable from the public framing, as measured in Section 5.6. Padding sizes, per-record lengths and the total container size are also visible. Hiding them would require fixed-size records, padding every file to a multiple of the chunk size, or an ORAM-like layout; none is implemented.

The drive binding is not cryptographic. It is a policy check implemented with a plaintext flag and an encrypted index field. The override exists by design. A spoofed volume serial defeats it, and an attacker with the password is not slowed down.

Default KDF parameters are modest. The default Argon2id is 64 MiB and three passes, which is below the RFC 9106 recommendation of 2 GiB for a memory-constrained environment IETF (2021). The parameters are stored in the header and can be raised at creation; the paper measures the default, and a user who chooses a 1 GiB memory cost will see a proportionally larger unlock latency and peak working set. The measured 102 ms is a default-configuration number, not a claim about the best achievable resistance. The parameter choice guards against offline guessing of a weak password; it is not a defense against defective randomness in the key material itself, the class of failure found in deployed devices Heninger et al. (2012).

Plaintext lives in memory. The password, the derived keys and the extracted content are ordinary Python objects. There is

no zeroization, no locked page, and no defense against a cold-boot or process-memory attacker Halderman et al. (2008). The threat model excludes this adversary, but the exclusion is a real boundary.

Secure deletion is best-effort. The `-shred-source` option overwrites a source file before unlinking it. On flash media with wear leveling, an overwrite is not guaranteed to reach the original physical blocks; the paper does not claim that shredded plaintext is unrecoverable.

Single-host performance. All timing is from one Windows machine with an unrecorded storage device. No comparison with LUKS, CFS, or a hardware encrypted drive was run, and no cold-cache measurement was taken. The numbers establish the order of magnitude and the constancy of rotation cost, not a portable performance ranking.

No independent audit. The implementation uses standard primitives and the composition is documented, but there is no formal verification and no third-party cryptographic review. The test suite and the tamper matrix are evidence of behavior, not a security proof. The AES primitives come from the platform crypto library, so the implementation inherits whatever side-channel resistance that library provides, and in particular does not address cache-timing attacks on AES Bernstein (2005).

Not evaluated. The GUI is exercised only by an offscreen smoke test. Crash consistency during header relocation was not tested. Concurrent access by two processes was not tested. The scrypt KDF path is implemented but not measured in this paper.

7 Related work

Block-level encryption (dm-crypt/LUKS Fruhwirth (2005)) protects a device but leaves the filesystem's metadata to the filesystem; an attacker who reads the ciphertext sees structure at the block layer, and the driver must be available on the reading machine. NIST SP 800-111 surveys the storage-encryption design space and the same trade-offs NIST (2007). CFS Blaze (1993) encrypts per file in a userspace filesystem and hides content but is tied to a mounted filesystem rather than a portable single file. ULocker sits between the two: a single portable file, a userspace reader, and an encrypted index that hides names and the tree.

The cryptographic building blocks are standard. AES-GCM is specified in FIPS 197 and in the mode recommendation SP 800-38D NIST (2001); Dworkin (2007), which also makes uniqueness of the initialization vector a requirement; the AEAD interface is RFC 5116 IETF (2008) and the associated-data construction is due to Rogaway Rogaway (2002). The per-chunk file-and-sequence associated data is the standard defense against cut-and-paste, and the fresh per-chunk nonce keeps the GCM input unique. Argon2id is the memory-hard KDF standardized in RFC 9106 IETF (2021); Biryukov et al. (2016); it follows the password-based KDF guidance of NIST SP 800-132 NIST (2010) and the bcrypt Provos and Mazières (1999) and scrypt Percival (2009); Percival and Josefsson (2016) line of memory-hard derivation functions. HKDF is used exactly as specified in RFC 5869 IETF (2010). The envelope construction is unusual in a file container only in that it re-wraps a small index rather than a per-file key, which is what makes rotation cheap.

8 Conclusion

ULocker is a container format for removable media that encrypts names, directory structure and content, streams in 1 MiB chunks, and re-keys in constant time with respect to payload size. Measured on one host, it passes 212 tests, detects 19 of 24 single mutations, hides all eleven planted metadata tokens, rotates a password in 105–108 ms for containers from 8 MiB to 1 GiB while rewriting 69,632 bytes, and sustains 130–261 MiB/s with 6.2–7.2 MiB of traced allocation independent of container size. It does not hide the number of records or a size profile, it has no rollback protection, and its drive binding is a policy check rather than a cryptographic one. Those limits are part of the result: a container format is defined as much by what its framing reveals as by what its ciphertext hides.

References

D. J. Bernstein. Cache-timing attacks on AES. IACR ePrint 2005/028, 2005.

A. Biryukov, D. Khovratovich, and T. Pornin. Argon2: A memory-hard function for password hashing and other applications. In *25th USENIX Security Symposium*, 2016.

M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, 1993.

M. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. NIST Special Publication 800-38D, National Institute of Standards and Technology, 2007.

C. Fruhwirth. New methods in hard disk encryption. Technical report, Vienna University of Technology, and the LUKS on-disk format specification, 2005.

J. A. Halderman, S. D. Schoen, N. Heninger, et al. Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium*, 2008.

N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium*, 2012.

IETF. An authenticated encryption mode interface for the transport layer security (TLS) extensible authenticated encryption with associated data (AEAD) construct. RFC 5116, 2008.

IETF. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, 2010.

IETF. The Argon2 cryptographic password hashing function. RFC 9106, 2021.

A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

NIST. Advanced encryption standard (AES). FIPS PUB 197, National Institute of Standards and Technology, 2001.

NIST. Guide to storage encryption technologies for end user devices. NIST Special Publication 800-111, National Institute of Standards and Technology, 2007.

NIST. Recommendation for password-based key derivation, part 1: Storage applications. NIST Special Publication 800-132, National Institute of Standards and Technology, 2010.

C. Percival. Stronger key derivation via sequential memory-hard functions. In *Proceedings of BSDCan 2009*, 2009.

C. Percival and S. Josefsson. The scrypt password-based key derivation function. RFC 7914, 2016.

F. A. P. Petitcolas. Watermarking steganography techniques. In *Information Hiding, First International Workshop, Revised Papers, Lecture Notes in Computer Science*, volume 1768, 2000.

N. Provos and D. Mazières. A future-adaptable password scheme. In *8th USENIX Security Symposium*, 1999.

P. Rogaway. Authenticated-encryption with associated-data. In *ACM Conference on Computer and Communications Security*, 2002.