MCES: Appendix and Test Results

Luke Miller

August 28, 2025

Abstract

Miller's Cantor Immune Encryption Scheme (MCES) is presented here with a full implementation and verification appendix. MCES's keystream was originally designed to beat a theorem regarding finding structure in highly entropic sources. After its initial NIST pass in the python test version, this version was designed to additionally resist traditional linear and differential attacks while additionally addressing modern threats such as machine-learning-based cryptanalysis. The cipher architecture combines entropy-driven walker functions, UTF-8 password expansion, and BLAKE3 post-mixing to produce a keystream that is both statistically uniform and highly resistant to structural bias. Argon2id is employed as the key derivation function, ensuring strong memory hardness and resistance to side-channel key recovery.

This appendix is intended as a practical companion to the source code. Each Rust program is decomposed into numbered sections and subsections, mapping directly to its functions, helpers, and data structures. By treating the codebase as a structured technical appendix, readers can move seamlessly between conceptual design and concrete implementation. Every function is annotated with its role in the larger cryptosystem, from primitive operations (`secure_zero`, `blake3_hash32`) through configuration builders, epoch reseeding, streaming generators, and high-level encryption wrappers. GUI tools, password managers, and benchmarking harnesses are likewise documented with the same granularity.

To validate security claims, MCES was subjected to a battery of empirical randomness tests, including the NIST Statistical Test Suite (STS), Dieharder, and PractRand. Across all experiments, MCES produced no anomalies beyond expected statistical variance. In NIST STS, all tests achieved pass rates above the recommended minimum ($\geq$96/100 sequences, $\geq$55/58 for excursion tests). Dieharder's full core battery was executed with 100–1,000,000 samples per test; all tests were assessed as PASSED, with p-values uniformly distributed across the acceptable range. PractRand analysis was extended to 1 terabyte of keystream output, and no anomalies were reported.

Performance measurements confirm that MCES is practical as well as robust. On a 12-thread host, throughput reached approximately 1.8 GB/s for keystream generation and 1.5 GB/s for full encryption, with negligible overhead for decryption. This places MCES in a performance class competitive with established stream ciphers while offering novel resistance properties grounded in entropy-driven state evolution.

Together, these results demonstrate that MCES is not only a theoretical construct but a deployable system with rigorously documented internals, verified randomness properties, and practical speed. The following appendix sections provide the complete breakdown of every program, enabling reproducibility, auditability, and extension by researchers and practitioners.

Contents

Section 10 — Speed and Randomness Testing **32**

Section 11 — Feasibility Analysis **34**

Section 12 — CLI Usage and Reproducibility **35**

Section 0 — Library and Headers — lib.rs / headers.rs

§0.1.0 Overview

Defines the public-facing entrypoints for MCES and its header structures.

§0.2.0 Constants

Core constants shared across MCES: header sizes, chunk sizes, versioning.

§0.2.1 VAULT_VERSION

Single-byte version identifier for vault files.

§0.2.2 MCES_CHUNK

Default streaming chunk size (4 MiB).

§0.2.3 MCES_HEADER_BYTES

Fixed size of vault header (61 bytes).

§0.2.4 MCES_TAG_BYTES

Size of BLAKE3 keyed MAC tag (32 bytes).

§0.3.0 VaultHeader Struct

Defines and validates the serialized vault header, containing salt, timestamp, nonce, and KDF parameters.

§0.3.1 encode()

Serialize a VaultHeader into its 61-byte binary format.

§0.3.2 decode()

Parse and validate a 61-byte header back into structured form.

§0.4.0 lib.rs Re-exports

Exposes the ‘headers‘ and ‘mces‘ modules as the public API of the crate.

§0.4.1 pub mod headers

Declares the headers module.

§0.4.2 pub mod mces

Declares the MCES core module.

§0.4.3 pub use headers::*

Re-exports header definitions for external use.

§0.4.4 pub use mces::*

Re-exports the MCES core library for external use.

Section 1 — MCES Core - mces.rs

§1.1.0 Overview

Rust mirror of the C MCES core library: keystream generation, masking, encryption/decryption, and streaming.

§1.2.0 Imports & Crate Uses

External crates and std imports needed for hashing, key derivation, zeroization, and threading.

§1.3.0 Primitives & Helpers

Low-level helper functions and primitives used throughout MCES.

§1.3.1 secure_zero

Securely zeroizes sensitive buffers.

§1.3.2 Small Helpers

Placeholder grouping for tiny utility functions.

§1.3.21 be64

Converts a 64-bit integer to big-endian bytes.

§1.3.22 blake3_hash32

Produces a fixed 32-byte digest from BLAKE3 XOF.

§1.3.23 sha256

Legacy helper: SHA-256 digest of input.

§1.3.24 hmac_sha256

Legacy helper: HMAC-SHA256 over input.

§1.3.25 hkdf_sha256

Legacy helper: HKDF-SHA256 key derivation.

§1.4.0 BLAKE3 KDF & MAC

Key derivation and message authentication using BLAKE3.

§1.4.1 kdf_blake3_split

Derives separate stream and MAC keys from a secret.

§1.4.2 mces_mac_blake3

Computes keyed BLAKE3 tag over message.

§1.5.0 Configuration Objects

Structures and utilities for password → config transformation.

§1.5.1 MCESConfig struct

Holds hash table count, digest vectors, and base key.

§1.5.2 utf8_codepoints_indices

Returns byte indices of UTF-8 codepoints in a string.

§1.6.0 Config Generation

Generate MCESConfig from a password and timestamp.

§1.6.1 generate_config_with_timestamp

Deterministically derives config from password + timestamp.

§1.6.2 generate_config

Wrapper using current system time.

§1.7.0 Epoch Seed & Walker Advance

Functions to initialize and advance walker indices.

§1.7.1 mces_epoch_seed

Seeds walker index and drift digest for an epoch.

§1.7.2 mces_advance_next_index

Computes next walker index based on drift + flags.

§1.8.0 Core Keystream

Generates keystream bytes from config with epoch rollover.

§1.8.1 generate_keystream

Fills output buffer with keystream bytes.

§1.9.0 Postmix Mask

Optional final masking layer using BLAKE3 XOF seek.

§1.9.1 apply_final_mask

Applies postmix-derived mask over keystream.

§1.10.0 Encrypt/Decrypt Wrappers

High-level encrypt/decrypt APIs using keystream + mask.

§1.10.1 encrypt_mces

Encrypts plaintext → ciphertext via XOR with keystream.

§1.10.2 decrypt_mces

Symmetric XOR decryption (same as encryption).

§1.11.0 Streaming Keystream

Thread-local streaming keystream with state caching.

§1.11.1 StreamTLS struct

Holds thread-local state for stream generation.

§1.11.2 STREAM_TLS thread-local

Global thread-local keystream cache.

§1.11.3 generate_stream

Streaming keystream generator with offset/seek support.

Section 2 — MCES Encrypt Function — mces_encrypt.rs

§2.1.0 Overview

Command-line encryption tool: reads a file, encrypts to ‘.vault‘ with password + Argon2id KDF, outputs vault and password.

§2.2.0 Imports & TLS State

External crates and thread-local buffers for keystream reuse.

§2.2.1 TLS_KS thread-local buffer

Scratch keystream buffer allocated per-thread to avoid repeated allocations.

§2.3.0 Small Helpers

Utility functions for byte encoding, alignment, random passwords, and XOR.

§2.3.1 u64_be

Converts 64-bit integer to big-endian bytes.

§2.3.2 round_up_32

Rounds an integer up to the nearest multiple of 32.

§2.3.3 random_unicode_password

Generates a random printable Unicode password with valid codepoints.

§2.3.4 xor_bytes_into

Applies XOR of two buffers into destination buffer.

§2.4.0 main: CLI Encryption Flow

Top-level program flow: parse CLI, build config, stream keystream, encrypt, MAC, and write vault.

§2.4.1 parse args

Collects filename and optional '–password' argument.

§2.4.2 open input & size

Opens input file, checks its size.

§2.4.3 password selection + TTY

Chooses custom or random Unicode password, prints conditionally to stdout.

§2.4.4 timestamp + nonce12

Creates timestamp (ns) and 12-byte random nonce.

§2.4.5 salt32

Derives BLAKE3 salt from timestamp and nonce.

§2.4.6 Argon2id params → OKM

Derives output keying material (stream + MAC keys) using Argon2id.

§2.4.7 split OKM → k_stream,k_mac

Splits Argon2id output into stream key and 32-byte MAC key.

§2.4.8 config from password

Builds MCES configuration from raw password + timestamp.

§2.4.9 postmix buffer

Constructs postmix input: label + k_stream + nonce + timestamp.

§2.4.10 header (61B)

Builds VaultHeader structure (salt, timestamp, nonce, KDF params).

§2.4.11 output file & tag reserve

Creates '.vault' file, writes header and placeholder MAC.

§2.4.12 init MAC domain

Initializes BLAKE3 MAC with domain, header, and file length.

§2.4.13 streaming + parallel XOR

Uses parallel Rayon slices to generate keystream and XOR plaintext → ciphertext.

§2.4.14 finalize tag

Computes final BLAKE3 tag, patches into vault header.

§2.4.15 durability: flush/fsync/delete

Ensures vault durability by flushing, syncing, and deleting original plaintext.

§2.4.16 hygiene

Zeroizes sensitive buffers (Argon2id OKM).

Section 3 — MCES Decrypt Function — `mces_decrypt.rs`

§3.1.0 Overview

Command-line decryption tool: reads a '.vault' file, validates header and MAC, rebuilds config, generates keystream, and outputs the decrypted plaintext.

§3.2.0 Imports & TLS State

External crates and thread-local state for streaming decryption.

§3.2.1 anyhow

Error handling and context for I/O operations.

§3.2.2 subtle::ConstantTimeEq

Provides constant-time equality checks for MAC verification.

§3.2.3 rayon::prelude

Parallel chunk processing for ciphertext decryption.

§3.2.4 zeroize::Zeroize

Ensures sensitive buffers are securely wiped.

§3.2.5 argon2

Argon2id KDF implementation for key material derivation.

§3.2.6 TLS_KS thread-local buffer

Scratch keystream buffer allocated per-thread for XOR operations.

§3.3.0 Small Helpers

Utility helpers specific to decryption.

§3.3.1 xor_bytes_into

Applies XOR of two buffers into destination buffer in-place.

§3.4.0 main: CLI Decryption Flow

Top-level program flow: parse CLI, open vault, validate header+MAC, reconstruct config+postmix, stream keystream, and output plaintext.

§3.4.1 parse args & open input

Collects vault filename from CLI arguments and opens file handle.

§3.4.2 read header (61B)

Reads VaultHeader from the first 61 bytes, validates structure.

§3.4.3 read tag (32B)

Reads stored BLAKE3 keyed MAC tag from file.

§3.4.4 file size checks

Ensures ciphertext length is valid relative to header+tag size.

§3.4.5 password prompt

Reads decryption password from TTY (rpassword) or stdin.

§3.4.6 Argon2id param checks

Validates header's KDF parameters (t_cost, m_cost, lanes).

§3.4.7 Argon2id derivation

Derives output keying material (stream key + 32B MAC key) using Argon2id with header salt.

§3.4.8 split OKM → k_stream,k_mac

Splits Argon2id output into keystream and MAC keys.

§3.4.9 init MAC domain

Initializes BLAKE3 keyed MAC with domain, header, and ciphertext length.

§3.4.10 stream MAC verification

Seeks to ciphertext, streams it chunk-by-chunk into BLAKE3 to verify tag against stored value.

§3.4.11 tag compare (ct_eq)

Performs constant-time equality check of computed vs stored MAC tag.

§3.4.12 rewind to ciphertext

Resets file pointer to ciphertext start for decryption pass.

§3.4.13 rebuild config

Calls generate_config_with_timestamp with password+timestamp from header.

§3.4.14 construct postmix buffer

Builds postmix = label + k_stream + nonce + timestamp.

§3.4.15 prepare output file

Strips '.vault' suffix from input filename and creates plaintext output file.

§3.4.16 streaming + parallel XOR

Uses Rayon to parallelize keystream generation and XOR ciphertext → plaintext.

§3.4.17 durability: flush/fsync/delete

Flushes and syncs output file; deletes source '.vault' file if suffix present.

§3.4.18 hygiene

Zeroizes Argon2id OKM and other sensitive buffers.

Section 4 — Speed Tester — mces_bench_stream.rs

§4.1.0 Imports

Brings in standard libraries, Rayon for parallelism, Zeroize for hygiene, and MCES core functions.

§4.2.0 Small Helpers

Lightweight utility functions used for encoding, alignment, and password generation.

§4.2.1 u64_be

Converts a 64-bit integer to big-endian bytes.

§4.2.2 round_up_32

Rounds a value up to the next multiple of 32.

§4.2.3 random_unicode_password

Generates a random Unicode password within valid codepoint ranges.

§4.3.0 main: Benchmark

Benchmarks streaming keystream performance, parallel encryption/decryption, and validates roundtrip correctness.

§4.3.1 data size from env

Reads MCES_MB environment variable, defaults to 100MB of data.

§4.3.2 random password

Generates a random Unicode password for config.

§4.3.3 timestamp + nonce

Captures system timestamp and generates a 12-byte nonce.

§4.3.4 salt32

Builds a BLAKE3 salt from timestamp and nonce.

§4.3.5 Argon2id → OKM

Derives output key material (stream + MAC keys) with Argon2id.

§4.3.6 config + postmix

Creates MCES config and postmix buffer from derived keys.

§4.3.7 buffers + plaintext

Allocates large buffers, fills plaintext from /dev/urandom.

§4.3.8 hardware threads + sweep

Detects hardware threads and sets benchmark thread sweep values.

§4.3.9 KS/ENC/DEC benchmark

Runs benchmarks for keystream generation, encryption, and decryption.

§4.3.9.1 KS only Measures raw keystream throughput (MB/s).

§4.3.9.2 ENC Measures parallel encryption throughput (MB/s).

§4.3.9.3 DEC Measures parallel decryption throughput (MB/s).

§4.3.9.4 results Prints results including throughput, roundtrip time, and correctness.

§4.3.10 hygiene

Zeroizes plaintext, ciphertext, buffers, and sensitive material after test.

Section 5 — Verdult 7 Test Harness — `mces_test_harness.rs`

§5.1.0 Overview

Full-system Verdult-7 harness for MCES: reproduces C semantics, tests AEAD, distinguishing, weak keys, avalanche sensitivity, and header invariants.

§5.2.0 Imports

Brings in std I/O, hashing, error handling, and MCES core modules.

§5.3.0 Params & CLI

Handles command-line configuration for test runs.

§5.3.1 Params struct

Holds test parameters: number of keys, IVs, bytes per stream, seed, and log path.

§5.3.2 parse_args

Parses CLI flags into Params (keys, ivs, bytes, seed, log).

§5.4.0 RNG

Implements deterministic splitmix64 RNG for reproducibility.

§5.4.1 splitmix64

Core 64-bit random generator step.

§5.4.2 rng_seed

Seeds global RNG state.

§5.4.3 rng_u64

Returns next 64-bit random value.

§5.4.4 rng_u32

Returns next 32-bit random value.

§5.5.0 Small Utils

Helper functions for statistics, encodings, and comparisons.

§5.5.1 round_up_32

Rounds to the nearest multiple of 32.

§5.5.2 hamming_bits

Counts differing bits between two buffers.

§5.5.3 serial_corr

Computes serial correlation of adjacent bytes.

§5.5.4 chi2_bytes

Performs chi-squared distribution test on byte frequencies.

§5.5.5 ctcmp32

Constant-time 32-byte comparison (returns 0 if equal).

§5.5.6 be32

Encode u32 as big-endian bytes.

§5.5.7 be64

Encode u64 as big-endian bytes.

§5.5.8 le64

Encode u64 as little-endian bytes.

§5.6.0 Deterministic ts/nonce

Generates timestamp + nonce deterministically (for reproducible harness tests).

§5.6.1 derive_ts_nonce

Derives 12B nonce + 64-bit timestamp from seed, key index, IV index.

§5.7.0 Passwords & Configs

Password creation, configs, salts, and header helpers.

§5.7.1 KeySlot struct

Bundles password with its MCESConfig.

§5.7.2 make_password

Creates ASCII passwords (33–126) deterministically.

§5.7.3 setup_keys

Generates multiple KeySlots for testing.

§5.7.4 salt_from_ts_nonce

Derives salt32 from timestamp + nonce.

§5.7.5 derive_okm_from_pw

Argon2id: derive stream + MAC keys from password + salt.

§5.7.6 build_postmix

Constructs postmix (label || k_stream || nonce || ts).

§5.7.7 build_header61

Constructs 61-byte vault header from fields.

§5.8.0 Scheme Operations

Implements in-memory vault encryption and decryption.

§5.8.1 scheme_encrypt_vault

Encrypt plaintext → vault (header + tag + ciphertext).

§5.8.2 scheme_decrypt_vault

Decrypts vault, verifies MAC, returns plaintext.

§5.9.0 Tests

Implements Verdult-7 style cryptanalysis tests against MCES.

§5.9.1 test1_aead

Validates AEAD: ciphertext/header bit flips must be detected.

§5.9.2 test2_kpa_heads

Known-plaintext recovery of keystream heads.

§5.9.3 test3_seek_equivalence

Checks equivalence of whole-stream vs. piecewise keystream.

§5.9.4 test4_distinguishing

Chi-squared + serial correlation distinguishing tests.

§5.9.5 test5_bit_bias

Bit-position bias detection across IVs.

§5.9.6 test6_weak_keys

Scans for weak keys via IV head collisions.

§5.9.7 test7_key_sensitivity

Avalanche test: bit-flip in password → keystream divergence.

§5.9.8 test8_tag_forgery

Ensures random tag modifications are always detected.

§5.9.9 test9_header_invariants

Validates salt derivation matches header values.

§5.10.0 Main

Entry point: parses args, seeds RNG, sets up keys, runs all tests, and logs results.

§5.10.1 main

Full orchestrator: drives the Verdult-7 test suite and writes results to file.

Section 6 — Password Manager — sigilbook.rs

§6.1.0 Overview

Velvet Sigilbook: password manager for vaults with USB synchronization and Linux-only permission hardening.

§6.2.0 Imports & Uses

Rust imports: anyhow, clap, rand, serde, sha2, std::fs/io/path, subtle, unicode_normalization, whoami, zeroize, process::Command.

§6.3.0 Constants

Global constants for directory layout, filenames, magic/version bytes, and scrypt parameters.

§6.3.1 ALLOWED_USER

Hardcoded username permitted to run sigilbook (VelvetGuard).

§6.3.2 BASE_DIR

Path under `/var/lib/velvet/sigilbook` for local configuration and seed file.

§6.3.3 SEED_FILE

Path to quick/dirty master key seed (`.sigil.seed`).

§6.3.4 USB_KEY_FILENAME / USB_DETECT_LABEL

Filenames for the encrypted password database (`VELVET_SIGILBOOK.sigil`) and the USB marker file (`VELVETKEY.info`).

§6.3.5 MAGIC / VERSION

File magic and version for header validation.

§6.3.6 Scrypt parameters

`LOG_N=14`, `R=8`, `P=1`, `DKLen=32`.

§6.4.0 CLI

clap Parser and Subcommands for entrypoints.

§6.4.1 Writeusb

Write velvet key marker (VELVETKEY.info) to a mounted USB.

§6.4.2 Get

Retrieve stored password for a given vault file.

§6.4.3 Save

Save password for vault file; supports “-” to read password from stdin.

§6.5.0 Data Models

serde-backed database structures.

§6.5.1 Db struct

Root database, holds list of Entry objects.

§6.5.2 Entry struct

File hash, password, and associated path list.

§6.6.0 Guards & Dirs

Security checks and base directory setup.

§6.6.1 velvet_guard

Enforces UID ≥1000 and exact NFC-normalized username match.

§6.6.2 ensure_base

Creates BASE_DIR with 0700 permissions.

§6.7.0 Crypto Primitives

Core cryptographic helpers.

§6.7.1 scrypt_seed

Derives 32-byte seed from master key and salt.

§6.7.2 xor_keystream

XOR stream cipher: SHA-256(seed||counter_le).

§6.7.3 b3_tag

Computes BLAKE3 keyed MAC over domain, header, length, ciphertext.

§6.8.0 Header Encode/Decode

Vault header handling.

§6.8.1 header_encode

Builds header vector: MAGIC||VER||slen||salt.

§6.8.2 header_decode

Validates header, extracts salt, returns header slice.

§6.9.0 Master Key

Quick/dirty master key seed management.

§6.9.1 seed_path

Returns path to master seed file.

§6.9.2 load_master_key

Loads existing key or prompts user to create new Unicode seed; enforces 0600 permissions.

§6.10.0 DB I/O

Database serialization and MAC enforcement.

§6.10.1 save_db

Serializes DB as JSON, encrypts with xor_keystream, prepends header, appends tag.

§6.10.2 load_db

Reads DB, validates MAC, decrypts, and deserializes into struct.

§6.11.0 Helpers

File and media helper utilities.

§6.11.1 file_hash

Computes SHA-256 hash of file contents (or metadata fallback).

§6.11.2 media_mounts

Returns candidate USB mount points.

§6.11.3 find_marker_any

Locates any USB drive with VELVETKEY.info marker.

§6.11.4 find_by_uuid

Resolves a mount by UUID from marker metadata.

§6.11.5 usb_db_path

Returns path to USB database file if present.

§6.12.0 Merge & Sync

Database merging and holding→USB synchronization.

§6.12.1 merge_unique

Deduplicates entries while merging.

§6.12.2 sync_holding_to_usb

Loads both DBs, merges, saves back to USB, deletes holding file.

§6.12.3 pick_db_path

Chooses DB path from USB detection.

§6.13.0 Operations

High-level password storage and retrieval.

§6.13.1 save_password

Stores password into the encrypted database on the Velvet USB.

§6.13.2 get_password

Reads password for a given vault file from the Velvet USB database.

§6.14.0 USB Marker

Writes and manages USB marker files.

§6.14.1 write_usb_key

Interactively selects USB drive, queries UUID with ‘lsblk‘, writes VELVETKEY.info JSON marker with 0600 perms.

§6.15.0 Main

Program entrypoint: dispatches to subcommands.

§6.15.1 match CLI

Calls writeusb, get, or save depending on user input.

§6.15.2 hygiene

Zeroizes in-memory password buffers after save.

Section 7 — Gui Frontend — `mces_gui.rs`

§7.1.0 Overview

Desktop GUI for MCES built with eframe/egui. Supports encrypt/decrypt of files, USB sigilbook detection, and optional password saving to key device.

§7.2.0 Imports & Uses

External crates and std for GUI, concurrency, and crypto.

§7.2.1 eframe/egui

Provides GUI widgets and event loop.

§7.2.2 blake3, rayon, zeroize

Crypto hashing, parallel XOR, and secure buffer wipes.

§7.2.3 mces::headers / mces::mces

`VaultHeader` struct, config, and keystream generation.

§7.2.4 std::sync::mpsc

Channels for async events between worker threads and GUI.

§7.2.5 whoami, std::process

User detection and external process spawning for sigilbook integration.

§7.2.6 percent_encoding, dirs

Robust decoding of `file://` URIs from GTK bookmarks and location of config dirs.

§7.3.0 USB Detection & Sigilbook Save

Helper functions for key detection and sigilbook I/O.

§7.3.1 usb_key_detected

Scans `/media/$USER` and `/run/media/$USER` for `VELVETKEY.info`.

§7.3.2 sigilbook_save

Spawns the sigilbook client to save the generated password into the USB database (stdin "-" flow).

§7.3.3 sigilbook_get

Invokes the sigilbook client to retrieve a stored password for the selected vault (if present).

§7.3.4 sigilbook_get_async

Asynchronous retrieval of password from the sigilbook client; on success launches a background decrypt worker and reports completion via the event channel.

§7.3.5 list_mounted_drives

Parses `/proc/mounts` and scans `/media/$USER`, `/run/media/$USER` to assemble unique, user-relevant mount points for the file manager.

§7.3.6 load_gtk_bookmarks

Reads GTK bookmark files (`gtk-3.0/bookmarks`, `gtk-4.0/bookmarks`); decodes `file://` URIs; returns existing paths as quick-access bookmarks.

§7.4.0 Events & Password Utility

Definitions for async events and password generation.

§7.4.1 Event enum

Variants for `EncryptDone`, `DecryptDone`, and `Error` passed to the GUI.

§7.4.2 random_unicode_password

Generates random Unicode passwords in a configurable codepoint range.

§7.4.3 Helpers

`u64_be` (big-endian encoder) and `round_up_32` (ceil to multiple of 32).

§7.5.0 Core Encrypt/Decrypt

Encryption and decryption functions invoked by the GUI.

§7.5.1 encrypt_file

Opens input file; generates password; derives Argon2id keys; builds `VaultHeader`; streams parallel XOR with MCES; writes vault; validates & writes MAC; deletes plaintext on success.

§7.5.2 decrypt_file

Opens `.vault`; validates header and MAC; derives Argon2id keys; reconstructs config/postmix; streams parallel XOR; writes plaintext; deletes the vault on success.

§7.6.0 GUI State

Application state struct.

§7.6.1 App fields

Tracks selected file path, status text, last generated password, password prompt state/input, busy flag, USB detection flag, save-to-key toggle, and tx/rx channels; maintains polling cadence (`last_poll`, `poll_interval`); and holds file-manager state: `current_dir`, `filter`, `selected_path`, discovered `mounts`, and GTK `bookmarks`.

§7.6.2 Default impl

Initializes fields with defaults; performs initial USB detection; populates mounts and bookmarks.

§7.7.0 GUI Implementation (eframe::App)

Main update loop and UI wiring.

§7.7.1 File picker row

Displays current file or "No file selected".

§7.7.2 USB status row

Live USB key detection + "Save password to key" checkbox (disabled if no key).

§7.7.3 Encrypt button

Spawns worker thread to run `encrypt_file`; emits `EncryptDone` or `Error`.

§7.7.4 Decrypt button

If USB key present, attempts `sigilbook_get` first; otherwise opens password modal and then spawns `decrypt_file`.

§7.7.5 Password modal

Centered secure entry window for decryption.

§7.7.6 Event handling

Processes async events, updates status, and (optionally) triggers `sigilbook_save` after successful encrypt.

§7.7.7 Busy polling

Minimal filesystem polling to reflect job completion in the UI.

§7.7.8 File manager panel

Right-side panel with dual-pane layout: toolbar exposes root, volumes (mounts), and GTK bookmarks; persistent search filter applies substring match over paths; scrollable directory view supports ".." parent entry, single-click selection, and double-click quick decrypt when a USB key is present.

§7.8.0 Main

Entry point.

§7.8.1 NativeOptions

Configures window size.

§7.8.2 run_native

Launches eframe event loop with `App::default`.

Section 8 — Stream for Dieharder, Practrand or Big Crush — `mces_stream_dieharder.rs`

§8.1.0 Overview

MCES v1 Cantor-Immune stream cipher streamer: generates continuous pseudorandom output for dieharder/PractRand testing using worker pools and round-based jobs.

§8.2.0 Imports & Helpers

External crates and helper functions.

§8.2.1 anyhow, blake3, zeroize

Error handling, hashing, secure buffer zeroization.

§8.2.2 rand, num_cpus, argon2

RNG, hardware thread detection, key derivation.

§8.2.3 MCES core

generate_config_with_timestamp, generate_stream, MCESConfig.

§8.2.4 Helpers

u64_be (big-endian encode), round_up_32 (align length).

§8.3.0 Password Generation

§8.3.1 random_unicode_password

Creates random Unicode passwords (30–101 cps) with valid codepoints.

§8.4.0 Worker Pool

Thread pool design for per-round job execution.

§8.4.1 Job struct

Holds offset, length, and pointer into output buffer.

§8.4.2 Shared state

Tracks job queue, round id, completion count, stop flag.

§8.4.3 Pool::new

Initializes worker threads based on hardware and hints.

§8.4.4 Worker loop

Each thread waits on Condvar, processes job with generate_stream, marks completion.

§8.4.5 Pool::run_round

Splits output buffer into stripes across lanes, fills with keystream.

§8.4.6 Pool::drop

Signals stop flag and joins threads cleanly.

§8.5.0 Main

Top-level program flow.

§8.5.1 stdout

Locks stdout for unbuffered streaming.

§8.5.2 password/seed

Generates random Unicode password, timestamp, nonce, salt, Argon2id OKM.

§8.5.3 config/postmix

Builds MCESConfig and postmix (label + k_stream + nonce + ts).

§8.5.4 pool creation

Initializes Pool with threads and shared state.

§8.5.5 streaming loop

Runs repeated Pool::run_round calls to fill buffer and write to stdout.

§8.5.6 hygiene

Zeroizes buffers and sensitive materials after exit.

Section 9 — Side-Channel Probe Harness — `attacker.rs`

§9.1.0 Overview

Simulates a co-resident "attacker" thread running while MCES encrypts. The attacker never sees the password or key; it only measures timing jitter from a cache-thrashing probe loop while the victim encrypts a large buffer.

§9.2.0 Imports & Types

Standard concurrency primitives (`Arc`, `Barrier`), threading, timing utilities, and MCES core functions.

§9.3.0 Victim Role

Runs MCES encryption on a large buffer while attacker probes.

§9.3.1 sync start

Barrier to align start with attacker.

§9.3.2 encrypt

Invoke `encrypt_mces` on plaintext buffer.

§9.4.0 Attacker Role

Cache-heavy probe loop, sampling iteration runtimes.

§9.4.1 sync start

Barrier to align start with victim.

§9.4.2 probe buffer

4096-element array to thrash caches.

§9.4.3 bounded run

Cap by iterations and elapsed time.

§9.4.4 cache-stress pass

Update buffer to maximize contention.

§9.4.5 sample elapsed time

Collect nanosecond runtime.

§9.4.6 summarize distribution

Min, max, mean reported.

§9.5.0 Main

Coordinates victim and attacker threads.

§9.5.1 password

Fixed control password (can vary per run).

§9.5.2 timestamp

Current system time for config.

§9.5.3 build config

Generate MCES config from password+ts.

§9.5.4 buffers

Allocate 32 MiB plaintext and ciphertext.

§9.5.5 barrier

Initialize two-party barrier.

§9.5.6 spawn victim

Launch victim thread.

§9.5.7 spawn attacker

Launch attacker thread (≈1s run).

§9.5.8 join

Wait for threads to complete.

§9.6.0 Experiment Notes

Run multiple trials with different passwords (lengths, Unicode mix). Save attacker traces and compare distributions using KS-test, MWU, or t-test. Pin threads to cores to reduce OS jitter; disable Turbo Boost. Consider adding noise threads to simulate multi-tenant systems. If distributions diverge significantly → potential leakage.

Section 10 — Speed and Randomness Testing

§R.1.0 Speed Benchmarks (mces_bench_stream)

THREADS	KS (MB/s)	ENC (MB/s)	DEC (MB/s)	Roundtrip (ms)	OK
1	560.02	436.05	541.69	413.939	YES
2	918.23	860.91	1057.26	210.740	YES
4	1512.59	1364.19	1856.80	127.160	YES
6	1569.27	1434.49	1805.08	125.111	YES
8	1801.24	1447.51	1751.71	126.171	YES
12	1800.89	1539.40	1781.94	121.079	YES

Note. Host reported 12 HW threads; data size 100 MB.

§R.2.0 NIST STS Summary (100 sequences unless noted)

Test	P-VALUE	Proportion	Notes
Frequency (Monobit)	0.455937	100/100	PASS
BlockFrequency	0.122325	100/100	PASS
CumulativeSums (forward)	0.350485	100/100	PASS
CumulativeSums (reverse)	0.249284	100/100	PASS
Runs	0.637119	99/100	PASS ($\geq$ 96 min)
LongestRun	0.129620	98/100	PASS ($\geq$ 96 min)
Rank	0.759756	99/100	PASS
FFT	0.224821	99/100	PASS
Universal	0.494392	99/100	PASS
ApproximateEntropy	0.249284	99/100	PASS
Serial (set 1)	0.946308	99/100	PASS
Serial (set 2)	0.304126	100/100	PASS
LinearComplexity	0.213309	97/100	PASS
RandomExcursions (58 seqs, min 55)	0.040108–0.699313	58/58, 57/58	All PASS (meets per-test minima)
RandomExcursionsVariant (58 seqs, min 55)	0.004981–0.657933	57/58, 58/58	All PASS (meets minima)
NonOverlappingTemplate (many templates)	0.000954–0.997823	**96/100–100/100**	All templates PASS; min proportion $\geq$ 96

NIST guidance (from log): For 100-seq tests, minimum pass rate $\approx$ 96; for RandomExcursions(Variant) with 58 sequences, minimum pass rate $\approx$ 55.

§R.3.0 Dieharder (core set)

Test	tsamples	psamples	p-value	Assessment
diehard_birthdays	100	100	0.96295231	PASSED
diehard_operm5	1,000,000	100	0.14372555	PASSED
diehard_rank_32x32	40,000	100	0.50230183	PASSED
diehard_rank_6x8	100,000	100	0.03782506	PASSED
diehard_bitstream	2,097,152	100	0.14919477	PASSED
diehard_opso	2,097,152	100	0.00514229	PASSED
diehard_oqso	2,097,152	100	0.94258884	PASSED
diehard_dna	2,097,152	100	0.22532051	PASSED
diehard_count_1s_str	256,000	100	0.81082814	PASSED
diehard_count_1s_byt	256,000	100	0.23908340	PASSED
diehard_parking_lot	12,000	100	0.18103491	PASSED
diehard_2dsphere	8,000	100	0.49373616	PASSED
diehard_3dsphere	4,000	100	0.77749845	PASSED
diehard_squeeze	100,000	100	0.13589375	PASSED
diehard_sums	100	100	0.22601624	PASSED
diehard_runs (1)	100,000	100	0.02668232	PASSED
diehard_runs (2)	100,000	100	0.63517538	PASSED
diehard_craps (1)	200,000	100	0.98533286	PASSED
diehard_craps (2)	200,000	100	0.80795612	PASSED
marsaglia_tsang_gcd (1)	10,000,000	100	0.85529437	PASSED
marsaglia_tsang_gcd (2)	10,000,000	100	0.92654882	PASSED
sts_monobit	100,000	100	0.65028346	PASSED
sts_runs	100,000	100	0.99039185	PASSED
sts_serial (multiple k)	100,000	100	0.0172–0.9805	PASSED
rgb_bitdist (k=1..12)	100,000	100	0.0868–0.9432	PASSED
rgb_minimum_distance	10,000	1000	0.189–0.861	PASSED
rgb_permutations (k=2..5)	100,000	100	0.104–0.986	PASSED
rgb_lagged_sum (k=0..32)	1,000,000	100	0.028–0.982	PASSED
rgb_kstest_test	10,000	1000	0.16850886	PASSED
dab_bytedistrib	51,200,000	1	0.65049358	PASSED
dab_dct	50,000	1	0.43126753	PASSED
dab_filltree (x2)	15,000,000	1	0.967–0.973	PASSED
dab_filltree2 (x2)	5,000,000	1	0.711–0.744	PASSED
dab_monobit2	65,000,000	1	0.66599936	PASSED

§R.4.0 PractRand (RNG_stdin32, folding=standard, 32-bit)

Length	Time (s)	Result
256 MB	3.2	no anomalies (165 tests)
512 MB	6.7	no anomalies (178 tests)
1 GB	13.4	no anomalies (192 tests)
2 GB	25.8	no anomalies (204 tests)
4 GB	50.8	no anomalies (216 tests)
8 GB	98.8	no anomalies (229 tests)
16 GB	192	no anomalies (240 tests)
32 GB	375	no anomalies (251 tests)
64 GB	751	no anomalies (263 tests)
128 GB	1497	no anomalies (273 tests)
256 GB	3036	no anomalies (284 tests)
512 GB	6207	no anomalies (295 tests)
1 TB	12478	no anomalies (304 tests)

Section 11 — Feasibility Analysis

§11.1.0 Overview

To complement empirical statistical testing, MCES was subjected to a symbolic recovery analysis using the Z3 SMT solver. The goal was to determine whether an adversary, given limited known plaintext/ciphertext pairs, could feasibly recover the internal walker state, drift digest, or postmix seed values.

§11.2.0 Methodology

The experiment constructed a reduced MCES configuration with tunable parameters:

- **Bit width:** variable symbolic state sizes (64–256 bits).
- **Rows:** walker table size (e.g., 6 rows).
- **Steps:** number of consecutive keystream outputs revealed.

For each configuration, Z3 was tasked with solving the system of equations linking known plaintext, ciphertext, and symbolic state variables. A timeout was imposed for infeasible problem sizes.

§11.3.0 Results

The results are summarized visually in Figure **??**.

- **Green (Unique):** A unique internal state could be recovered.
- **Red (Multiple):** Multiple candidate states satisfied the equations; recovery was ambiguous.
- **Gray (Timeout):** The solver could not finish within the allotted time, indicating intractable complexity.

§11.4.0 Interpretation

The feasibility map demonstrates that:

- At small bit widths and extremely shallow step counts, partial state recovery may be possible.
- With realistic parameters ($\geq$100-bit width, multi-step output), the system consistently produces either multiple candidate states or solver timeouts, indicating infeasibility of exact recovery.
- The walker function, drift digest, and postmix mixing jointly contribute to an exponential explosion in state space, making symbolic inversion computationally impractical.

§11.5.0 Conclusion

This symbolic recovery analysis provides supporting evidence that MCES is resistant to algebraic plaintext-recovery attacks. Even under idealized adversarial conditions with full access to known plaintext/ciphertext pairs, state recovery becomes infeasible once the system operates at its intended parameter sizes.

Section 12 — CLI Usage and Reproducibility

§12.1.0 Build Instructions

All binaries live under `src/bin/`. Build everything with:

```
cargo build --release
```

Run a specific tool:

```
cargo run --release --bin <binary_name> -- <args>
```

§12.2.0 Program Index

mces_encrypt — Encrypts a file into `.vault`.

```
cargo run --release --bin mces_encrypt file.txt
cargo run --release --bin mces_encrypt file.txt --password "pw"
```

Produces `file.txt.vault` and prints password.

mces_decrypt — Decrypts a vault back to plaintext.

```
cargo run --release --bin mces_decrypt file.txt.vault
```

mces_bench_stream — Benchmarks throughput (KS/ENC/DEC).

```
MCES_MB=100 cargo run --release --bin mces_bench_stream
```

Prints keystream/encryption/decryption MB/s results.

mces_test_harness — Verdult-7 style cryptanalysis harness. Tests AEAD malleability, KPA heads, seek equivalence, distinguishing, bias, avalanche, etc.

```
cargo run --release --bin mces_test_harness \
  -keys 10 -ivs 64 -bytes 1048576 -seed 0xC0DEFACE12345678
```

Writes `mces_test_harness.log`.

mces_stream_dieharder — Continuous keystream generator. Pipe directly into external batteries (Dieharder, PractRand, BigCrush).

```
cargo run --release --bin mces_stream_dieharder | dieharder -a -g 200
cargo run --release --bin mces_stream_dieharder | ./RNG_test stdin32
```

attacker — Side-channel probe harness (victim vs attacker).

```
cargo run --release --bin attacker
```

Prints jitter distribution stats.

mces_gui — Desktop GUI (eframe/egui) for encrypt/decrypt. Includes sigilbook USB integration.

```
cargo run --release --bin mces_gui
```

§12.3.0 Notes

- **PractRand/Dieharder:** Use long keystreams ($\geq$1 TB) for maximum coverage.
- **NIST STS:** Requires pre-generated binary sequences (`mces_stream_dieharder` recommended).
- **Side-channel:** Pin threads with `taskset` or `numactl` for consistency.
- **GUI:** Launches a native desktop app for interactive use, password management, and sigilbook USB integration.