Lethesafe – Whitepaper

Version 1.0
Stand: 18.02.2026
Dieses Dokument wird ab Version 1.0 formal versioniert.
Frühere Fassungen wurden ohne Versionskennzeichnung veröffentlicht.

Abstract

Lethesafe ist eine Sammlung lokal ausführbarer Werkzeuge zur Umsetzung von Time-Lock-Puzzles (TLP), die zeitverzögerte Geheimnisfreigaben ohne zentrale Infrastruktur ermöglicht. Lethesafe implementiert ein nicht-RSA-basiertes, strikt sequenzielles Time-Lock-Verfahren ohne Suchraum. Lethesafe verschlüsselt keine Geheimnisse im klassischen Sinn, sondern verzögert deterministisch die Existenz eines für den Zugriff notwendigen Schlüssels. Optional kann die Verzögerung nutzerseitig zeitbasiert angegeben werden (z. B. Stunden oder Tage). Die hierfür notwendige Rundenzahl wird durch einen lokalen Kalibrierungslauf automatisch bestimmt; die kryptographischen Verfahren und Dateiformate bleiben unverändert. Das System kombiniert eine kryptographisch abgesicherte Hashkette mit passwortgeschützten Startwerten sowie Referenzimplementierungen mit Web-Interface und Kommandozeilenwerkzeugen. Alle kritischen Operationen laufen offline auf dem Gerät der Nutzer.

Dieses Whitepaper beschreibt das konzeptionelle und kryptographische Modell von Lethesafe. Es dient als Spezifikation dieses Modells und definiert dessen grundlegende Struktur, Annahmen und Sicherheitsgrenzen. Dieses Dokument ist von der Referenzimplementierung getrennt zu betrachten. Änderungen an der Implementierung führen nicht automatisch zu Änderungen dieses Dokuments, sofern das zugrunde liegende Modell unverändert bleibt.

1. Problemstellung & Zielbild

Zeitkapseln werden genutzt, um eine Zielzeichenkette K erst nach einer definierten Menge an CPU-Arbeit entstehen zu lassen. Diese Zielzeichenkette kann anschließend als Masterpasswort für externe Verschlüsselungssysteme dienen. In Szenarien wie Notfall-Backups, digitaler Nachlassverwaltung oder erpresserischen Zugriffssituationen muss der Geheimnisträger sicherstellen, dass weder Angreifer noch er selbst vor Ablauf der Zeitbarriere Zugriff auf die verschlüsselten Informationen erlangen kann. Lethesafe adressiert dafür folgende Ziele:

- **Autarke Nutzung:** Keine Abhängigkeit von Servern, Blockchains oder Drittparteien.
- **Nachvollziehbarkeit:** Vollständig einsehbarer Python-Code in Repository lethesafe-core und Entwicklung.
- **Benutzerfreundlichkeit:** Browser-Oberfläche und CLI-Tools unterstützen wahlweise runden- oder zeitbasierte Eingaben zur Definition der Verzögerung.
- **Flexibilität:** Mehrere Rundenzahlen in einem Lauf, Cloning bestehender Kapseln und konfigurierbare Passwortpolitik.

2. Systemübersicht

2.1 Architektur

Die Architektur von Lethesafe folgt einer strikt geschichteten Trennung von Verantwortlichkeiten. Ziel ist es, sicherheitskritische kryptographische Operationen vollständig von Benutzerinteraktion, Transportmechanismen und Ablaufsteuerung zu entkoppeln.
Zusätzlich wird die semantische Beschreibung einer Zeitkapsel strikt von ihrer textuellen Repräsentation getrennt. Die Normalisierung und Validierung des Puzzle-Formats erfolgt ausschließlich an der Systemgrenze; unterhalb dieser Grenze arbeiten alle Komponenten ausschließlich mit kanonischen, semantischen Parametern.
Alle Komponenten sind so gestaltet, dass sie lokal, deterministisch und ohne Abhängigkeit von externer Infrastruktur arbeiten.

Die Systemarchitektur gliedert sich in vier funktionale Ebenen:

2.1.1 Kryptographischer Core

Der kryptographische Core bildet das sicherheitsrelevante Fundament von Lethesafe. Er enthält ausschließlich deterministische Berechnungen zur Erzeugung, Klonung und Entschlüsselung von Zeitkapseln.

Eigenschaften des Core:

- Implementiert sämtliche kryptographischen Primitive (Hashketten, KDFs, MACs)
- Kennt weder Benutzeroberflächen, Dateiformate noch Transportprotokolle
- Arbeitet vollständig zustandslos und deterministisch
- Stellt eine klar definierte Funktionsschnittstelle für übergeordnete Ebenen bereit

Der Core ist bewusst minimal gehalten und kann unabhängig von Web- oder CLI-Komponenten auditiert, getestet und wiederverwendet werden.

2.1.2 Workflow- & Adapter-Schicht

Oberhalb des kryptographischen Cores befindet sich eine dedizierte Workflow- und Adapter-Schicht. Diese Ebene übersetzt anwendungsnahe Anforderungen in wohldefinierte, semantische Core-Aufrufe. Sie verarbeitet keine Dateiformate, keine JSON-Strukturen und keine feldbasierten Repräsentationen von Zeitkapseln. Sämtliche Eingaben liegen dieser Schicht ausschließlich in normalisierter, kanonischer Form vor.

Aufgaben dieser Schicht:

- Orchestrierung der Abläufe für Erstellen, Klonen und Entschlüsseln
- Kapselung von Laufzuständen wie Fortschritt und Abbruchsignalen
- Einheitliche Nutzung des Cores durch unterschiedliche Interfaces
- Trennung von Ablaufsteuerung und kryptographischer Logik

Diese Schicht enthält selbst keine kryptographischen Berechnungen und fungiert ausschließlich als kontrollierter Vermittler zwischen Interfaces und Core.

2.1.3 Interface-Schichten
Lethesafe stellt zwei gleichwertige, aber technisch unterschiedliche Interface-Schichten bereit:

CLI-Suite

- Direkte Nutzung des kryptographischen Cores
- Lineare, synchron ausgeführte Workflows
- Keine persistente Zustandsverwaltung
- Vollständig offline nutzbar, geeignet für Skripte und One-File-Binaries

Web-Interface

- Browserbasierte Benutzeroberfläche
- Nutzung der Workflow- & Adapter-Schicht zur Ablaufsteuerung
- HTTP dient ausschließlich als lokales Transportmedium
- Keine kryptographische Logik im Frontend oder Web-Host

Beide Interfaces erzeugen identische Zeitkapseln und sind vollständig interoperabel. Sie sind für das Einlesen, Normalisieren und kanonische Serialisieren von Zeitkapseln verantwortlich und geben ausschließlich semantische Parameter an die Workflow-Schicht weiter.

2.1.4 Prozessbezogene Zustandsverwaltung
Für laufende Rechenprozesse existiert eine bewusst nicht-persistente Zustandsverwaltung:

- Fortschritt wird ausschließlich im Arbeitsspeicher gehalten
- Abbruchsignale werden pro Lauf eindeutig identifiziert
- Es existiert keine Möglichkeit zur Wiederaufnahme unterbrochener Prozesse

Diese Entscheidung ist sicherheitsrelevant:
Ein abgebrochener Lauf führt stets zum vollständigen Verlust des bisherigen Rechenaufwands.

Fortschrittsanzeigen dienen ausschließlich der Information über den aktuellen Stand eines laufenden Hashprozesses. Sie haben keinen Einfluss auf die kryptographische Berechnung, deren Geschwindigkeit oder deren Ergebnis.

2.2 Architekturprinzipien
Die Architektur von Lethesafe folgt folgenden Grundsätzen:

- Strikte Trennung von Kryptographie und Interaktion
- Keine impliziten Zustände oder versteckten Zwischenergebnisse
- Keine Persistenz sicherheitskritischer Zwischenstände
- Deterministisches Verhalten unabhängig vom Interface
- Volle Offline-Fähigkeit aller sicherheitsrelevanten Operationen

Durch diese Struktur bleibt die kryptographische Integrität von Lethesafe unabhängig von Implementierungsdetails der Benutzeroberfläche oder zukünftigen Erweiterungen der Interfaces.

Die Ausführung der sequentiellen Hashkette ist strikt auf den kryptographischen Core beschränkt. Weder CLI noch Web-Interface enthalten eigene Implementierungen kryptographischer Primitive oder alternative Berechnungspfade.

Frontend-Komponenten übernehmen ausschließlich:

- Eingabevalidierung
- Orchestrierung von Abläufen
- Anzeige von Fortschritt

Sie können weder Hashketten eigenständig berechnen noch kryptographische Parameter verändern oder Berechnungsschritte überspringen.

Damit existiert eine eindeutige und alleinige Quelle kryptographischer Wahrheit innerhalb des Systems.

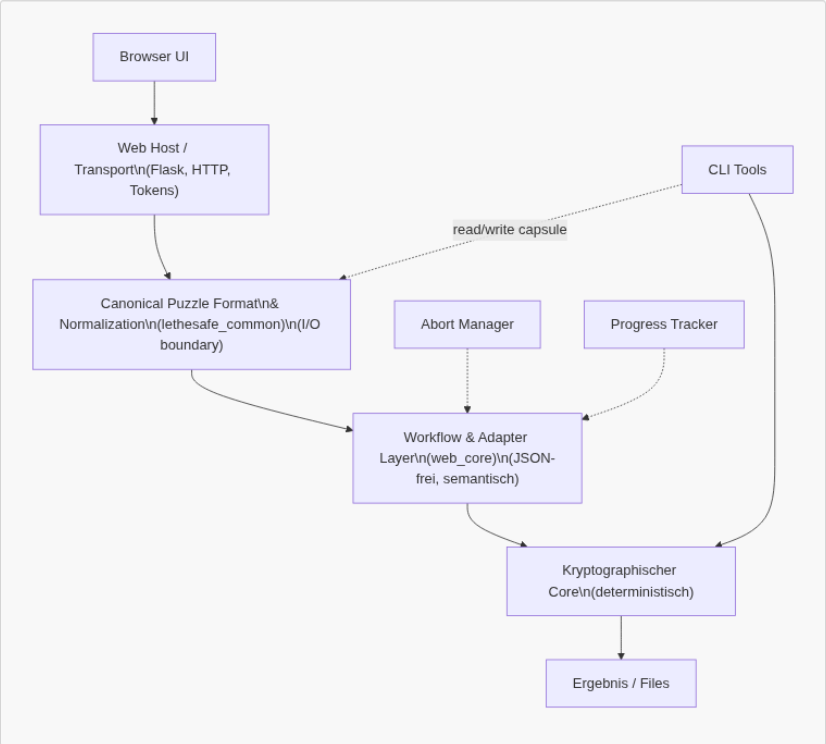

Abb. 1: Geschichtete Architektur von Lethesafe.
Web-basierte Interfaces nutzen eine zusätzliche Workflow- und Adapter-Schicht zur Ablaufsteuerung, während die CLI direkt mit dem kryptographischen Core interagiert. Alle sicherheitskritischen Berechnungen finden ausschließlich im Core statt; prozessbezogene Zustände sind nicht persistent.

3. Kryptographische Grundlagen

3.1 Zufallsquellen & Entropie

Alle sicherheitskritischen Funktionen von Lethesafe, einschließlich der Hashberechnungen, der Ableitung von Schlüsseln und der Erstellung von Zeitkapseln, werden im **Core** ausgeführt. Der **Web-Core** ist verantwortlich für die Kommunikation zwischen Frontend und Backend, stellt sicher, dass der Benutzer eine nahtlose Interaktion mit den Workflows hat, und überträgt die Daten an den Core, der die entsprechenden Berechnungen durchführt.

- create_capsules nutzt _derive_entropy (SHA-512 über Label, os.urandom und optionale Nutzerentropie) zur Initialisierung des Startwerts S sowie zur Ableitung der Zielzeichenkette K. Die Nutzerentropie erhöht dabei ausschließlich die Zufälligkeit des Initialzustands und hat keinen Einfluss auf die Dauer oder Struktur der Zeitverzögerung.
- Optional sammelt das Web-Frontend Mausbewegungen (create.html + JS) als zusätzliche Entropiequelle.

3.2 Hashketten & Ziele

- _hash_chain_targets berechnet deterministisch eine SHA-256-Kette über S, bis die maximale Rundenzahl erreicht ist.
- Die Funktion akzeptiert eine should_abort-Callback, sodass laufende Hashketten sicher beendet werden können.

3.3 Zielzeichenketten-Ableitung

Für jede Runde r gilt:

$H_r = SHA256^r(S)$
$Puzzle_r = K \oplus H^r$

Die Zielzeichenkette K ist an das jeweilige Hashziel H_r gebunden. Nur durch das vollständige sequenzielle Durchlaufen der Hashkette kann H_r rekonstruiert und daraus mittels XOR-Operation die Zielzeichenkette K gewonnen werden.

Die XOR-Operation dient ausschließlich der Bindung von K an das Hashziel und stellt keine Verschlüsselung semantischer Daten dar. Lethesafe verarbeitet zu keinem Zeitpunkt vertrauliche Informationen, sondern verzögert ausschließlich die Entstehung der Zielzeichenkette K.

Die Verwendung der XOR-Operation erfüllt eine strukturelle Funktion:
Sie trennt die Zielzeichenkette K konzeptionell vom Hashziel H_r.
H_r ist ausschließlich das Ergebnis eines zeitlich erzwungenen, sequenziellen Rechenprozesses, während K ein eigenständiger Zielwert bleibt, der erst durch die Kombination mit H_r rekonstruiert werden kann.

Diese Trennung ist Voraussetzung für das Klonen von Zeitkapseln:
Da K nicht mit H_r identisch ist, kann dieselbe Zielzeichenkette K mit unterschiedlichen Rundenzahlen erneut an neue Hashziele gebunden werden. Auf diese Weise lassen sich neue Zeitkapseln mit verändertem Zeitaufwand erzeugen, ohne den Zielwert selbst zu verändern.

Die XOR-Operation dient damit nicht der Verschlüsselung von Nutzdaten, sondern ausschließlich

der kontrollierten Bindung eines eigenständigen Zielwerts an einen zeitlich erzwungenen Rechenprozess.

3.4 Passwortgeschützter Startwert

_protect_start schützt S mit folgenden Bausteinen:

Schritt	Verfahren	Zweck
Key-Derivation	PBKDF2-HMAC-SHA256 mit 400.000 Iterationen	Resistenz gegen Brute-Force
Keystream	BLAKE2b (person=LethSenc)	One-Time-Pad für S
Authentizität	HMAC-SHA256 mit BLAKE2b-abgeleitetem Schlüssel (person=LethSmac)	Schutz vor Manipulation

Das resultierende Objekt start_value_protected enthält ciphertext, salt, nonce, iterations und mac (Base64). _decrypt_start verifiziert MACs via hmac.compare_digest und liefert S nur bei korrektem Passwort.

Die Iterationszahl ist bewusst als Parameter Teil des Puzzle-Formats, um zukünftige Anpassungen an Hardwareentwicklungen zu ermöglichen, ohne bestehende Kapseln zu invalidieren.

Die Verwendung eines Start-Passworts ist optional und dient ausschließlich dem Schutz vor unbefugtem Starten des Hashprozesses; sie hat keinen Einfluss auf die Dauer oder Sicherheit der Zeitverzögerung selbst.

3.5 Puzzle-Dateiformat (normativ)

Lethesafe definiert für `version = 2` ein kanonisches, eindeutig spezifiziertes Puzzle-Dateiformat. Alle Zeitkapseln dieser Version müssen diesem Format entsprechen. Unterschiede zwischen Erzeugungswerkzeugen (CLI, Web) dürfen die kryptographische Semantik nicht beeinflussen.

3.5.1 Kanonisches Format

Eine Zeitkapsel der Version 2 MUSS als JSON-Objekt mit den folgenden Top-Level-Feldern vorliegen:

```
{
  "format": "lethesafe-puzzle",
  "version": 2,
  "mode": "new" | "clone",
  "hash_function": "sha256",
  "rounds": <positive integer>,
  "puzzle_base64": "<Base64(Puzzler)>",
  "secret_checksum_hex": "<lowercase hex SHA256(K)>",
  "start_value_protected": { ... }
}
```

Normative Felddefinitionen:

`format`

Muss exakt den Wert "lethesafe-puzzle" haben.

version
Muss den Wert 2 haben.
Die Versionierung des Puzzle-Formats erlaubt spätere Erweiterungen, etwa alternative Hashfunktionen oder KDF-Parameter, ohne die Semantik bestehender Zeitkapseln zu verändern.

mode
Gibt an, ob es sich um eine neu erzeugte oder geklonte Zeitkapsel handelt.

hash_function
Für Version 2 fest auf "sha256" definiert.

rounds
Anzahl der sequenziellen Hashrunden (rounds ≥ 1).

puzzle_base64
Base64-kodierte Darstellung von *Puzzler*, definiert als
Puzzler = K ⊕ Hr, wobei Hr = SHA256^rounds(S).

secret_checksum_hex
SHA-256-Digest der Zielzeichenkette K, kodiert als lowercase Hexadezimalstring.
Das Feld secret_checksum_hex dient ausschließlich der Plausibilitätsprüfung nach erfolgreicher Rekonstruktion von K, insbesondere bei Klon-Läufen, und hat keinen Einfluss auf den kryptographischen Rechenprozess selbst.

start_value_protected
Objekt zur passwortbasierten Absicherung des Startwerts S.
Es enthält mindestens die Felder:
ciphertext, salt, nonce, iterations, kdf, mac, mac_function.

3.5.2 Kryptographische Semantik

Für jede Zeitkapsel gilt:

- Hr = SHA256^rounds(S)
- Puzzler = K ⊕ Hr
- K = Puzzler ⊕ Hr

Die XOR-Operation dient ausschließlich der **Bindung eines eigenständigen Zielwerts** K **an ein zeitlich erzwungenes Hashziel**.
Sie stellt **keine Verschlüsselung semantischer Nutzdaten** dar.

Die Trennung von K und Hr ermöglicht es, dieselbe Zielzeichenkette K mit unterschiedlichen Rundenzahlen erneut an neue Hashziele zu binden (Klonen), ohne den Zielwert selbst zu verändern.

3.5.3 Kanonische Kodierung & Normalisierung

- `puzzle_base64` **MUSS** strikt Base64-kodiert sein (keine Whitespaces, keine URL-safe Varianten).
- `secret_checksum_hex` **MUSS** lowercase sein und exakt 64 Hexzeichen (32 Byte) umfassen.
- Erzeugende Werkzeuge **MÜSSEN** die kanonischen Feldnamen verwenden.

3.5.4 Optional Metadaten

Zusätzliche, nicht sicherheitsrelevante Informationen **DÜRFEN** enthalten sein, sofern sie in einem separaten Objekt `meta` gekapselt sind:

```
"meta": {
  "created_utc": "2026-02-08T16:39:02Z",
  "rounds_mode": "round_based" | "time_based",
  "capsule_index": 1,
  "capsule_rounds": 20000
}
```

Regeln:

Metadaten **DÜRFEN** die kryptographische Semantik nicht beeinflussen.
Leser **MÜSSEN** unbekannte Metadaten ignorieren.

3.5.5 Abwärtskompatibilität (Reader-Normalisierung)

Lesende Implementierungen **DÜRFEN** aus Gründen der Abwärtskompatibilität folgende Legacy-Aliasse akzeptieren und beim Einlesen normalisieren:

- `puzzle` → Alias für `puzzle_base64`
- `secret_checksum` → Alias für `secret_checksum_hex` (wenn hex-kodiert)
- `start_value_protection`, `start_value_iterations` → Legacy-Felder ohne semantische Relevanz

Erzeugende Werkzeuge DÜRFEN diese Legacy-Felder nicht mehr ausgeben.

3.5.6 Verbindlichkeit

Eine Zeitkapsel mit `version = 2`, die nicht dem kanonischen Format entspricht, gilt als nicht konform.
Parser dürfen tolerant sein, **die Spezifikation ist es nicht**.

3.6 Implementierungshärtung

Die Referenzimplementierung validiert sämtliche strukturellen und semantischen Parameter strikt, bevor kryptographische Berechnungen beginnen.

Dies umfasst insbesondere:

- Exakte Übereinstimmung der Algorithmuskennungen mit den unterstützten Werten

- Prüfung numerischer Parameter auf positive Ganzzahlen
- Validierung Base64-dekodierter Felder auf exakt erwartete Byte-Längen
- Prüfung von Prüfsummen auf korrekte Digest-Länge
- Ausschluss impliziter Typumwandlungen oder stiller Fallback-Mechanismen

Die Implementierung folgt einem strikt fehlerschließenden (fail-closed) Modell.
Jede strukturelle oder semantische Abweichung führt zum sofortigen Abbruch.

Diese Validierungsmechanismen verändern weder das kryptographische Verfahren noch die Zeitverzögerungssemantik, sondern reduzieren ausschließlich Mehrdeutigkeiten auf Implementierungsebene.

4. Workflow-Design

4.1 Erstellung (/api/create)

Die Kommunikation zwischen der Benutzeroberfläche (Web/CLI) und den kryptografischen Berechnungen im **Core** erfolgt über den **Web-Core**, der als Vermittler fungiert. Die API-Endpunkte, die für die Erstellung, Klonung und Entschlüsselung von Zeitkapseln zuständig sind, sind dabei ausschließlich für die Kommunikation mit dem **Web-Core** zuständig, der die Berechnungen an den **Core** weiterleitet und die Ergebnisse verarbeitet.

1. Frontend validiert Eingaben, sammelt Entropie und initialisiert den Lauf; Zeitabschätzungen erfolgen ausschließlich während des laufenden Hashprozesses.
2. Nach Formular-Submit erzeugt der Browser ein Lauf-Token (crypto.randomUUID) und sendet dieses in X-Run-Token-Header.
3. Flask registriert das Token beim abort_manager, ruft create_capsules, cached Resultate (_RESULT_CACHE) und liefert Download-Tokens für [K] zur Verwendung als externes Masterpasswort, Passwortdatei und Puzzle-Dateien.
4. Browser bietet Dateien via /download/<token> als Blob an; eine passwortfreie zeitkapsel_secret.txt dient dem Offline-Export.

Zeitbasierter Modus (optional):
Alternativ zur direkten Angabe der Rundenzahl kann der Nutzer eine gewünschte Zeitdauer angeben. Das System führt vor dem eigentlichen Hashlauf einen kurzen lokalen Kalibrierungslauf durch, um die reale Hashrate des Geräts zu bestimmen und daraus die erforderliche Rundenzahl abzuleiten. Der eigentliche Hashprozess startet erst nach expliziter Bestätigung dieser Parameter.

4.2 Cloning (/api/clone)

- Eingeladene Puzzle-Datei wird geparst (_load_puzzle).
- unlock_capsule-Logik rekonstruiert S und K (erneuter Hashlauf über die Originalrunden).
- Nutzer entscheiden, ob das ursprüngliche Passwort weiterverwendet oder via _protect_start ersetzt wird.
- Neue Runden werden berechnet und als eigenständige JSON-Dateien exportiert.

4.3 Entschlüsselung (/api/unlock)

- Die API akzeptiert Puzzle-Datei + Passwort, validiert Größe (MAX_CONTENT_LENGTH = 32 * 1024 * 1024 in app.py) und ruft unlock_capsule.

- Das Ergebnis wird ausschließlich Base64-kodiert ausgeliefert. Optionaler Download liefert zeitkapsel_secret.txt mit [K] und Timestamp.

4.4 Laufabbruch & UX

- Frontend-Seiten nutzen AbortController und navigator.sendBeacon (lethesafe.js) um /api/run/cancel zu triggern.
- Progressbars fragen /api/run/progress/<token> ab und berechnen die ETA ausschließlich aus den tatsächlich abgeschlossenen Runden (keine persistente Hashraten-Schätzung).
- Navigation wird solange blockiert, bis der Lauf abgeschlossen oder abgebrochen ist (BeforeUnload-Guard).

Der Rechenprozess speichert zu keinem Zeitpunkt persistente Zwischenstände.
Fortschritt existiert ausschließlich im Arbeitsspeicher des laufenden Prozesses.
Ein Abbruch – unabhängig von Ursache oder Zeitpunkt – führt zum vollständigen Verlust des bisherigen Rechenaufwands; eine Wiederaufnahme ist nicht möglich.

5. Sicherheitsbetrachtung

Angriff/Fehlerfall	Gegenmaßnahme
Gestohlene Puzzle-Datei	Hashlauf zwingend nötig; Startwert ist ohne Passwort nicht rekonstruierbar.
Manipulierte JSON-Struktur	maker.py validiert Felder, _load_puzzle prüft Format und Base64-Dekodierbarkeit.
Brute-Force auf Passwort	PBKDF2 mit 400k Iterationen und BLAKE2b-MAC erhöht Aufwand; unbegrenzte Eingabeversuche sollen legitime Anwender nicht ausbremsen, die Sicherheit der Zeitverzögerung ist unabhängig von der Wahl oder Existenz eines Start-Passworts.
Fehlhafte Upload-Requests	Upload-Größenlimits und Validierung von request.files verhindern die Verarbeitung unvollständiger oder missbräuchlicher HTTP-Anfragen.
Nutzerinitiierter Laufabbruch	Run-Tokens + should_abort Callback erlauben einen jederzeitigen, kontrollierten Abbruch laufender Hashprozesse ohne Seiteneffekte oder Speicherlecks.
Wiederverwendung identischer Dateinamen	CLI (lethesafe_maker.py) erzeugt eindeutige Namen (build_filename). Web-Ausgaben sind tokenisiert und überschreiben nichts.

Result Leakage via Cache	_RESULT_CACHE lebt nur im Prozess und wird durch Browser-Downloads geleert (kein persistent Storage).

Threat Model Annahmen - Der Client kann die Laufzeit nicht beschleunigen, da der verwendete SHA256-Hashlauf inhärent sequenziell ist. - Sobald [K] einmal offengelegt wurde, gilt es als temporär existent und muss nach der beabsichtigten Verwendung vollständig vernichtet werden. Jede weitere Existenz – sei es als Datei, Kopie oder gespeicherte Zeichenfolge – hebt den zeitlichen Schutzmechanismus vollständig auf. - Hardware-Seitenkanäle (z. B. CPU Voltage-Glitches) liegen außerhalb des aktuellen Scopes.

Die zeitbasierte Parametrisierung stellt keine zusätzliche Angriffsfläche dar, da sie ausschließlich lokal erfolgt und ausschließlich zur Berechnung der Rundenzahl dient.

6. CLI-Tools & Offline-Modus

CLI-Suite (cli/lethesafe_maker.py, cli/lethesafe_unlocker.py) – stellt die Offline-Tools zur Erstellung, zum Klonen und zum Entschlüsseln von Zeitkapseln für Experten bereit. Diese Tools basieren auf denselben Workflows wie die Web-Version, aber ohne HTTP-Interface, und ermöglichen **One-File Builds** für lokale und serverseitige Implementierungen. Die CLI-Tools delegieren sämtliche kryptographischen Berechnungen vollständig an den Core und enthalten selbst keine Hashketten- oder Schlüsselableitungslogik.

6.1 lethesafe_maker.py

- Vollständig textbasiert, erlaubt „Neu“ und „Clone“-Modus, zeigt [K] an und schreibt wahlweise Secrets auf Disk.
- Enthält Komfortfunktionen wie prompt_rounds, sanitize_basename, compute_hashes_for_rounds und userfreundliche Emojis.
- Unterstützt optional eine zeitbasierte Verzögerungseingabe (z. B. 6h, 3d) mit lokaler Kalibrierung der Hashrate.
- Die berechnete Rundenzahl wird vor Start angezeigt und als regulärer Parameter in der Zeitkapsel gespeichert.
- Exportiert Metadaten (mode, source_file, secret_checksum), damit Web- und CLI-Dateien kompatibel bleiben.
- Zeigt eine Live-ETA basierend auf der tatsächlich erreichten Rundenzahl und erlaubt unbegrenzt viele Passwortversuche (Fehleingaben führen lediglich zu einer erneuten Abfrage).

6.2 lethesafe_unlocker.py

- Fragt das Zeitkapsel-Passwort ab, validiert Startwerte und speichert [K] optional als key_<rounds>r.txt inklusive UTC-Timestamp.
- Verwendet dieselbe Live-ETA-Ausgabe wie der Maker und begrenzt Passwortversuche nicht.

7. Betriebs- & Deployment-Aspekte

- **Lokalbetrieb:** FLASK_APP=app.py flask run im Verzeichnis lethesafe_web reicht aus; keine DB oder externen Dienste nötig.

- **Result-Downloads:** _RESULT_CACHE speichert (Dateiname, Bytes) im Speicher. Downloads erfolgen über /download/<token> mit send_file.
- **Ressourcenbegrenzung:** MAX_CONTENT_LENGTH verhindert DoS über riesige Uploads, Hashketten laufen rein CPU-bound.
- **Internationalisierung:** Default-Sprache ist Deutsch; Strings sind in Templates/JS hardcodiert. Mehrsprachigkeit ist als künftige Erweiterung angedacht.

8. Performance & Nutzererlebnis

- Browserseitige ETA-Anzeigen basieren ausschließlich auf Live-Daten aus dem Progress-Tracker (completed_rounds, elapsed_time). Historische Hashraten oder localStorage-Schätzungen werden nicht mehr verwendet.
- Zeitbasierte Verzögerungen verwenden eine kurze Vorabmessung der Hashrate; Fortschrittsanzeigen und ETA während des eigentlichen Hashlaufs basieren weiterhin ausschließlich auf real abgeschlossenen Runden.
- Fortschrittsanzeigen zeigen getrennte Phasen (z. B. clone.html unterscheidet Original-Hashlauf und neue Kapseln) und aktualisieren sich in Echtzeit über /api/run/progress/<token>.
- Passwort-Eingaben sind bewusst nicht limitiert; falsche Eingaben resultieren lediglich in einer erneuten Abfrage, sowohl im Web-Frontend als auch in den CLI-Tools.
- Mausentropie-Aufzeichnung limitiert sich auf 200 Punkte und wird gedrosselt (alle 40 ms), um UI-Responsiveness zu erhalten.

9. Roadmap & Erweiterungsmöglichkeiten

1. **CLI-Automatisierung:** Parameterbasierte Nutzung ohne Prompting (Batch-Skripte, CI).
2. **Erweiterte Validierung:** Zusätzliche Format-/Integritätschecks vor langen Hashläufen (z. B. JSON-Schema oder Signaturen).
3. **Datei-Workflow-Härtung:** Optionale Vorabdefinition des Secret-Speicherorts (Schutz gegen Ausfälle am Laufende).
4. **UPX/Packaging-Optimierung:** Kleinere Binaries für Ressourcenkontexte.
5. **Kryptographische Agilität:** Austauschbare Hashfunktionen oder zukünftige VDF-Module, sobald erforderlich.
6. **Mehrsprachige UI & Dokumentation:** Internationalisierung der Templates und JS-Strings.

10. Fazit

Lethesafe verbindet bewährte Kryptographie (SHA256, PBKDF2, BLAKE2b, HMAC) mit praxisnahen Bedienkonzepten. Sowohl Web- als auch CLI-Tools garantieren, dass Zeitkapseln komplett lokal entstehen, geklont und entschlüsselt werden – ohne Vertrauen in den Hersteller. Das Projekt ist bereit für Audits, Community-Beiträge und produktive Nutzung auf Desktops oder spezialisierten Devices. Nächste Schritte bestehen darin, Automatisierungsschnittstellen, zusätzliche Validierung und optionale Mehrsprachigkeit zu liefern, um Lethesafe als Referenzimplementierung für Time Lock-Puzzles zu etablieren.

Dieses Dokument beschreibt die technische Architektur, die kryptographischen Grundlagen und die Sicherheitsannahmen von Lethesafe in ihrem stabilen konzeptionellen Zustand. Änderungen an Benutzeroberfläche, Implementierungsdetails oder Performance-Optimierungen haben keine Auswirkungen auf die hier beschriebenen Prinzipien

11. Dokumenthistorie

Version	Datum	Beschreibung
1.0	18.02.2026	Erste offiziell versionierte Fassung