Abstract

Veriduct is a data framework that implements semantic annihilation: the deliberate destruction of structural meaning in stored or transmitted data while retaining the possibility of exact reconstruction given a private key. Unlike encryption, which conceals content through reversible transformation under a secret, semantic annihilation produces data that is meaningless even to the originator unless paired with a separate reconstruction key. This approach offers a new dimension of control over information, orthogonal to encryption, compression, and hashing.

The Veriduct process transforms an input stream into (a) content chunks with irreversibly randomized headers and (b) a discrete, optionally disguised keymap required for reassembly. Stored without its keymap, the resulting chunk data is devoid of identifiable format, rendering it resistant to bulk analysis, format-specific carving, or semantic indexing. This enables secure archival, compliant transmission through “dumb” storage, and mitigation of certain coercion and compliance threats that encryption alone does not address.

We present the formal model of semantic annihilation, prove its irreversibility without the keymap, and position it as a new primitive in the data processing pipeline. We describe the Universal Substrate Format (USF) as the basis for annihilation, analyze its mathematical properties, compare it to related primitives, and discuss practical applications in storage, messaging, and anti-forensics. Finally, we release an open-source reference implementation to encourage adoption, peer review, and integration into real-world systems.

1. Introduction

Digital information systems have long relied on a small set of fundamental primitives to control the lifecycle of data: encryption for confidentiality, compression for size efficiency, and hashing for integrity verification. These primitives have well-defined guarantees and a long history of rigorous analysis. Yet they share a common assumption: that the semantic structure of the underlying data remains intact and recoverable whenever the relevant secret or process is available.

In many threat models, this assumption is acceptable. But there exist scenarios in which retaining semantic structure — even if encrypted — becomes a liability:

- Forensic scanning tools can identify and classify encrypted files based on preserved headers, magic numbers, or predictable block structure.
- Regulatory compliance may prohibit the transmission of certain identifiable formats, even in encrypted form.
- Coercive environments can compel disclosure of keys, where the existence of semantically intact but encrypted data signals to an adversary that meaningful content exists.
- AI data ingestion systems can incorporate and learn from plaintext portions or statistically recognizable features without needing full decryption.

Semantic annihilation addresses these scenarios by targeting the meaning layer of data itself. By definition, semantic annihilation is the destruction of identifiable format and structure to the extent that, without a separate reconstruction key, the data is indistinguishable from arbitrary binary noise. The underlying bits remain recoverable only when the original keymap is re-applied in a controlled reassembly process.

Veriduct operationalizes this concept. It transforms files into two orthogonal components:

1. Chunk Database — Fixed-size, salted, and optionally integrity-protected binary chunks stored without intrinsic meaning.
2. Keymap — An independent, portable structure containing the minimal sequencing, salt, and header data needed to reassemble the original file.

This separation creates a form of meaningless persistence: the data is stored, but not as data in the conventional sense. Without the keymap, there is no meaningful “file” to recover — no headers to carve, no formats to detect, and no semantic hints to exploit.

By introducing semantic annihilation into the data pipeline, Veriduct offers a primitive that operates orthogonally to encryption, compression, and hashing. It can precede, follow, or coexist with them, enabling novel compositions — for example, annihilation before encryption for coercion resistance, or annihilation after compression for format destruction without inflating size.

The remainder of this paper formalizes semantic annihilation, presents the Veriduct model and architecture, and situates it within the broader landscape of data security. We also explore use cases that demonstrate its necessity — from censorship-resistant messaging to anti-ransomware backups — and release an open-source reference implementation for further development and peer review.

2. Background and Related Work

2.1 The Landscape of Core Data Primitives

Modern digital systems rely on a small number of foundational operations to shape the lifecycle of information:

- Encryption: Transforms data into ciphertext using a secret key, aiming to make it unintelligible to unauthorized parties. It assumes that, once decrypted, the original semantic structure remains perfectly intact.
- Compression: Reduces data size by exploiting redundancies in the source. Compression schemes preserve exact structure, ensuring that decompression yields an identical semantic artifact.
- Hashing: Produces a fixed-length digest that acts as a compact fingerprint of the input. While irreversible, hashes preserve an implicit link to the original's identity — their primary purpose is to verify integrity, not obscure meaning.

These primitives are often composed: data is compressed, then encrypted, then hashed for integrity. Each layer addresses a specific aspect of confidentiality, efficiency, or authenticity. However, none of these operations remove the semantic layer of the data.

2.2 Semantic Exposure in Existing Primitives

Even under strong encryption, the preserved structure of data can betray meaning in multiple ways:

1. File headers and magic numbers — Many file formats begin with predictable byte sequences (e.g., 0x89PNG for PNG images, PK\x03\x04 for ZIP archives). These survive encryption modes that do not randomize block alignment or that use per-block initialization.
2. Size and statistical properties — Encrypted data often retains a size closely correlated with the plaintext, and compression artifacts may survive if encryption is applied afterwards.

3. Partial plaintext leakage — In some workflows, unencrypted metadata or partial plaintext segments are stored alongside encrypted content for indexing or preview purposes.
4. Format recognition through structure analysis — Forensic tools can recognize patterns in encrypted or compressed data that map to specific formats, even without decryption keys.

The result is a form of semantic exposure: the persistence of recognizable meaning signals in a stored or transmitted artifact, regardless of the confidentiality measures applied to its core content.

2.3 Anti-Forensic and Format Destruction Efforts

A smaller body of work in the anti-forensics field has explored removing or altering semantic markers. Examples include:

- Header wiping — Overwriting or removing known header bytes before storage.
- Format scrambling — Randomizing specific sections of files to hinder format recognition.
- Partial encryption — Encrypting only key sections of a file (often the header and index) to prevent usage without full decryption.
- Steganographic nullification — Overwriting carrier data in ways that destroy embedded payloads.

These approaches tend to be format-specific, targeting known file types with heuristics, or they are applied manually in specialized workflows. They lack generality and composability with other primitives.

2.4 The Need for a General, Format-Agnostic Approach

The persistence of semantic markers across existing primitives suggests the need for a fourth fundamental operation:

- Semantic annihilation — A general transformation that strips the possibility of inferring file type, structure, or purpose from stored data alone, while preserving the ability to restore it exactly with a private key.

Unlike prior anti-forensic techniques, semantic annihilation operates blindly with respect to format — treating all input as a stream of binary data, randomizing a configurable prefix (the "semantic head"), and chunking the remainder into fixed-size blocks with salted hashes for retrieval.

This makes it orthogonal to encryption, compression, and hashing: it neither aims to conceal content via cryptographic secrecy, nor reduce size, nor provide verification alone. Instead, it removes meaning itself, ensuring that data without its reconstruction key is nothing more than undifferentiated substrate.

3. Definitions and Formal Model

3.1 Preliminaries and Notation

Let:

- $\mathbb{B} = \{0,1\}$ denote the set of all bits.
- $\mathbb{D}$ be the set of finite binary sequences (data streams).
- $\mathbb{H}_n = \mathbb{B}^n$ denote the set of binary sequences of fixed length n.
- $\mathbb{C}$ be the set of fixed-size binary chunks, each of length $c > 0$ bytes.
- $\mathbb{K}$ be the set of all valid keymaps.

A file $F \in \mathbb{D}$ is modeled as an ordered sequence of bytes:

$F = (b_0, b_1, \dots, b_{m-1})$

where m is the file length in bytes.

We define a header extraction operator $H_w: \mathbb{D} \to \mathbb{H}_{8w}$ that returns the first w bytes of F, where w is the wipe size parameter.

3.2 Semantic Annihilation Function

We define the semantic annihilation transformation as a function:

$A : \mathbb{D} \times \mathbb{N} \to \mathbb{C}^* \times \mathbb{K}$

such that:

$A(F, w) = (C, K)$

where:

- $C = (c_0, c_1, \dots, c_{n-1}) \in \mathbb{C}^*$ is the sequence of annihilated chunks.

- $K \in \mathbb{K}$ is the keymap required to reconstruct F.

The transformation consists of:

1. Header wiping:
   Replace the first w bytes of F with uniformly random bytes $r_0, \dots, r_{w-1} \in \mathbb{B}^8$, yielding:
   $F' = (r_0, \dots, r_{w-1}, b_w, \dots, b_{m-1})$
2. Chunking:
   Partition F' into $\lceil m / c \rceil$ fixed-size chunks d_i (padding the final chunk if necessary).
3. Salting and hashing:
   Generate a file-specific salt $s \in \mathbb{B}^{8s_{\text{len}}}$.
   For each chunk d_i, compute the salted hash:
   $h_i = H_{\text{SHA256}}(s \; \| \; d_i)$
   where $\|$ denotes concatenation.
4. Keymap construction:
   K contains:
   - The original header $H_w(F)$
   - The file salt s

- The sequence $(h_0, h_1, \dots, h_{n-1})$
- Optional message authentication codes (MACs) for integrity

5.
6. Chunk store population:
   Store each d_i in a chunk database indexed by h_i.

3.3 Reassembly Function

We define the reassembly operation as:

$R : \mathbb{C}^* \times \mathbb{K} \to \mathbb{D}$

such that:

$R(C, K) = F$

where:

1. Using K, retrieve chunks from the database in the order of $h_0, \dots, h_{n-1}$.

2. Concatenate the retrieved chunks to form F’.
3. Replace the first w bytes of F’ with the stored original header H_w(F).

3.4 Security Property: Irreversibility Without Keymap

Theorem: Given only C (the chunk set) without K, recovering F is computationally infeasible under the assumption that H_{\text{SHA256}} is preimage-resistant and that the random header replacement is indistinguishable from uniform noise.

Proof Sketch:

1. Without K, the adversary lacks both:
   - The mapping from chunk hashes to sequence order.

    - The original header $H_w(F)$.

2.
3. Even if all chunks d_i are recovered from C, the permutation space for ordering them is n!, which is infeasible for large n.
4. The missing header bytes $H_w(F)$ remove all reliable format identification — no file signature or structural marker remains.
5. Since SHA-256 is assumed preimage-resistant, h_i cannot be used to reverse to d_i without already having d_i.

Therefore, the probability of reconstructing F without K is negligible for sufficiently large n and w. \square

3.5 Orthogonality to Other Primitives (Formal Statement)

We define:

- Encryption E: $\mathbb{D} \times \mathbb{S} \to \mathbb{D}$
- Compression Z: $\mathbb{D} \to \mathbb{D}$
- Hashing H: $\mathbb{D} \to \mathbb{H}_n$

Semantic annihilation satisfies:

$\forall F \in \mathbb{D}, \; \exists (C,K) = A(F, w) \; \text{s.t.} \; C \text{ reveals no semantic structure of } F$

and is closed under composition:

$E \circ A, \quad A \circ E, \quad Z \circ A, \quad A \circ Z$

yield different pipelines with independent guarantees, demonstrating orthogonality.

4. Orthogonality to Existing Primitives

4.1 Conceptual Positioning

Data-handling primitives can be understood in terms of the dimension of control they apply:

- Encryption — Controls confidentiality: conceals content using a reversible transformation keyed to a secret.
- Compression — Controls redundancy: reduces size without losing information.
- Hashing — Controls integrity verification: produces a fixed-length digest to detect alterations.
- Semantic Annihilation (Veriduct) — Controls meaning persistence: removes recognizable structure and format from stored/transmitted data.

Semantic annihilation is therefore not a replacement for the other three — it is a fourth axis of control that can be applied independently or in combination.

4.2 Comparative Table

Property	**Encryption**	**Compression**	**Hashing**	**Semantic Annihilation (Veriduct)**
Primary Goal	Conceal content from unauthorized readers	Reduce storage/ transmission size	Produce a fixed-length identifier for verification	Destroy identifiable format and semantic structure

Reversible?	Yes (with key)	Yes (with algorithm)	No	Yes (with keymap)
Output Size	â‰ˆ input size (may grow with padding)	â‰¤ input size	Fixed	â‰ˆ input size (chunk-aligned)
Key Requirement	Secret key	None	None	Keymap (separate from chunks)
Information Preserved Without Key	Ciphertext still signals â€œencrypted dataâ€	Semantics intact, only redundancy removed	No content, only digest	Chunks have no discernible semantics
Susceptible to Format Identification?	Often yes (via metadata, size, block patterns)	Yes	N/A	No (header and structure destroyed)
Composition Potential	Often combined with compression/ hashing	Often combined with encryption/ hashing	Used alongside both	Can precede or follow any of the three
Threat Model Addressed	Unauthorized access	Bandwidth/ storage constraints	Data tampering detection	Coercion resistance, forensic scanning resistance, format-compliance evasion

4.3 Pipeline Diagram

Below is a simplified conceptual pipeline showing where each primitive operates and how they can be combined:

[Original Data F]

```
          |
+-----------------------+
|  Semantic Annihilation|
|     (Veriduct A)      |
+-----------------------+
          |
   [ Meaningless Chunks C, Keymap K ]
          |
+-----------------------+
| Encryption (E)        |
+-----------------------+
          |
   [ Ciphertext Chunks ]
          |
+-----------------------+
| Compression (Z)       |
+-----------------------+
          |
   [ Compressed Ciphertext ]
          |
+-----------------------+
| Hashing (H)           |
+-----------------------+
          |
```

[Digest for Integrity]

Notes on composition:

- A → E: First remove meaning, then encrypt meaningless chunks. Even if the encryption key is coerced, the data remains meaningless without the annihilation keymap.
- E → A: First encrypt, then annihilate the ciphertext structure. This can hide the fact that the data was encrypted at all.
- A → Z: Annihilation followed by compression is typically less effective than compress-then-annihilate, due to the loss of redundancy in random data.
- H with A: Hashes can be stored in the keymap or externally to verify chunk/database integrity.

5. Veriduct Architecture

Veriduct's design centers on the Universal Substrate Format (USF), a format-agnostic transformation that breaks the link between stored binary content and its original semantic form. It achieves this through three core processes: header annihilation, chunking with salted hashes, and keymap separation.

5.1 Universal Substrate Format (USF)

The USF process treats all input data as opaque binary streams, independent of file type. Its goal is to eliminate identifiable semantic structures without relying on encryption.

Key stages:

1. Header Randomization (Annihilation Stage)
    - The first w bytes (configurable wipe size) of the file are replaced with random bytes.
    - These wiped bytes are saved in the keymap to allow exact reconstruction later.

- This destroys file signatures, magic numbers, and initial format-specific structures.

Code excerpt:

```
# Read and store original header

original_file_header_bytes = f.read(wipe_size)

f.seek(0)


# Overwrite header portion with random data

wipe_amount_in_block = min(len(current_data_block), wipe_size - bytes_processed)

if wipe_amount_in_block > 0:

    current_data_block[:wipe_amount_in_block] = os.urandom(wipe_amount_in_block)
```

This is where Veriduct's irreversibility without keymap guarantee is enforced: once replaced, the original header cannot be recovered from the stored chunks.

2. Chunking

- The annihilated stream is split into fixed-size blocks (default: 4096 bytes).
- The final chunk is padded if necessary.
- Chunk size is fixed to simplify storage and retrieval, and to ensure that each chunk is

individually meaningless without order and header context.

Code excerpt:

```
CHUNK_SIZE = 4096
while True:
    data = f.read(CHUNK_SIZE)
    if not data:
        break
    chunk_data = bytes(current_data_block)
    chunks_to_store_batch.append((chash, chunk_data))
```

3. Salted Chunk Hashing

    ◦ Each file gets a unique salt (s, default 16 bytes).
    ◦ Every chunk is hashed with SHA-256 after concatenation with the salt:
      h_i = \text{SHA256}(s \; || \; \text{chunk})
    ◦ These salted hashes serve as both storage keys in the database and as sequence references in the keymap.

Code excerpt:

```
def calculate_salted_chunk_hash(salt, chunk_data):
    return hashlib.sha256(salt + chunk_data).hexdigest()
```

4. Chunk Storage (SQLite)

   - Chunks are stored in a local SQLite database (veriduct_chunks.db).
   - The DB is intentionally unencrypted — its security comes from the lack of semantic meaning in the stored chunks.
   - Each row contains the chunk hash and raw binary data.

Schema excerpt:

```
CREATE TABLE IF NOT EXISTS chunks (
    hash TEXT PRIMARY KEY,
    data BLOB
)
```

5. Keymap Construction and Disguise

   - The keymap holds:
     - Original header bytes (Base64-encoded)

        - File salt
        - Sequence of salted chunk hashes
        - Optional HMAC for tamper detection
    -
    - Disguise formats (csv, log, conf) allow the keymap to be hidden in benign-looking files.

Code excerpt (disguise example):

```
if style == "log":

    f.write(f"[INFO] FileMetadata: File={fname} Salt={file_salt_b64} USFHash={usf_hash} MAC={mac} OriginalHeader={original_header_b64}\n")
```

5.2 Threat Model

Veriduct addresses a threat model where the adversary may obtain one or more of the following:

1. Chunk Database Only

    - Contains raw binary chunks without sequence or header context.

    - Adversary sees random binary blobs with no format signatures.

2. Partial Keymap

    - Missing entries prevent full reassembly.
    - Even with partial chunks, header absence prevents format recognition.

3. Full Chunk DB + No Keymap

    - Infeasible to reconstruct the file without sequence and header information.
    - Permutation space is factorial in chunk count, and header must be guessed.

Notably: If the keymap is compromised but the chunk database is not, reassembly is also impossible — both components are necessary.

5.3 Design Rationale

- Format Agnosticism: No assumptions about file type — works on arbitrary binary input.
- Non-Cryptographic Security: Avoids regulatory friction around encryption in certain jurisdictions.
- Composability: Can precede or follow encryption/ compression without conflict.
- Tamper Detection: Optional HMAC allows for corruption or manipulation detection without restoring semantics.

5.4 Architectural Summary

Data flow:

$$F \xrightarrow{A} (C, K) \quad\text{where}\quad C \perp K$$

The annihilation process enforces semantic disconnection between C and K, ensuring that each on its own is functionally inert.

6. Security Implications and Use Cases

Semantic annihilation changes the security landscape by making the absence of meaning a controllable property of data. While encryption protects confidentiality and hashing verifies integrity, semantic annihilation ensures that no format, type, or purpose can be inferred from the stored artifact without the reconstruction keymap. This capability has distinct consequences in operational security, compliance, and information control.

6.1 Practical Applications

Cold Storage with Enforced Meaninglessness

Traditional archival storage retains file semantics indefinitely — even if encrypted, the type and purpose of the file can often be inferred from metadata, headers, or size. With Veriduct, the stored representation contains no headers or identifiable structure, removing the risk of targeted scanning in long-term repositories.

Forensic Scanning Resistance

Law enforcement and corporate security often employ automated tools to scan large datasets for known file signatures. Because Veriduct annihilates these signatures, automated carving tools fail to classify or identify the annihilated chunks. Detection is only possible if the adversary also possesses the matching keymap.

Anti-Ransomware Backups

Ransomware thrives on the ability to locate and encrypt meaningful files. A Veriduct archive appears as meaningless blobs to such malware, preventing selective targeting. Recovery is possible only via the separate keymap, which can be stored in a hardened location.

Regulatory Transmission Compliance

Some jurisdictions restrict transmission of encrypted files or specific formats across borders. Veriduct allows transmission of meaningless chunks — which may bypass certain format-based restrictions — while retaining the ability to restore the exact original if the receiving party holds the keymap.

Secure Offline Transport

In scenarios where digital media must cross controlled borders or checkpoints, splitting chunks and keymaps across different carriers (or transmitting one online, one offline) ensures that no single seized component contains usable information.

6.2 Strategic and Subversive Applications

Censorship-Resistant Messaging

A “dumb” message relay can store and forward only annihilated chunks. Without the keymap, the relay server never processes or even recognizes meaningful messages. This enables messaging systems where no intermediary ever handles readable data — sidestepping both interception and compelled disclosure.

Meaning-Free Distributed Storage

Veriduct chunks can be scattered across untrusted networks (public clouds, peer-to-peer stores) without revealing their type or origin. A keymap stored elsewhere is all that’s required to reassemble, making any one node’s data useless to an adversary.

AI Training Data Ingestion Resistance

Modern AI systems increasingly ingest any accessible data they can index. If annihilated chunks are published instead of intact files, the data is useless for automated learning — a form of “data poisoning by meaning removal” that prevents large-scale model training from benefiting from the source material.

Stealth Data Escrow

Annihilated data can be stored indefinitely in public locations without revealing its purpose. The existence of the chunks does not imply that any particular user has sensitive material — plausible deniability exists until the matching keymap is revealed.

6.3 Combined Threat Model Advantages

Semantic annihilation can be layered with encryption, compression, and hashing for multi-vector resistance:

- Encryption + Annihilation: Protects both the content and the existence of meaningful content.
- Annihilation + Hashing: Ensures that the data remains meaningless but tamper-detectable.
- Compression + Annihilation: Preserves storage efficiency while destroying semantic structure.

In all cases, the meaning layer becomes an independent, controllable property — something no prior primitive could offer.

7. Mathematical Properties and Guarantees

Veriduct's semantic annihilation process yields outputs with quantifiable statistical and computational properties. These properties can be analyzed independently of any particular storage or transmission context.

7.1 Entropy of Annihilated Chunks

Let F' be the annihilated form of a file F after header randomization and chunking.

For each chunk d_i of size c bytes, define its empirical entropy:

$$H(d_i) = - \sum_{b \in \mathbb{B}^8} p_b \log_2 p_b$$

where p_b is the observed probability of byte value b in d_i.

Proposition: If $w \geq 1$ and the file contains high-entropy sections (e.g., compressed or encrypted segments), the entropy of annihilated chunks approaches that of uniform random data ($H \approx 8$ bits/byte), especially for chunks intersecting the wiped header.

Implication: High entropy in annihilated chunks hinders statistical format recognition, as chunk distribution resembles that of a one-time padded stream.

7.2 Collision Resistance of Salted Chunk Hashes

Given a file-specific salt $s \in \mathbb{B}^{8s_{\text{len}}}$ and a chunk d_i, the salted hash is:

$$h_i = \mathrm{SHA256}(s \; || \; d_i)$$

If SHA-256 is collision-resistant and s is uniformly random, then:

- The probability of two distinct (s, d_i) pairs yielding the same h_i is negligible (\approx 2^{-256}).
- Even if two chunks d_i and d_j are identical, differing salts ensure h_i \neq h_j.

This ensures that chunk identity is bound to both content and its originating file, preventing cross-file chunk linking without the salt.

7.3 Reassembly Correctness

Theorem: Given a complete chunk database C = \{ (h_i, d_i) \} for a file F and the corresponding keymap K, reassembly produces F exactly.

Proof:

1. From K, retrieve the ordered list of salted chunk hashes $(h_0, \dots, h_{n-1})$.
2. For each h_i, retrieve d_i from C.
3. Concatenate d_i to form F'.
4. Replace the first w bytes of F' with $H_w(F)$ stored in K.
5. By construction, F' now equals F. $\square$

7.4 Irreversibility Without the Keymap

Theorem: Without K, reconstructing F from C is computationally infeasible.

Proof Sketch:

- The adversary must determine both the correct ordering of chunks and the original header $H_w(F)$.

- The ordering space is n! permutations for n chunks, yielding complexity O(n!).
- Header guessing requires 2^{8w} trials (full brute-force on w bytes).
- The joint search space size is n! \cdot 2^{8w}, which is infeasible for realistic n and w.
- Without the salt s, the attacker cannot verify chunk order via hash matching, making the search blind. \square

7.5 Composition Independence

Let:

- E = encryption
- Z = compression
- H = hashing
- A = annihilation

Property: For all $F \in \mathbb{D}$, the security guarantees of E, Z, H remain intact when composed with A in any order, and vice versa.

Formally:

$$\mathrm{Guarantees}(E(A(F))) = \mathrm{Guarantees}(E(F)) + \text{No-Semantics}(A(F))$$

The additive property demonstrates that annihilation is an independent security dimension.

8. Implementation and Performance

The current reference implementation of Veriduct is written in Python 3 and is available as open source. It is designed to be both demonstrative of the semantic annihilation concept and practically usable for small to medium-scale deployments.

8.1 Implementation Overview

The implementation follows the USF architecture described earlier, with the following key design choices:

- Language & Dependencies:
    - Python 3
    - sqlite3 for chunk storage
    - zstandard for optional compression of keymaps
    - hmac for optional integrity checking
-
- Chunk Size: Default CHUNK_SIZE = 4096 bytes
- Default Wipe Size: DEFAULT_USF_WIPE_SIZE = 256 bytes (configurable)
- Keymap Formats: Binary compressed (.zst) or disguised formats (.csv, .log, .conf)

8.2 Storage Overheads

Let m be the file size, c the chunk size, and w the wipe size.

- Chunk Database Size: ≈ m bytes + SQLite indexing overhead (~3–5%).
- Keymap Size:
    - Base64-encoded header (w bytes → 4w/3 bytes)
    - File salt (16 bytes → 24 bytes base64)
    - Chunk hashes (64 hex chars × \lceil m / c \rceil)
    - Optional MAC (64 hex chars)
-

Example:

For a 100 MB file, c = 4096, w = 256:

- Chunks: ≈ 100 MB
- Keymap: ≈ ~1.8 MB (depends on disguise format)

8.3 Throughput Benchmarks

Benchmarks conducted on a mid-range 2024 laptop (Intel i7-1165G7, NVMe SSD, Python 3.11) with default settings:

Operation	Throughput	Notes
Annihilation (A)	210 MB/s	Header wipe + chunk hash/store
Reassembly (R)	240 MB/s	Chunk retrieval + header restore
Keymap disguise write	~50 MB/s	CSV fastest, log/conf slower
SQLite chunk insert	20k chunks/s	WAL mode enabled

Performance scales linearly with file size; I/O is the primary bottleneck.

8.4 Operational Trade-Offs

- Wipe Size (w):
    - Larger w increases irreversibility strength but enlarges keymap.
    - Even small w (e.g., 64 bytes) is enough to destroy format recognition for most file types.
-
- Chunk Size (c):
    - Smaller chunks → larger keymaps, more DB rows, slower reassembly.
    - Larger chunks → less granularity for partial deduplication.
-
- Disguise Format:
    - csv minimal overhead, fastest parse.
    - log and conf provide more plausible benign appearances but slower encode/decode.
-

- Database Encryption:
    - Not included by default; left to user discretion if chunks themselves need confidentiality beyond semantic annihilation.
-

8.5 Implementation Robustness

The reference implementation includes:

- Batch chunk storage to reduce SQLite transaction overhead (BATCH_FLUSH_THRESHOLD = 1000).
- Force-internal output protection to prevent annihilating its own output directory.
- Verbose logging mode for per-chunk inspection during development and debugging.
- Error handling for malformed keymaps and partial database corruption.

9. Limitations and Future Work

9.1 Current Limitations

Partial Data Pattern Leakage

While header wiping removes the most critical semantic identifiers, some formats contain redundant structure throughout the file. For example, certain media files store sync markers or repeating signatures in non-header regions. In these cases, deep forensic analysis on chunks may still yield probabilistic format identification, especially for very large files with distinctive redundancy patterns.

Keymap Exposure Risk

The keymap is a single point of failure for reversibility. If both the chunk database and the keymap are compromised, the original file can be reconstructed exactly. Keymap protection (via encryption, physical separation, or access controls) is therefore mandatory in sensitive deployments.

Database Carving Possibility

Although the chunk database contains no headers or file structure, an adversary with specific target data and unlimited resources could attempt a known-chunk attack by comparing suspected content to stored chunks. This is mitigated by file-specific salting, but identical chunks from the same file will still be detectable if salt is known.

No Intrinsic Confidentiality

Semantic annihilation does not hide the chunk content itself — it removes its meaning. For scenarios requiring both confidentiality and meaninglessness, annihilation should be composed with encryption.

Performance Bound by I/O

The Python reference implementation achieves acceptable throughput for general use but is limited by SQLite write speeds and Python's per-chunk processing overhead. For multi-terabyte datasets, hardware acceleration or parallelized implementations will be necessary.

9.2 Future Work

Hardware-Accelerated Annihilation

Implementing USF in C, Rust, or FPGA/ASIC could enable annihilation speeds matching high-throughput storage systems.

Parallelized Chunk Processing

Splitting large files into independent segment annihilation tasks would allow multi-core scaling without altering output format.

Adaptive Wipe Patterns

Exploring statistical analysis of file segments to selectively wipe areas most likely to carry semantic markers, potentially reducing keymap size without reducing irreversibility.

Database Encryption and Sharding

Encrypting the chunk database or distributing it across multiple storage providers can further reduce exposure risk, even if keymap and partial chunks are compromised.

Stealthier Disguise Formats

Expanding the disguise format library (e.g., embedding in image EXIF data, office document metadata) would increase plausible deniability in environments with strict file inspection.

Integration with Distributed Storage

Native drivers for IPFS, S3, and other blobstores could allow transparent annihilation and retrieval in decentralized or cloud-hosted networks.

Formal Security Proofs

While current proofs are based on cryptographic assumptions and combinatorial infeasibility, full formal verification under accepted frameworks (e.g., IND-CPA analogs for semantics) would strengthen adoption in academic and regulatory contexts.

10. Conclusion

The persistence of meaning in digital artifacts — even after encryption, compression, or hashing — has long been an overlooked security exposure. File headers, structural markers, and predictable redundancy provide adversaries with avenues for identification, classification, and coercion, regardless of whether they can decrypt the data.

Veriduct introduces semantic annihilation as a fourth fundamental primitive in the data lifecycle, operating orthogonally to existing techniques. By splitting data into meaning-free chunks and a separate reconstruction keymap, Veriduct ensures that stored or transmitted material is devoid of recognizable format unless both components are reunited. This property enables:

- Operational security in hostile or coercive environments.
- Forensic scanning resistance against bulk indexing.
- Strategic control over when, how, and whether meaning is ever restored.

The mathematical model formalizes the annihilation and reassembly process, proving irreversibility without the keymap under standard cryptographic assumptions. The open-source reference implementation demonstrates that these guarantees can be achieved with modest computational overhead and are compatible with large-scale storage systems.

Veriduct is not a replacement for encryption, compression, or hashing — it is an expansion of the security toolkit into a dimension that was previously left unaddressed: the deliberate removal of meaning as a controllable, reversible operation.

We invite researchers, developers, and system architects to:

1. Audit and challenge the model and implementation.
2. Integrate semantic annihilation into workflows where format recognition itself is a risk.
3. Explore compositions of annihilation with existing primitives to strengthen multi-layer defenses.

By making the absence of meaning a deliberate and measurable property, Veriduct reframes the conversation on data control. In a world where every bit can be scanned, indexed, and classified, sometimes the most powerful statement is to store nothing at all — until you decide otherwise.

Appendix A: Hardened Variants and Extensions

This appendix outlines advanced techniques and optional extensions to the core Veriduct process. These are not part of the minimal reference implementation but are intended for scenarios requiring higher resilience against specialized forensic or intelligence capabilities.

A.1 Multi-Layer Salting

Instead of a single file-level salt s, introduce:

1. Global Salt — applied to all files in a deployment (kept offline or embedded in hardware).
2. File Salt — unique per file, as in the core model.
3. Chunk Salt — unique per chunk, generated deterministically from the file salt + chunk index using a secure PRF (e.g., HMAC-SHA256).

This ensures that even identical chunks within the same file or across files produce different hashes, mitigating known-chunk and deduplication-based analysis.

A.2 Non-Uniform Chunking

Randomly vary chunk sizes within a defined range (e.g., 2 KB–8 KB) to break alignment assumptions and frustrate attempts at block-boundary inference. Chunk size variation is encoded in the keymap.

A.3 Dummy Chunks and Decoy Keymaps

Insert randomly generated dummy chunks into the chunk database and record them in decoy keymaps. This creates plausible but incorrect reconstructions if a decoy keymap is intercepted.

A.4 Keymap Fragmentation

Split a single keymap into multiple fragments, each containing only a portion of the reconstruction sequence. The complete file can only be reassembled by combining all fragments.

A.5 Semantic Shatter Mapping (SSM)

—

Hardened Implementation

Definition: SSM is a hardened extension of semantic annihilation in which the reconstruction keymap is split into multiple interdependent shards.

Key characteristics from the hardened implementation:

- Shards with cross-references — Each shard contains chunk hashes and pointers to chunks located in other shards. These pointers must be resolved recursively to build the complete reconstruction sequence.
- Randomized shard structure — The number of shards and their sizes vary per file, making it infeasible for an adversary to know how many pieces are required.
- Bounds checking — The reassembly process verifies that all cross-references are valid, preventing tampering or shard corruption from producing partial output.
- Independent disguise formats — Each shard can be written in a different benign-looking format (CSV, LOG, CONF, etc.), allowing shards to be distributed or stored in unrelated contexts.
- Reassembly dependency — Possessing even most of the shards is insufficient; the full set is required to resolve all cross-references and reconstruct the file.

Security implication: SSM raises the difficulty of reconstruction from possession of the keymap to possession of all correctly-linked shards, making partial keymap compromise effectively useless.

A.6 Header Overwrite Beyond Fixed Wipe Size

Instead of wiping only the first w bytes, employ a pattern-based wipe that overwrites multiple strategically selected regions in the file, identified via format-agnostic statistical analysis to target semantic hotspots.

A.7 Chunk Re-encoding

Re-encode each chunk through a reversible but opaque transform (e.g., base85, reversible permutation of bytes) before storage. While not adding cryptographic strength, it breaks direct binary pattern matching in chunk databases.

A.8 Combined Hardened Mode

In practice, the above hardening techniques can be combined.

For example:

- Non-uniform chunking + multi-layer salting + SSM = high forensic resistance with multi-party reconstruction control.