Digstore

The Content-Addressable WASM Store Format

Michael Taylor

Version 1.0 · May 2026

Abstract

Digstore is a self-contained, content-addressable, encrypted-at-rest store format. A Digstore store is a WebAssembly module. Store content compiles into a single `.wasm` binary whose data section embeds the chunked, encrypted content, the merkle commitments, the root history, the store's public key, and a set of trusted-host keys. The module exposes a fixed export ABI. A host runtime instantiates it in a sandbox and retrieves content by executing it. The artifact is executable, not passive. The content and the logic that serves it ship as one binary.

A store module gates its own access. Before releasing content it runs an attestation handshake against the host, establishes a session, and (where configured) verifies JWT authorization. Every response can carry a zero-knowledge proof of execution: a succinct proof that the output is the correct result of running the module on the request, verified against the module's hash. The module embeds no secret of any kind. There is nothing in it to extract or forge.

Files are addressed by URNs of shape `urn:dig:<chain>:<storeID>[:<rootHash>][/<resourceKey>]`. Content is chunked with content-defined chunking, hashed with SHA-256, committed in a per-generation merkle tree, and encrypted at rest with a key derived from the URN itself. Invalid URNs do not error. The module returns deterministic decoy bytes drawn from a logarithmic size distribution, so a host cannot distinguish a real lookup from a miss.

A store is identified by a 32-byte store ID. Every commit produces a new module file named `{storeID}-{rootHash}.wasm`, computes a new merkle root, and appends it to the root history. The module is the single distribution unit: one portable executable that is both the data and the server.

The developer-facing workflow is Git. `init`, `add`, `commit`, `log`, `diff`, `checkout`, and `clone` behave as a developer expects. A commit is a generation and its root hash is the commit identifier. The chunking, the URN-derived encryption, the merkle commitments, and the compile-to-WebAssembly step all run beneath that familiar surface, so the cryptography is a property of the format rather than a task in the workflow.

The privacy consequence is the heart of the format. The two values that turn a URN into content are both derived from the URN and nothing else: the retrieval key that locates a resource is a hash of the URN, and the decryption key is derived from the URN by HKDF-SHA256. A provider serving a digstore module (a DIG Node acting as a neutral pipe) holds an opaque executable and ciphertext. It never sees the URN. Without the URN it cannot compute the retrieval key for any resource it does not already know, cannot derive a decryption key, and cannot read what it serves. Decryption runs on the client after it receives the encrypted chunk, never on the provider. Arbitrary stores cannot be scanned at rest or inspected in flight: a provider has no way to learn what it carries without advance knowledge of the exact URN.

This paper covers the store entity, the WASM module format and its compilation pipeline, the host/module ABI, the URN system, the content model, the encryption and zero-knowledge schemes, the attestation and execution-proof systems, the host runtime, and the command-line interface. It documents the local store format only. Network distribution, external identity anchoring, and payment settlement are out of scope.

Table of Contents

Part 1: Foundation

1. The Big Ideas

1. Every URN is a key. Retrieval key, encryption key, access token: all derived from the URN and nothing else. The retrieval key that locates a resource is a hash of the URN. The decryption key is derived from the URN by HKDF-SHA256. Anyone holding the URN can locate and decrypt. Anyone lacking it can do neither. There is no separate access-control list. The URN is the credential.

2. The store is an executable. A digstore store compiles to a WebAssembly module. The content lives in the module's data section. The logic to authenticate a host, gate access, and serve content lives in the code section. The host does not parse the module. The host runs it inside a sandbox through a fixed export ABI. A store that wants to enforce authorization carries that enforcement inside itself.

3. Content is provably genuine; execution is provably correct. A response is bound to the store's trusted root by a merkle inclusion proof and to the URN by GCM-authenticated encryption, so no host can make a client accept fabricated, substituted, or truncated content (§9.4, §11). The module has no secret to help it try. On top of that, a zero-knowledge proof of execution attests that the output is the correct result of running the genuine module on the request, verified against the module's hash (§13). The proof carries no secret. There is nothing to extract and nothing to forge: a valid proof exists only for a genuine run. The module is secretless by design.

4. Invalid URNs are indistinguishable from valid URNs. A host serving requests cannot tell which URNs map to real content. Invalid URNs return deterministic decoy bytes whose size is drawn from a logarithmic distribution seeded by the URN, with a real-looking proof blob attached. Valid URNs return ciphertext plus a real proof. Both look alike on the wire. Enumeration degrades to guessing.

5. The provider is blind. Retrieval and decryption keys are both functions of the URN, so a provider that serves a module learns nothing from doing so. It receives a retrieval key (a hash) and returns ciphertext. It never holds the URN, so it cannot reverse the hash, cannot derive the decryption key, and cannot inspect the content it relays. A DIG Node is a neutral pipe by construction, not by policy.

The module is the source of truth for content. A 32-byte store ID names the store. The current root hash names its state.

2. Positioning

A conventional content store is a passive container. A host parses it, serves raw bytes, and all access logic lives in the client. Digstore puts that logic into the artifact. The store is an executable that defends itself. It decides whom to serve, decrypts only what a valid request names, and signs what it returns.

Capability	Passive content store	Digstore WASM module
Artifact	File parsed by the host	Executable module run by the host
Served by host as	Raw bytes	Result of a sandboxed execution
Access control	Client-side, by possession	In-module: attestation + session + optional JWT
Response integrity	Merkle proof against root	Merkle proof plus zero-knowledge proof of execution
Content addressing	Varies	SHA-256 throughout; URN-addressed
Encryption at rest	Out-of-band or none	URN-derived AES-256-GCM, embedded in module
Invalid lookups	Error / 404	Deterministic decoy, logarithmic size distribution

Byte-range retrieval	Application-specific	Resource-key + range, served by module
Distribution unit	One container file	One `{storeID}-{rootHash}.wasm` file

What the digstore module delivers:

- **Self-enforcing access.** Authorization checks run inside the module. A host cannot serve content the module would have refused, because the host does not hold the decryption logic. It holds an opaque executable.
- **Provenance per response.** A zero-knowledge proof of execution attests that the genuine module produced the response. A relay cannot substitute, truncate, or fabricate content without invalidating it, and there is no secret it could extract to forge one.
- **Host attestation.** A module refuses to operate until the host proves its identity against trusted keys baked in at compile time.
- **Provider blindness.** Retrieval and decryption keys derive from the URN. A provider relays ciphertext addressed by a hash and never sees the URN, so it cannot scan a store at rest or inspect a request in flight. Neutral-pipe status is a cryptographic guarantee, not an operational promise.
- **A Git-shaped interface.** The developer workflow is `init`, `add`, `commit`, `log`, `diff`, `checkout`, `clone`: the verbs and the mental model Git already taught. Encryption, chunking, and WASM compilation stay under that surface (§20).
- **One portable executable.** The store is a single `.wasm` file. Copying it is a backup. Running it is a server.

What the WASM store deliberately does not do:

- **No working tree.** Content is read out by URN through module execution, not checked out.
- **No branching.** Linear root history. Each commit produces one new module and one new root hash.
- **No partial module.** The whole store is one module. Either you have it or you don't.
- **No host-side decryption.** The host runs the module. It does not parse content out of it.

3. Design Heritage

Digstore borrows, with attribution.

Component	Heritage	Digstore adaptation
Content addressing	Git, IPFS	SHA-256 throughout; URN scheme distinct from CID
Executable data store	WebAssembly, eBPF, capability machines	Content + access logic compiled to a sandboxed WASM module
Sandboxed execution	wasmtime, wasmer	Host runtime with timeout and memory bounds; fixed import/export ABI
Content-defined chunking	rsync, restic, borg, FastCDC	Gear-based rolling hash; target 64 KiB, min 16 KiB, max 256 KiB

Merkle commitment	Git, Ethereum, Plasma	Per-generation tree over chunk leaves; root is the generation root hash
Stream-cipher-from-hash	NaCl, age, libsodium	HKDF-SHA256 from URN to AES-256-GCM key; fixed nonce, unique key per URN
URN format	RFC 8141	`urn:dig:<chain>:<storeID>[:<rootHash>][/<r`
Code obfuscation	Software-protection literature	Deterministic instruction substitution, opaque predicates, bogus code, control-flow nops
Host attestation	TEE remote attestation	BLS host attestation against compile-time trusted keys
Execution proofs	zkVM / proof-carrying code (zk-STARK, SNARK)	Zero-knowledge proof of correct WASM execution; no embedded secret

What Digstore does not inherit:

- **Git's object graph.** No trees, blobs, or commits-as-objects. A generation is a flat chunk set with a merkle tree.
- **IPFS's DHT.** Location is determined out-of-band, not by a content router.

Part 2: The WASM Store Format

4. Store Structure

A store is a unified entity tying together a 32-byte store ID, a WASM module, and local configuration. In code this is the Store entity (`dig-store`), wrapping the compiled module and a `StoreConfig`.

4.1 Store Configuration

Local configuration is small and portable (`dig-store-config`):

```
pub struct StoreConfig {
   pub store_id: Bytes32,      // 32-byte store identifier
   pub data_dir: String,       // local working directory
   pub max_size: u64,          // maximum store size in bytes
   pub visibility: Visibility, // Public (URN decrypts) | Private (URN + salt)
}

pub enum Visibility {
   Public,             // default: anyone holding the URN can decrypt
   Private(SecretSalt) // URN alone is not enough; salt held by publisher
}
```

No absolute paths leak into addressing. The `visibility` field is the publisher's access-model choice, made once at `init` and fixed for the store's life. A public store is readable by anyone holding the URN. A private store stays sealed even to a URN-holder without the salt (§11.4). Per-store policy (such as JWT authentication) is likewise carried as configuration and compiled into the module, not enforced by the host.

4.2 Store ID

32 bytes, hex-encoded (64 characters). The store ID is a stable identifier for the store. It may be a random value or derived from the publisher's key. It is curried into every URN that references the store and is the only mandatory addressing component. Anchoring the store ID to an external identity is out of scope for this document.

4.3 Generations and Root Hash

Term	Meaning
Generation	A store state identified by a root hash, with an ordinal `GenerationId` (u64)
Root hash	Merkle root over the generation's chunk set (`GenerationState.root`)
Root history	Append-only list of every root hash the store has produced

```
pub struct GenerationState {
   pub id: GenerationId, // monotonic u64
   pub root: Bytes32,    // merkle root of this generation
   pub timestamp: u64,   // unix seconds
}

pub struct Generation {
   pub state: GenerationState,
   pub tree: MerkleTree, // tree over the generation's chunks
}
```

Each commit produces exactly one new generation, one new module file, and one new root hash. The root history grows monotonically and is exported by the module via `get_roothash_history`. Any past root hash can be quoted in a URN to address the store at that generation.

4.4 On-Disk Layout

```
~/.dig/
 {store_id}.staging.bin              // binary staging area (build time)
 generations/
   {roothash_hex}/
     manifest.json                   // generation metadata
     chunks/{chunk_hash_hex}         // one file per unique chunk
 modules/
   {store_id}-{roothash}.wasm        // compiled module, one per generation
 config.toml                         // global configuration
```

The compiled module is the distribution unit. Backups copy the module. Verification opens the module. Serving instantiates the module.

5. The Store Module

The store module is a WebAssembly binary produced by the compiler (`dig-compiler`). It is the single file a host serves. Compiler version 1.0.0; module format version 1.

5.1 Module Sections

The compiler emits the standard WASM sections in order:

Section	Contents
Type	Function signatures (`[] -> [i64]`, `[i32,i32] -> [i64]`, etc.)
Import	Host functions in the `dig_host` module (§6.2)
Function	Type indices for each defined function
Memory	One linear memory: min 1 page (64 KiB), max 256 pages (16 MiB)
Export	The store ABI (§6.1) plus memory, alloc, dealloc, init
Code	Function bodies, obfuscated if enabled (§17)
Data	Embedded content: chunks, key table, metadata, trusted keys (no secret)

```
MemoryType {
   minimum: 1,         // 64 KiB
   maximum: Some(256), // 16 MiB module-declared cap
   memory64: false,
   shared: false,
}
```

The module-declared memory cap (16 MiB) is the inner bound. The host runtime additionally enforces an outer memory limit and a wall-clock timeout (§18).

5.2 What Gets Embedded

The compiler loads every generation from disk, deduplicates chunks across generations into a single chunk index, builds a resource-key table, and embeds the result in the data section.

```
pub struct CompilationContext {
   pub config: CompilerConfig,
   pub store_config: Option<StoreConfig>,
   pub generations: Vec<LoadedGeneration>,
   pub chunk_index: ChunkIndex, // deduplicated chunks, hash -> global index
   pub key_table: KeyTable,     // resource key -> ordered chunk indices
   pub wasm_module: Option<Vec<u8>>,
   pub stats: CompilationStats,
}

pub struct KeyTableEntry {
   pub static_key: Bytes32,       // resource key (32 bytes)
   pub generation: Bytes32,       // generation root hash
   pub chunk_indices: Vec<u32>,   // ordered chunk indices that reassemble the resource
   pub total_size: u64,           // reassembled content size
}
```

Embedded in the data section, for each generation:

- **Chunks (the interleaved pool).** The deduplicated, encrypted chunk bodies, each addressed by its SHA-256 hash, placed into one interleaved pool with random filler in the gaps and no resource boundaries (§8.3). A chunk shared across generations is stored once.
- **Key table.** Maps each resource key to the ordered chunk indices and total size that reassemble it.
- **Metadata.** Store ID, root history, store public key, authentication info.
- **Metadata manifest.** A plaintext, publisher-authored manifest of descriptive store and authorship information (§8.4). Unlike content, the manifest is not encrypted, so any holder of the module can read it without a URN.
- **Trusted host keys.** The set of BLS host public keys the module will attest against (§12).

The module embeds no secret: no signing key and no decryption key. Content keys live with URN-holders (§11). Execution proofs use no secret (§13). A dump of the data section is ciphertext, public metadata, and public keys.

5.3 Compilation Pipeline

The compiler runs a fixed sequence of stages (`dig-compiler`):

1. **Config load.** Parse `store.json`, extract `store_id`, validate required fields.
2. **Generation load.** Read each `generations/{roothash}/` directory: `manifest.json`, chunk files, build the per-generation merkle tree.
3. **Chunk process.** Deduplicate chunks across all generations into the global chunk index.
4. **Key-table build.** Map resource keys to chunk-index sequences per generation.
5. **Code build.** Emit the type, import, function, memory, and export sections.
6. **Data embed.** Encode chunks, metadata, key table, and trusted keys into the data section (binary codec, little-endian).
7. **Obfuscation.** Optionally apply code-obfuscation passes (§17).
8. **Optimization.** Optional `wasm-opt`.
9. **Validation.** Re-parse and validate the emitted module.
10. **Output.** Write atomically to `{hex(store_id)}-{hex(roothash)}.wasm` via temp file and rename.

```
pub struct CompilationResult {
   pub store_id: Bytes32,
   pub roothash: Bytes32,
   pub output_path: PathBuf, // {store_id}-{roothash}.wasm
   pub output_size: u64,
   pub stats: CompilationStats,
}
```

Compilation requires at least one trusted host key. The compiler refuses to emit a module with an empty trusted-key set (`CompilerError::NoTrustedKeys`).

6. The Host/Module ABI

The host and module communicate across a fixed application binary interface. All data crosses the boundary through linear memory. Integer return values pack a pointer and a length.

6.1 Return Encoding

A function that returns data returns an i64 packing a 32-bit pointer in the high half and a 32-bit length in the low half:

```
pub const fn pack_ptr_len(ptr: u32, len: u32) -> i64 {
   ((ptr as i64) << 32) | (len as i64)
}
pub const fn unpack_ptr_len(packed: i64) -> (u32, u32) {
   ((packed >> 32) as u32, (packed & 0xFFFF_FFFF) as u32)
}
// Error sentinel: len == 0 and (ptr as i32) < 0 => ptr is an error code.
pub const fn is_error(packed: i64) -> bool {
   let (ptr, len) = unpack_ptr_len(packed);
   len == 0 && (ptr as i32) < 0
}
```

6.2 Module Exports

Every store module exposes:

Export	Signature	Returns
`get_store_id`	`() -> i64`	Store ID (32 bytes)
`get_current_roothash`	`() -> i64`	Current generation root hash
`get_roothash_history`	`() -> i64`	Array of all root hashes
`get_public_key`	`() -> i64`	Store public key (48 bytes, BLS G1)
`get_metadata`	`() -> i64`	Metadata manifest, plaintext (§8.4)
`get_authentication_info`	`() -> i64`	AuthenticationInfo
`get_content`	`(req_ptr: i32, req_len: i32) -> i64`	ContentResponse or decoy
`get_proof`	`(req_ptr: i32, req_len: i32) -> i64`	ProofResponse
`alloc`	`(size: i32) -> i32`	Pointer into linear memory
`dealloc`	`(ptr: i32, size: i32) -> ()`	(none)
`init`	`() -> i32`	0 = success, < 0 = error
`memory`	(exported memory)	Linear memory

`get_content` and `get_proof` take a request encoded into module memory by the host. The host calls `alloc`, writes the request, then calls the export with the pointer and length. `get_metadata` takes no request and returns the plaintext manifest directly; it is not gated by the URN, the session, or attestation (§8.4).

6.3 Host Imports (dig_host)

The module imports host services. Each import returns an i32: a non-negative value is the length written to the shared return buffer; a negative value is an error code. The module reads results back with `host_read_return_buffer`.

Import	Signature	Purpose
`host_get_public_key`	`() -> i32`	Host's BLS public key
`host_create_attestation`	`(challenge_ptr) -> i32`	Host signs an attestation challenge
`host_establish_session`	`(challenge_ptr) -> i32`	Create a session after attestation
`host_verify_session`	`() -> i32`	1 = valid, 0 = invalid
`jwks_fetch`	`(url_ptr, url_len) -> i32`	Fetch a JWKS document for JWT validation
`host_get_current_time`	`() -> i64`	Unix timestamp (seconds)
`host_random_bytes`	`(count) -> i32`	Cryptographic random bytes
`host_read_return_buffer`	`(dest_ptr) -> i32`	Copy the host return buffer into module memory

`jwks_fetch` is session-gated. The host returns `NoSession` (-100) or `SessionExpired` (-101) until the module has established a valid session. Identity and attestation imports do not require a session: they bootstrap it. This is what lets a module enforce JWT authorization. It fetches the relevant JWKS and validates a presented token before deciding whether `get_content` returns real bytes or a decoy.

6.4 The Return Buffer

The host owns a shared return buffer. Import results are written into it and the module reads out of it:

```
DEFAULT_CAPACITY = 64 KiB
MAX_RETURN_BUFFER_SIZE = 16 MiB
```

Host imports are configured by `HostImportsConfig`:

```
pub struct HostImportsConfig {
   pub return_buffer_capacity: usize, // default 64 KiB
   pub max_return_buffer_size: usize, // default 16 MiB
   pub max_random_bytes: u32,         // default 1024
   pub host_version: String,
}
```

6.5 Error Codes

```
pub enum ErrorCode {
   GeneralError      = -1,
   InvalidParameter  = -2,
   BufferTooSmall    = -3,
   NoSession         = -100,
   SessionExpired    = -101,
   AttestationFailed = -102,
   NetworkError      = -200,
   Timeout           = -203,
   NotFound          = -300,
   ValidationFailed  = -301,
}
```

7. URN System

Every resource is addressed by a Uniform Resource Name. The URN is the sole input to both locating and decrypting a resource (§1, §11). An invalid URN is indistinguishable from a valid one at retrieval time (§14).

7.1 Format

```
urn:dig:<chain>:<storeID>[:<rootHash>][/<resourceKey>]
```

```
pub struct Urn {
   pub chain: String,                 // chain identifier, e.g. "chia"
   pub store_id: Bytes32,             // 32-byte store id (required)
   pub root_hash: Option<Bytes32>,    // optional generation pin
   pub resource_key: Option<String>,  // optional resource path
}
```

7.2 Components

Component	Role	Required
`urn`	Scheme	Required. The literal `urn`.
`dig`	Namespace	Required. The literal `dig`.
`<chain>`	Chain	Required. Chain identifier (e.g. `chia`).
`<storeID>`	Store ID	Required. 64 hex characters.
`<rootHash>`	Generation	Optional. Pins a generation; absent means the current root.
`<resourceKey>`	Resource	Optional. Path-like key; absent means store-level.

7.3 Resolution

The URN is canonicalized, then two values are derived from the canonical string and nothing else:

- **Retrieval key.** `retrieval_key = SHA-256(canonical_urn)`. The 32-byte key under which the resource is located. This is the only address that ever leaves the client; the URN itself does not.
- **Decryption key.** `decryption_key = HKDF-SHA256(canonical_urn)`. The AES-256-GCM key used to decrypt the resource's chunks (§11).

If `rootHash` is omitted, resolution targets the current generation; if present, that exact generation. A resolved `resourceKey` maps through the module's key table to the ordered chunk indices that reassemble the resource. Resolution that finds no entry does not fail: it returns a deterministic decoy (§14).

8. Content Model

8.1 Chunking

Content is split into variable-size chunks by content-defined chunking. A Gear-based rolling hash slides over the byte stream; a boundary is cut where the masked rolling hash matches, bounded by a minimum and a maximum size. Identical content produces identical boundaries regardless of surrounding edits, which maximizes deduplication across generations.

```
pub struct ChunkerConfig {
   pub min_size: usize,    // 16 KiB
   pub target_size: usize, // 64 KiB
   pub max_size: usize,    // 256 KiB
   pub mask: u64,          // boundary mask for the target size
}
```

Parameter	Value	Effect
Minimum chunk	16 KiB	Floor on chunk size; suppresses pathological tiny chunks
Target chunk	64 KiB	Expected average; sets the boundary mask width
Maximum chunk	256 KiB	Hard cap; forces a boundary if none is found

Each chunk is hashed with SHA-256. The hash is the chunk's content address and its key in the deduplicated chunk index.

8.2 Generations

A generation is the complete chunk set for one store state. A commit deduplicates the new state's chunks against the existing index, builds the generation's merkle tree over its chunk leaves, and records the root. Chunks shared with prior generations are not re-stored. The generation's root hash is its identifier and is appended to the root history.

8.3 The Interleaved Chunk Pool

Inside the module, chunks are not grouped by resource. Every chunk across every resource is placed into a single interleaved pool, the gaps padded with random filler, and no resource boundary is observable in the data section. A resource is reassembled by a path walk: the key table yields the ordered chunk indices, and the module gathers exactly those chunks from the pool in order.

```
pub struct PathWalk {
   pub resource_key: Bytes32,   // requested resource
   pub chunk_indices: Vec<u32>, // ordered indices into the pool
   pub cursor: usize,           // current position in the walk
}
```

Because the pool carries filler and no boundaries, the data section reveals neither how many resources a store holds nor where one ends and the next begins. Reassembly requires the key table entry, which requires the resource key, which requires the URN.

8.4 The Metadata Manifest

A store carries a publisher-authored **metadata manifest**: a structured, plaintext description of the store, its authorship, and related information, in the spirit of a package manifest. The manifest is embedded in the data section and is read through the `get_metadata` export (§6.2). It is the one part of a store that is deliberately readable without a URN.

Plaintext by design. Unlike content, the manifest is not encrypted (§11). It is the store's public face: a host, an indexer, or a browser can read it directly from the module, with no URN, no session, and no attestation. This is a deliberate carve-out from the encryption model. Content remains URN-gated and provider-blind; the manifest is intentionally open so a store can describe itself. A publisher who wants no public description ships an empty manifest.

Schema. The manifest is a set of standard fields plus an open-ended map. Only `name` is required.

Field	Type	Meaning
`schema_version`	integer	Manifest schema version
`name`	string (required)	Human-readable store name
`version`	string	Publisher's content version, e.g. semantic version
`description`	string	Short description of the store
`authors`	array	Author entries: name and optional handle or contact
`license`	string	License identifier
`homepage`	string	Canonical homepage URL
`repository`	string	Source repository URL
`keywords`	array	Free-text tags for discovery
`categories`	array	Classification categories
`icon`	string	Resource key or URL of an icon
`content_type`	string	Primary content type the store serves
`links`	map	Named URLs, e.g. docs, support

`custom`	map	Open-ended publisher-defined key/value pairs

```
pub struct MetadataManifest {
   pub schema_version: u32,
   pub name: String,                          // required
   pub version: Option<String>,
   pub description: Option<String>,
   pub authors: Vec<Author>,
   pub license: Option<String>,
   pub homepage: Option<String>,
   pub repository: Option<String>,
   pub keywords: Vec<String>,
   pub categories: Vec<String>,
   pub icon: Option<String>,
   pub content_type: Option<String>,
   pub links: BTreeMap<String, String>,
   pub custom: BTreeMap<String, Value>,    // open-ended
}
```

Editing and re-compilation. The manifest is part of the module bytes, so changing it changes the module and therefore the program hash. A metadata edit is a re-compilation, not a new generation: the content root is unchanged because no content chunk changed, while the program hash advances because the bytes did. The distinction matters wherever the two values are tracked separately: a manifest edit moves the program hash and leaves the content root and the root history untouched.

8.5 Social Conventions

Content addressing is private by default: without the exact URN, a resource cannot be located or read. That same property makes a store opaque to anyone who wants to discover what it offers. **Social conventions** are an optional, opt-in layer that makes selected resources discoverable without weakening the guarantee for everything else.

How discovery is possible without the URN. A resource key is the only secret part of a URN; the store ID and chain are public, and the URN derivation is public. So a party that already knows a store's ID can construct the URN for any *agreed, well-known* resource key, derive its retrieval key, fetch it, and, on a public store, derive its decryption key and read it. Discovery therefore works for conventionally named resources precisely because their resource keys are public by agreement, while secret-keyed resources stay opaque because their keys are not.

Convention	Resource key	Purpose
Default resource	`index.html`	The store's landing resource, rendered as its default view
Discovery manifest	`/.well-known/dig/manifest.json`	A machine-readable list of the resources the publisher chooses to expose, with labels and types

Public versus private stores. On a public store, a conventional resource is both locatable and readable by anyone who knows the store ID, which is the intended discovery behavior. On a private store, the conventional resource key still yields a retrieval key, but decryption additionally requires the secret salt (§11.4), so the resource is locatable in principle yet sealed in practice. A publisher who wants a discoverable store ships it public; a publisher who wants an opaque store ships it private, and the conventions reveal nothing.

Opt-in and partial. Conventions are conventions, not protocol requirements. A store may expose all, some, or none of them. A consumer that reads conventions lists only what the publisher placed at conventional keys; it cannot enumerate secret-keyed content, because nothing in the module maps a public name to it. The convention layer adds discoverability for what a publisher elects to surface and changes nothing about the privacy of what they do not.

9. Merkle Proofs

Every response can be bound to a generation's root by a merkle inclusion proof, so a client can verify that the bytes it received belong to the store state it trusts.

9.1 Tree Construction

A generation's chunks are the leaves of a binary merkle tree. A leaf is `SHA-256(chunk)`. An interior node is `SHA-256(left || right)`. When a level has an odd number of nodes, the last node is carried up. The root is the generation's root hash.

9.2 Inclusion Proof

```
pub struct MerkleProof {
   pub leaf: Bytes32,          // SHA-256 of the chunk
   pub path: Vec<ProofStep>,   // sibling hashes from leaf to root
   pub root: Bytes32,          // claimed generation root
}
pub struct ProofStep {
   pub hash: Bytes32,          // sibling hash at this level
   pub is_left: bool,          // sibling's side
}
```

9.3 Verification

The verifier starts from the leaf and folds each step: combine the running hash with the sibling on the indicated side, hash, and ascend. The proof is accepted if and only if the recomputed root equals the trusted root.

9.4 Binding to the Trusted Root

The verifier supplies the root from the store's root history, or from an external anchor it trusts, never from the response. This is what makes content provably genuine: any altered, substituted, or truncated chunk changes its leaf, breaks the path, and fails to recompute the trusted root. A host cannot fabricate content that verifies, and the module holds no secret that would help it try. Integrity rests on the hash function and the trusted root alone.

9.5 Proof Size

A proof carries `ceil(log2(n))` sibling hashes for a generation of `n` chunks, 32 bytes each. Verification is logarithmic in the number of chunks and requires only the leaf, the path, and the trusted root.

Part 3: Security and Zero-Knowledge

10. Threat Model

10.1 Adversaries

- **A curious or malicious provider.** A DIG Node that stores and serves the module and wishes to learn what it carries or to alter what it returns.
- **A network observer.** A party between client and provider watching requests and responses.
- **An enumerating client.** A party without the URN attempting to discover which addresses map to real content.

10.2 Guarantees

- **Confidentiality.** Content is unreadable without the URN, because the decryption key is derived from the URN (§11).
- **Integrity.** Responses are bound to the trusted root by a merkle proof (§9) and to the URN by GCM authentication (§11), and to the genuine module by an execution proof (§13).
- **Blindness.** The provider receives a retrieval key, a hash, and returns ciphertext. It cannot scan a store at rest or inspect a request in flight (§15).
- **Indistinguishability.** Invalid URNs return deterministic decoys that look like real responses (§14).

10.3 Assumptions and Non-Goals

- The primitives are secure: SHA-256, AES-256-GCM, HKDF-SHA256, BLS signatures, and the zero-knowledge proof system.
- The URN is a capability. Anyone who learns it gains access. How URNs are distributed and to whom is out of scope.
- Resistance to traffic analysis beyond decoys (§14) and oblivious access (§14) is out of scope for this document.

11. URN-Based Encryption

11.1 Key Derivation

The decryption key for a resource is derived from its canonical URN by HKDF-SHA256, yielding a 32-byte AES-256 key:

```
decryption_key = HKDF-SHA256(ikm = canonical_urn, salt, info) -> 32 bytes
```

The key is a deterministic function of the URN. No key material is stored in the module, transmitted, or held by the provider. Possession of the URN is necessary and sufficient to derive the key for a public store.

11.2 Encryption

Each resource's chunks are encrypted with AES-256-GCM under the resource's URN-derived key. A fixed nonce is used. This is safe because the key is unique per URN: GCM's nonce-reuse hazard arises only when a single key is reused across different plaintexts under the same nonce, and distinct URNs never share a key. The GCM authentication tag binds the ciphertext to the key, so tampering is detected on decryption and surfaced as a failure rather than silently accepted.

11.3 Decryption

Decryption runs on the client, after it receives the ciphertext chunk. The client derives the key from the URN it holds, verifies the GCM tag, and recovers the plaintext. The provider performs no decryption and never holds the key. A failed tag is reported as tampering.

11.4 Private Stores

A private store mixes a secret salt into the key derivation, so the URN alone is insufficient to derive the decryption key. The salt is held by the publisher and shared out-of-band with authorized readers. A URN-holder who lacks the salt can locate a resource (the retrieval key still derives from the URN) but cannot read it. This is the mechanism behind the public-versus-private distinction in the visibility configuration (§4.1) and in social-convention discovery (§8.5).

12. Host Attestation and Sessions

A store module refuses to serve content until the host proves its identity. The module embeds a set of trusted BLS keys at compile time (§5.2), and attestation binds the running host to one of those keys. A host that cannot attest receives decoys, never content.

12.1 The Handshake

The module issues a challenge; the host signs it and returns a response:

```
pub struct AttestationChallenge {
   pub nonce: [u8; 32],     // module-generated random nonce
   pub store_id: [u8; 32],  // the store being served
   pub timestamp: u64,      // unix seconds, for freshness
}

pub struct AttestationResponse {
   pub host_public_key: [u8; 48],  // host BLS public key (G1)
   pub host_instance_id: [u8; 32], // host instance identifier
   pub signature: [u8; 96],        // BLS signature (G2) over the challenge
}
```

The module calls `host_create_attestation` with the challenge. The host signs the challenge with its BLS key and returns the response through the return buffer.

12.2 Verification

The module verifies the BLS signature over the challenge under `host_public_key`, checks the timestamp for freshness, and checks that `host_public_key` is a member of the trusted set embedded at compile time. If any check fails the module refuses (`AttestationFailed`, -102) and subsequent content calls return decoys. There is no fallback path that serves content without attestation.

12.3 Trusted Key Format

Trusted host keys are recorded as versioned entries of the form `dig-host-key-v1:<hex>`, where the hex encodes a BLS public key. The compiler embeds one or more such entries and, as noted in §5.3, refuses to emit a module whose trusted set is empty.

12.4 Sessions

After successful attestation the module establishes a session through `host_establish_session`. Session-gated imports such as `jwks_fetch` succeed only while a valid session exists; before that they return `NoSession` (-100). The module checks validity with `host_verify_session`, and an expired session returns `SessionExpired` (-101). The session is the precondition for any JWT-authorization logic the module chooses to enforce before releasing real content.

13. Execution Proofs

Beyond proving content genuine (§9), the format can prove the serving computation correct. An execution proof is a succinct, zero-knowledge attestation that the genuine module produced the response.

13.1 What Is Proven

A proof attests the statement: running the module whose hash is `program_hash` on the given request produces the given output. The proof is verified against the program hash, so a verifier needs no trust in the host and no secret from the module. The module embeds no proving secret; a valid proof exists only for a genuine run, so there is nothing to extract and nothing to forge.

13.2 Proof Structure

```
pub struct ExecutionProof {
   pub program_hash: Bytes32,    // hash of the served module
   pub public_input: Vec<u8>,    // request binding: client nonce + recent Chia block (see 13.8)
   pub public_output: Bytes32,   // commitment to the output bytes
   pub proof: Vec<u8>,           // succinct zero-knowledge proof
   pub chia_block: ChiaBlockRef, // recent Chia block header; bound into public_input (see 13.8)
   pub node_pubkey: [u8; 48],    // serving node BLS public key (G1)
   pub node_signature: [u8; 96], // BLS signature (G2) over (proof || public_input)
}

pub struct ChiaBlockRef {
   pub header_hash: Bytes32,     // header hash of a recent Chia block
   pub height: u32,              // block height
   pub timestamp: u64,           // block timestamp (unix seconds)
}

pub struct ProofResponse {
   pub proof: ExecutionProof,
   pub roothash: Bytes32,        // generation the response is bound to
}
```

13.3 The Proving Pipeline

The serving node executes the module on the request inside the proving environment, which emits the proof together with the public input and the output commitment. The proof is generated per request and bound to that request's public input.

13.4 Verification

A verifier checks the proof against `program_hash` and the public input, confirms the output commitment matches the bytes received, and confirms the response is bound to a root it trusts. Verification reveals nothing beyond the truth of the statement.

13.5 The Nonce Binding

The public input includes a fresh client nonce, so a proof is valid only for the specific request that produced it. A node cannot replay an old proof against a new request, and a relay cannot detach a proof from its request. Each genuine response carries its own non-reusable proof.

13.6 Cost and the Hardware Alternative

Generating a zero-knowledge proof per request is the most expensive operation in the serving path. Where that cost is prohibitive, the format permits an alternative: execution inside a trusted execution environment or an HSM-attested enclave that vouches for the run, trading the cryptographic proof for a hardware attestation of the same statement. Content retrieval and merkle verification are unaffected and remain in force regardless of which mode is used.

13.7 Node Attribution

The serving node is identified by the BLS key it already uses for attestation (§12), one key for both roles. `node_signature` is a BLS signature over `(proof || public_input)`, so the run is attributed to that node and the attribution is verified by BLS verification under `node_pubkey`. A proof therefore says not only that a genuine run occurred but which node performed it, and the client-nonce binding (§13.5) keeps the attribution from being lifted onto another request.

13.8 Chain-Anchored Freshness

A proof also commits to a recent Chia block header, which bounds when the proof could have been produced. The serving node reads the chain's current peak and binds that block's header hash, height, and timestamp into the proof's public input, so the zero-knowledge proof and the node signature both cover it.

A Chia block header hash cannot be known before the block is produced. A valid proof that commits to a given header therefore could only have been generated after that block existed. A verifier reads the committed timestamp and height, confirms the header is a real block on the chain it trusts, for example through a public Chia RPC, and rejects the proof if the block falls outside an acceptable freshness window. This anchors a proof to wall-clock time on Chia rather than to the serving node's own clock.

The block commitment complements the client nonce (§13.5). The nonce makes each response unique to its request and defeats replay; the block header makes each response provably recent and defeats precomputation. A node cannot pre-generate proofs ahead of time, because it cannot predict future block hashes, and it cannot revive a stale proof, because the committed block would fall outside the window. Together they bind a proof in both dimensions: to this request, and to this point in the chain's history.

14. Oblivious Retrieval

Two mechanisms keep a provider from learning what a client requested: decoys make a miss look like a hit, and oblivious access makes the in-module access pattern independent of the resource.

14.1 The Indistinguishability Goal

A provider that serves requests must not be able to separate real lookups from misses, nor infer which resource a real lookup concerned. The format pursues this so that enumeration yields no signal and traffic patterns leak nothing about content.

14.2 Decoys

A retrieval key that resolves to no entry does not produce an error. The module returns deterministic decoy bytes whose size is drawn from a logarithmic distribution seeded by the requested key, accompanied by a real-looking proof blob, under a normal success status. A miss returns a 200, never a 404. Because the decoy is a deterministic function of the key, the same miss always returns the same bytes, so a decoy is indistinguishable from a genuine immutable response and is as cacheable as any other. On the wire and in a cache, a real lookup and a miss look alike.

14.3 Oblivious Access

When a real lookup is served, the module gathers the resource's chunks from the interleaved pool (§8.3) through an oblivious access pattern: the sequence of pool accesses is randomized and padded so that it does not reveal which chunks, or how many, belong to the requested resource. The access pattern a provider could observe is decorrelated from the request. Combined with the absence of resource boundaries in the pool, this denies the provider both the content and the shape of what it serves.

14.4 Re-Randomization

Repeated retrievals of the same resource do not reproduce an identical observable access pattern. The access is re-randomized per execution, so a provider cannot identify a resource by the recurrence of a fixed pattern. A cache hit, by contrast, performs no execution at all and therefore exposes nothing.

15. Provider Blindness and the Neutral Pipe

Provider blindness is the consequence the whole format is arranged to produce. It is structural, not a matter of provider policy.

15.1 What the Provider Holds

A provider holds the module, which is code plus an interleaved pool of ciphertext and filler plus public metadata and public keys, and it receives requests addressed by a retrieval key, a 32-byte hash. It returns ciphertext and proofs. It holds no URN and no decryption key.

15.2 What the Provider Can and Cannot Learn

The provider can observe	The provider cannot learn
Module size and chunk count	The plaintext of any content
Retrieval keys requested (hashes)	The URN behind a retrieval key
That a request occurred and its timing	Which resource a request concerned, or whether it hit
The public metadata manifest (§8.4)	Anything the publisher did not place in the manifest

The retrieval key is a hash of the URN, so a provider cannot reverse it to recover the URN, and without the URN it can derive no decryption key. It cannot scan a store at rest, because the data section is ciphertext and filler, and it cannot inspect a request in flight, because the request is a hash.

15.3 Why This Is Structural

Neutral-pipe status does not depend on the provider behaving well. The keys it would need to read content are functions of a URN it never receives. A DIG Node is a neutral pipe by construction: even a hostile operator with full access to the module and every request still holds only ciphertext keyed by hashes. The guarantee is cryptographic, not contractual.

16. Temporal Keys

Access can be bounded in time. A capability may carry a validity window, and the module enforces it using the host clock.

The module reads the current time through `host_get_current_time` (§6.3) and checks a request's temporal component against the permitted window before releasing real content. A capability outside its window is treated like any other unauthorized request: the module returns a decoy, not an error, preserving indistinguishability (§14). Temporal bounding composes with the URN-as-capability model: the URN locates and decrypts, and the temporal check gates whether the module will serve at the moment of the request. Because the check uses the host-supplied time, it depends on the host clock the attestation handshake already established trust in (§12).

17. The Secretless Module

The module embeds no secret of any kind. This is the property that makes wide replication safe and provider blindness structural.

17.1 Obfuscation

The compiler can optionally apply code obfuscation to the serving logic: deterministic instruction substitution, opaque predicates, bogus code paths, and control-flow nops (§3). Obfuscation raises the cost of reverse-engineering the module's logic, but the format's security does not rest on it. A fully de-obfuscated module still reveals only ciphertext, public metadata, and public keys, because there is no secret in the binary to recover.

17.2 No Embedded Secret

- **No decryption key.** Content keys are derived from URNs and held by readers (§11), never stored in the module.
- **No signing key.** A store's signing identity is the publisher's wallet BLS key, used out-of-band; the module does not carry it.
- **No proving secret.** Execution proofs are verified against the public program hash and use no embedded secret (§13).

Because there is nothing to extract, copying the module is safe: a stolen module is exactly as useful as a public one, which is to say it serves only ciphertext to anyone lacking the URNs. Security lives in the URNs, not in the artifact.

Part 4: Runtime and Tooling

18. The Host Runtime

The host runtime (`dig-host`) loads a module, supplies the `dig_host` imports, and executes exports inside a sandbox with hard bounds. It never parses content out of the module; it interacts only across the ABI (§6).

18.1 Instantiation

The runtime parses and validates the module, instantiates it on a wasmtime-class engine, and wires the host imports. The data section is opaque to the runtime: it is reached only by calling exports, which return packed pointer/length results into linear memory.

18.2 Execution Bounds

- **Wall-clock timeout.** Each export call is bounded in time; an overrun is terminated.
- **Memory ceiling.** The runtime enforces an outer memory limit above the module's declared 16 MiB cap (§5.1).
- **Metering.** Instruction or fuel metering may bound work per call.

18.3 The Import Surface

The runtime implements the imports the module expects: `host_get_public_key`, `host_create_attestation`, `host_establish_session`, `host_verify_session`, `jwks_fetch`, `host_get_current_time`, `host_random_bytes`, and `host_read_return_buffer`. Results are written into the shared return buffer (§6.4) and read back by the module.

18.4 Serving a Request

```
1. host calls alloc(request_len) -> ptr
2. host writes the encoded request at ptr
3. host calls get_content(ptr, request_len) -> packed i64
4. if is_error(packed): handle error code
  else: (out_ptr, out_len) = unpack_ptr_len(packed)
5. host reads out_len bytes at out_ptr (ciphertext + proof, or decoy)
6. host calls dealloc as needed
```

The same pattern serves `get_proof`. The runtime returns to the client exactly what the module produced: it neither decrypts nor inspects the payload.

19. Compilation

Compilation is the deterministic transform from on-disk generations to a single module file (§5.3). This section records its inputs and the trusted-key input specifically.

19.1 Inputs

- The store configuration (`store.json`) and store ID.
- Every generation directory under `generations/`.
- The set of trusted host keys to embed (§12.3).
- Compiler options: obfuscation, optimization.

19.2 Trusted Host Keys

```
pub struct TrustedHostKey {
   pub public_key: [u8; 48], // host BLS public key (G1)
   pub label: String,        // versioned label, e.g. "dig-host-key-v1:<hex>"
}
```

At least one trusted host key is required. The compiler refuses an empty set (`CompilerError::NoTrustedKeys`, §5.3). These keys are the identities a running host must attest against before the module will serve content (§12).

19.3 Determinism

Compilation is deterministic: the same inputs produce byte-identical output. This is what lets the program hash function as a stable identifier of a module and lets a verifier confirm an execution proof against it (§13). Two parties compiling the same store with the same trusted keys obtain the same module and the same program hash.

19.4 Output

The compiler writes `{hex(store_id)}-{hex(roothash)}.wasm` atomically, via a temporary file and a rename, and reports the output path, size, and statistics (§5.3).

20. Developer Experience: A Git-Compatible Workflow

The command-line interface (`digstore`) presents Git's verbs and mental model. The chunking, URN-derived encryption, merkle commitment, and compile-to-WASM steps run beneath the familiar surface.

20.1 init

`digstore init` creates a store: it generates or accepts a store ID, records the visibility choice (public or private, §4.1), and writes the local configuration and directory layout (§4.4).

20.2 add

`digstore add <path>` stages content into the binary staging area. Content is chunked by content-defined chunking (§8.1) as it is staged, and chunk hashes are computed for deduplication.

20.3 commit

`digstore commit` finalizes a generation: it deduplicates the staged chunks, builds the generation's merkle tree, records the new root in the history, and compiles a new module (§5.3). The root hash is the commit identifier.

20.4 log and diff

`digstore log` lists generations in order, each identified by its root hash. `digstore diff <a> <b>` compares two generations by their chunk sets and resource keys.

20.5 checkout

`digstore checkout <root>` materializes a generation's content locally by reading resources out by URN through the module, rather than maintaining a persistent working tree (§2).

20.6 clone

`digstore clone <urn|url>` fetches a store's module from a remote and verifies it locally (§21). A clone obtains the whole module, the single distribution unit.

20.7 Command Reference

Command	Form	Effect
init	`digstore init`	Create a new store and local layout
add	`digstore add <path>`	Stage and chunk content
commit	`digstore commit`	Create a generation and compile the module
status	`digstore status`	Show staged changes
log	`digstore log`	List generations (root = commit id)
diff	`digstore diff <a> <b>`	Compare two generations
checkout	`digstore checkout <root>`	Materialize a generation
cat	`digstore cat <urn>`	Read a resource by URN
remote	`digstore remote add <name> <url>`	Configure a remote
clone	`digstore clone <urn|url>`	Fetch a store from a remote
push	`digstore push <remote>`	Push to a remote (§21)
pull	`digstore pull <remote>`	Pull from a remote (§21)

21. Remotes: Push and Pull over HTTPS

A remote is an HTTPS endpoint that stores and serves a store's module and accepts authorized updates. The remote protocol supports `clone`, `fetch`, `pull`, and `push` over a small REST surface.

21.1 Overview

A client configures a remote with `digstore remote add <name> <url>` and thereafter clones, pulls, and pushes against it. The remote serves the module and content the way any host does (§18), so a remote is also a serving provider and inherits provider blindness (§15).

21.2 REST Surface

Method and path	Purpose
`GET /stores/{storeID}`	Store descriptor: current root, size, public key
`GET /stores/{storeID}/roots`	Root history
`HEAD /stores/{storeID}/module`	Existence, size, and ETag (the current root)
`GET /stores/{storeID}/module`	Download the module (`application/wasm`)
`PUT /stores/{storeID}/module`	Push a new module or delta (authorized)
`POST /stores/{storeID}/content`	Retrieve content by retrieval key, root, and range
`POST /stores/{storeID}/proof`	Retrieve a proof for a retrieval key and root
`GET /stores/{storeID}/delta?from=&to=`	Chunk delta between two generations
`POST /stores/{storeID}/delta`	Negotiated delta from a client have-summary

21.3 Clone and Fetch

`clone` downloads the module via `GET /module` and verifies it locally. `fetch` retrieves the descriptor and root history to learn whether the remote holds a newer generation, without downloading content.

21.4 Pull and Head Advancement

`pull` brings the local store up to the remote's current head, downloading the module (or a delta, §21.5) for the newer generation and verifying it. The remote's head is the generation it currently serves.

A remote may decouple acceptance of a push from advancement of the served head. A remote MAY accept a push into a pending or staged state and defer advancing the head until an external authorization completes. A pending generation is not served: until the head advances, the remote continues to serve its last confirmed generation, and clients pulling from the remote receive that confirmed generation rather than the pending one. What constitutes the external authorization is defined by the remote and is outside the scope of this format.

21.5 Delta Sync

Rather than transfer a whole module, a client and remote can exchange only the chunks that differ. `GET /delta?from=&to=` returns the chunks present in the target generation and absent from the source along a linear ancestry; `POST /delta` negotiates a delta from a client-supplied summary of the chunks it already holds. Because chunks are content-addressed and deduplicated (§8), the delta is exactly the new chunk set plus the key-table changes.

21.6 Push

A push uploads a new module, or a delta, to the remote through `PUT /module`. The push is authorized by a signature: the publisher signs `SHA-256` of the pushed root, bound to the store ID, with the store's BLS key, and the remote verifies that signature against the store's public key. A remote MAY additionally require a bearer token alongside the BLS signature, providing a transport-level authorization check independent of the signature itself. A push must be a fast-forward of the remote's current head; a non-fast-forward push is rejected (§21.8). Acceptance of the push and advancement of the served head may be separated as described in §21.4.

21.7 ETags and Caching

The module's ETag is its root. A client holding a generation issues a conditional request with `If-None-Match` and receives `304 Not Modified` when its root matches the remote's head, avoiding a re-download. Because content at a root is immutable, cached module and content responses remain valid until the root changes; a new generation is a new root and a new cache identity, so updates never invalidate prior cache entries.

21.8 Status Codes

Code	Meaning
200	Content (real or decoy), descriptor, or module bytes
201	Push accepted; a new generation is now the head
202	Push accepted into a pending state; head not yet advanced (§21.4)
304	`If-None-Match` matched the current root
401 / 403	Push not authorized: bad signature, or missing bearer token where required
404	Unknown store, or unknown root for a module or descriptor. Never for content.
409	Non-fast-forward push: the pushed parent is not the current head
413 / 422	Module exceeds the size limit, or failed validation
429	Rate limited

Part 5: Properties and References

22. Security Properties

A summary of what the format guarantees, and on what it depends.

Property	Rests on

Confidentiality of content	URN secrecy; HKDF-SHA256 key derivation; AES-256-GCM (§11)
Integrity of responses	SHA-256 merkle proofs bound to a trusted root (§9)
Authenticated encryption	AES-256-GCM tags; unique key per URN (§11.2)
Provider blindness	Retrieval and decryption keys are functions of a URN the provider never holds (§15)
Indistinguishability	Deterministic decoys and oblivious access (§14)
Correct execution	Zero-knowledge proofs against the program hash, or hardware attestation (§13)
Host trust	BLS attestation against compile-time trusted keys (§12)
Secretlessness	No decryption key, signing key, or proving secret in the module (§17)

These properties hold as long as the standard cryptographic primitives remain secure: SHA-256, AES-256-GCM, HKDF-SHA256, BLS signatures, and the zero-knowledge proof system. The URN is a capability, so confidentiality is contingent on URNs being kept secret by those they are shared with. The format defends content against a hostile provider and a network observer; it does not by itself defend a URN that its holder discloses.

23. Acknowledgements and References

Digstore stands on established work. Content-defined chunking follows the rsync and FastCDC line with a gear-based rolling hash. Merkle commitments follow Git and Ethereum. Authenticated encryption uses AES-256-GCM; key derivation uses HKDF-SHA256. Execution and sandboxing build on WebAssembly and wasmtime-class runtimes. Signatures and host attestation use BLS over BLS12-381, as in the Chia ecosystem. Execution proofs draw on the zkVM and proof-carrying-code literature.

Representative implementation crates:

Concern	Crate
WASM execution	`wasmtime`
WASM emission and parsing	`wasm-encoder`, `wasmparser`
Hashing	`sha2`
Authenticated encryption	`aes-gcm`
Key derivation	`hkdf`
Signatures (BLS)	`chia-bls` (`blst`)

Content-defined chunking	`fastcdc`

This document specifies the local store format and its HTTPS remote protocol. Network distribution across many providers, external identity anchoring of the store ID, and payment settlement are out of scope and are addressed by separate DIG Network specifications.