WHITE PAPER

Two Strangers, One Machine

The security model: what each party is exposed to, and what neither has to trust

The `.ee` format, its three levels of key custody, and an honest account of where the limits are

VERSION 1.0

CLASSIFICATION ·PUBLIC

CONTENTS

What this paper contains

FIGURES				
	01 System architecture	02 Level 2 request flow	03 Container layout	04–05 Key custody · Two-sided threat model

01 Executive Summary

Two strangers meet at an Edge Node.

The first is a company with content to deliver. They hand it to us to serve from hardware they will never see, in a house they cannot name, owned by someone they will never meet.

The second is a person with a machine sitting mostly idle in that house. They give over disk and bandwidth on it to software they did not write, so that it can hold and serve files belonging to companies they have never heard of.

Neither is being unreasonable. Both are being asked to trust the same thing — an environment neither of them controls. This paper is an account of how much of that trust we managed to remove, and exactly where we could not.

Most security architectures approach this by asking how the node can be made trustworthy. That question has no good answer when the machine underneath the software belongs to somebody else. So we asked a different one.

> The question is not how much we can trust an Edge Node. It is how little we have to.

The answer is the `.ee` container format and three encryption levels. A customer picks a level at packaging time, and the level's central consequence is key custody: who holds the key that opens the content.

LEVEL	WHO CAN READ THE CONTENT	WHAT THE EDGE NODE HOLDS
Level 0	Anyone	Plaintext. Public content; nothing is hidden from anyone.
Level 1	The browser only	Ciphertext only. **The node has no key and cannot decrypt.**
Level 2	The node, per request	A key fetched per request, released when the response ends.

Level 1 is the strong claim, and it holds. The workload's index is encrypted with a key the Edge Node never receives. The node cannot enumerate the files inside a workload, let alone read them. It serves opaque blocks and the browser SDK does the decryption. Level 2 trades that property for server-side file addressing, range requests and central revocation, and this paper is explicit about what it gives up in exchange.

The distinction that matters at Level 1 is between *cannot* and *will not*. An operator who promises not to look at your content has made a commitment, and commitments can be broken, forgotten, or overridden by whoever acquires the company. An operator who holds no key has not made a commitment at all. There is nothing for them to break. This paper is about moving as much of the guarantee as possible out of the first category and into the second — and about being exact regarding where that becomes impossible, which it does at Level 2.

What we do not claim

Three things get claimed routinely in this category that we are not going to claim.

Not information-theoretic security. The system rests on AES-256-GCM and HKDF-SHA256. Those are computational assumptions — secure against a bounded adversary, not against unlimited computation. That is the ordinary basis for every deployed cryptosystem and it needs no apology. It needs only not to be dressed up as something stronger.

Not unlinkability between nodes. Content addressing is how caching, deduplication and integrity work here. Two Edge Nodes holding the same artifact can always determine that they do.

Not forward secrecy. We have a different and, for this use case, more useful property. Section 9.4 describes it and says why it is not forward secrecy.

What this paper is about, and what it is not

This describes a **distribution** model: how a packaged unit of customer content — a *workload*, in this paper's vocabulary, meaning a `.ee` file rather than a running process — reaches an Edge Node, and how little that node must be told in order to serve it. The format was designed to be independent of what the bytes inside it represent, which is why it is described as a distribution substrate rather than a delivery feature.

It is not an **execution** model. Evolving Edge does not run customer code on Edge Nodes; nodes cache and serve files. Keeping a *running* workload unreadable is a different problem, is not solved by anything here, and is not claimed. Section 3.3 states the roadmap boundary in full.

Performance and compliance

This paper contains no latency table, no throughput figure and no compliance attestation. Those belong in a security paper only when there is a benchmark or an auditor standing behind them, and a performance chapter will appear here when there is a benchmark with hardware, workload and method stated. On compliance posture, see Chapter 11.

02 Why Trust Has to Live in the Architecture

2.1 Every marketplace you have studied solves trust after the transaction

eBay. Etsy. Uber. Airbnb. The mechanism these marketplaces are known for is reputation — reviews, ratings, dispute resolution. A paper trail and a human process. They lean on other things too, escrow and background checks and insurance among them, but reputation is what made connecting strangers feel survivable at scale.

Reputation depends on one thing nobody ever names. An **interval**. The ride ends, and then you rate it. The package arrives, and then you open it. The stay finishes, and then you say what the place was actually like. There is a window between the transaction and the verdict, and the entire apparatus lives inside that window, converting other people's past experience in it into your confidence about the next one.

2.2 Here, the window closes before it opens

A request arrives at an Edge Node and is served. For the property this paper is about — whether the content was readable on that machine — there is no interval afterward in which to find out. By the time a customer would have something to complain about, the thing they would complain about is over. Their content was exposed or it was not, and no rating anybody leaves changes which.

For confidentiality, reputation is not a weak mechanism in this market. It is an unavailable one.

That is a claim about one property, not about the whole system, and the distinction matters because this paper later relies on after-the-fact evidence for other things. Level 2 key issuance is audit-logged, abnormal issuance patterns are detectable, and customers can read their own audit feed (Sections 8.4 and 9.6). Those are real and useful. What they cannot do is un-expose content that has already been served. Detection works; prevention is what has no window.

The same holds in the other direction. An operator cannot wait and see whether their content exposure was acceptable, for the same reason.

That leaves one place for the confidentiality guarantee to live: in the architecture. Not in a policy document, not in an operator agreement, and not in a reputation score. If it is not a property of the system, there is no later moment at which it can be repaired.

This is the reason the rest of this paper is about key custody rather than about controls, procedures or commitments. Key custody is the part of the design that does not depend on anyone's conduct.

2.3 Each party owns exactly half

It is worth being precise about the asymmetry, because it is easy to get wrong.

This is not hardware nobody owns. Everything here is owned.

The operator owns the hardware. Not the process running on it.
The customer owns the workload. Not the machine it is served from.

Each party controls exactly half of the thing they are depending on, and the two halves are not equal in power. An operator with root can reach into any process on their own machine — read its memory, attach a debugger, capture its traffic. Control of the machine beats control of the process, every time. So the boundary a customer would most like to rely on, the one between "Evolving Edge's software" and "somebody else's box," is not a boundary that holds against an adversary who owns the box.

A customer would otherwise be asked to trust two parties: us, for what the software does, and a stranger, for what their machine is permitted to do to that software. An architecture that resolves to "trust both" is not one a security team should accept, and not one we would want to defend.

So Evolving Edge designs on the assumption that every Edge Node operator will use every capability their hardware ownership gives them. Not because we expect them to, but because a guarantee that depends on them declining to is not a guarantee.

> Our starting assumption is that an Edge Node operator is not a trusted insider — even though customer content passes through their machine.

Chapter 4 states the same assumption in the other direction, because the operator did not sign up to be trusted either.

2.4 How the existing answers work

Four approaches are in general use, and each makes a different thing the object of trust.

Centralized cloud asks you to trust the provider. This works, is well understood, and is the reason most content is delivered from a small number of large facilities. It also means the provider can read your content, and the facility is where it is, not where your users are.

Trusted execution environments move the trust into the silicon. An enclave protects code and data from the host operating system, which is exactly the property we would want here. It also relocates the trust rather than removing it: the object of trust becomes the hardware, on machines we do not select and cannot audit in a residential deployment.

Homomorphic encryption lets a server compute on data it cannot read. It is a genuine advance and, for the record, it is *computationally* secure — it rests on lattice assumptions, not on information theory. Its practical limits are the overhead and the narrow set of operations available, both of which vary enormously by scheme.

Secure multi-party computation splits a computation across parties so that no single party learns the input. It is interactive, which costs round trips, and the information-theoretic security sometimes attributed to it is a property of specific honest-majority protocols, not of MPC in general.

None of these is wrong. The last two simply answer a different question — one about *execution*, how a machine computes on data it cannot read. This paper answers a question about *distribution*: how a packaged unit of content gets onto a machine at the edge, and how little that machine has to be told in order to serve it. Evolving Edge does not execute customer code on Edge Nodes, so the execution question is not one this system currently faces; Section 3.3 says why it appears here at all.

2.5 What this paper covers

The `.ee` container format, its three encryption levels, and the security property each one actually provides. Every substantive claim in Chapter 9 is stated as a claim, given a basis, and paired with the test that checks it. Where a property does not hold, or holds with a caveat, that is written down in the same place as the claim rather than in a footnote somewhere else.

03 The Evolving Edge Architecture

3.1 Three components, three different levels of trust

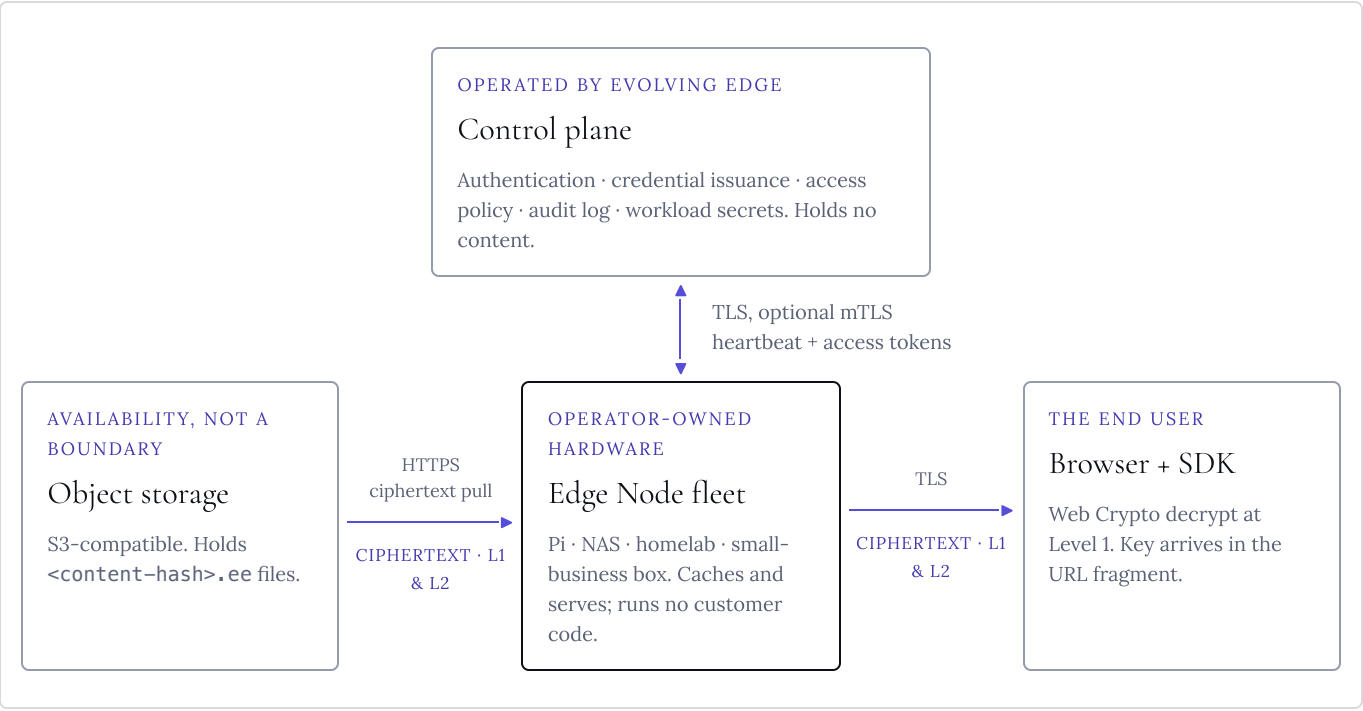

FIGURE 1 · System architecture. The storage→node and node→browser legs carry ciphertext for Level 1 and Level 2. The heavier rule marks the one component on hardware Evolving Edge does not own.

The control plane is operated by Evolving Edge. It authenticates customers, mints credentials, brokers access decisions, holds workload secrets, and serves as the authoritative metadata layer. It does not hold customer content.

Edge Nodes are the Evolving Edge node software, running on operator-owned hardware — a Raspberry Pi, a NAS, a homelab server, a small business box. They cache workloads locally and serve them to nearby end users. Their proximity to end users is the reason the network exists, and their ownership is the reason this paper does. The software is ours; the machine is not; and as Section 2.3 sets out, the machine wins.

Object storage is S3-compatible and holds the `.ee` workloads. It is an availability component, not a confidentiality boundary. For Level 1 and Level 2 workloads the bytes in the bucket are already ciphertext; losing the bucket would be an availability incident, not a disclosure.

3.2 What each component holds

COMPONENT	OPERATED BY	HOLDS CONTENT?	HOLDS KEYS?
Control plane	Evolving Edge	No	Workload secrets (Level 2 issuance)
Edge Node	Independent operator	Yes, as `.ee` ciphertext	Level 2 only, per request
Object storage	Evolving Edge / provider	Yes, as `.ee` ciphertext	No
End user browser	The end user	Decrypted content	Level 1 only, from the URL fragment

The row that matters is the second one. An Edge Node holds content and, for the level most customers should be using, holds no key that opens it.

3.3 Three phases, one of which ships

What Evolving Edge does not do

> **Evolving Edge does not yet execute customer code on Edge Nodes.** No customer workload runs on operator hardware today. An Edge Node caches and serves files. It does not yet host inference, it does not yet run containers or virtual machines, and it does not yet accept arbitrary customer compute. Nothing in this paper describes a running-workload capability outside the lab, because there is not one.

We lead with that because the rest of this section describes a roadmap, and a roadmap in a security paper is an invitation to misread.

The roadmap, and why it appears here at all

Evolving Edge is building toward edge compute in three phases: **edge content delivery**, then **edge AI inference**, then **general edge compute**. Only the first is built. The second and third are only available in the lab - not in limited availability, not in private beta, and not running for any customer.

The reason to mention them in a security document is narrow. The model described in this paper is not a CDN feature bolted onto a delivery product. It is the distribution substrate, and it was designed to be phase-independent — so a security team evaluating it is evaluating something we expect to still be defending in three years, not something that gets replaced when the product changes shape.

What the substrate is, and what is genuinely invariant

A `.ee` workload is a packaged, content-addressed, independently keyed unit. Today the contents are files. The format was designed so that the container, its integrity properties and its transport path do not care what the bytes represent.

The following are intended to hold regardless of phase:

- The `.ee` container, its cleartext header, its encrypted index and its per-block AEAD
- Content addressing by BLAKE3 hash, and hash-only artifact naming with no customer identifier in the filename
- Node identity, machine-bound credentials and enrollment (Chapter 7)
- The *question* of key custody — who holds the key that opens a workload — as the axis that decides who can read it

Note what is deliberately absent from that list. **The three encryption levels are not claimed to carry forward unchanged.** Level 1's guarantee depends on the Edge Node never receiving a key, which is workable when the node's job is to hand ciphertext to a browser that decrypts it. A node cannot execute an image it cannot decrypt. So an execution phase requires either a different custody model or an isolation boundary that can hold a key without the operator reaching it — and which of those it will be is an open design question, not a settled one. Any claim that Level 1 extends to inference or general compute would be a claim we cannot currently support.

Likewise, per-request access tokens and the audit trail are Level 2 mechanisms, not platform-wide ones. Level 0 has no tokens and no access decisions, and its index is cleartext, including file paths and MIME types.

What execution would require, and does not have

An execution phase introduces a second and genuinely different problem: keeping a workload unreadable while it is *running*, not merely while it is stored and transmitted. The approach we are investigating is encrypted micro-VMs, placing execution inside an isolation boundary rather than in an ordinary process on an operator's machine.

> **None of that exists.** No micro-VM isolation ships at this time. No runtime-protection property is claimed, designed or tested in production. The limits in Section 8.4 and Section 9.5 describe the system as it is.

A reader evaluating this platform should evaluate the distribution model, because that is the entirety of what runs. The execution model will need its own paper, its own threat model and its own tests, and will get them when there is something to test.

04 The Security Model

4.1 Computational security, stated plainly

Everything in this system rests on AES-256-GCM, HKDF-SHA256 and BLAKE3. These are computational assumptions: secure against an adversary with bounded resources, not against unlimited computation.

This is the ordinary footing of every deployed cryptosystem in production anywhere. We state it here so that nothing later in the paper has to be read as implying otherwise.

4.2 The threat model runs in two directions

Most threat models in this category have one adversary and one victim. This one has two parties who are each, from the other's point of view, the risk.

The customer asks	ONE MACHINE	The operator asks
Is my content readable by whoever holds the machine?	EXPOSURE	Is my machine reachable by whoever holds the content?
Can anything else on that box contaminate what I put there?	CONTAMINATION	Can what they put there contaminate anything else on my box?
Can I verify nothing was tampered with?	VERIFICATION	Can I verify nothing is happening that I did not agree to?
ANSWERED BY Key custody and content addressing — the rest of this paper.		ANSWERED BY The absence of customer code, plus operator docs and the agreement. Not by cryptography.

FIGURE 5 · The two-sided threat model. The same three questions — exposure, contamination, verification — asked from opposite ends of one machine. The symmetry is in the questions; the mechanisms that answer them are not symmetric.

What the customer is exposed to

THE ADVERSARY CAN	THE ADVERSARY CANNOT
• Run arbitrary code as root on any number of Edge Nodes • Read node memory, disk and network traffic • Observe request timing, object sizes and cache behavior • Collude across any number of compromised nodes	• Break AES-256-GCM or BLAKE3 • Compromise the control plane

The second assumption deserves to be said out loud rather than buried. The control plane holds workload secrets. Compromising it compromises Level 2 content. That is a single point of trust, and a paper that positions itself against centralized trust models is obliged to name its own.

> Level 1 content survives a control plane compromise, because the control plane never holds a key that decrypts Level 1 content for anyone.

That asymmetry is the single most important reason to choose Level 1 for high-sensitivity workloads.

What the operator is exposed to

An operator is not a vendor and has not accepted a duty of care. They are a person who agreed to let a machine in their home do useful work, and they carry their own risk for doing it. A threat model that described only the customer's risk would be describing half of a two-sided arrangement and calling it the whole thing.

One structural fact shapes the operator's exposure today: an Edge Node caches and serves files, and no customer code executes on it. What runs on an operator's machine is Evolving Edge's own software. Its job is to fetch hash-named ciphertext from storage, verify it, hold it in a cache directory, decrypt it for Level 2 requests the control plane authorizes, and serve bytes over TLS to nearby end users.

That enumeration deliberately includes the Level 2 decryption step, because it is the one thing an operator's machine does with a stranger's protected content, and Chapter 8 describes it in full rather than leaving an operator to find it there.

THE OPERATOR'S CONCERN	WHERE IT STANDS TODAY
Their files and home network	There is no customer-supplied code to contain, because none is accepted. What the node process itself can reach on the host, and the fact that it listens for inbound requests, are properties of the node software and its installation — not cryptographic guarantees, and not claims made by this paper.
Their cost	Bandwidth and power are the operator's real ongoing exposure, and they are a commercial question rather than a security property. This paper does not address terms.
Their own use of the machine	Resource use and the ability to stop are operational properties of the node software and the operator agreement, not cryptographic ones.

The first row is worded carefully and the wording is the point. We are not claiming that an isolation boundary protects an operator from a hostile customer workload. We are saying there is no customer workload to be isolated from. That claim holds only while nothing executes, and Section 3.3 is explicit that it stops holding if execution phases arrive — at which point the operator's column becomes the harder half of this threat model and will need its own analysis, not a paragraph.

What this paper does **not** establish, and an operator should not read into it: any bound on what the node process can reach on the host filesystem or the local network. No sandboxing, user isolation or capability restriction is described here, because none is being claimed. Those are questions for operator documentation and for the agreement, and an operator evaluating the deal should ask them there.

The same three fears, pointed in opposite directions

Read either column of Figure 5 on its own and it looks like an unusually suspicious list. Read across and it is the same three questions — exposure, contamination, verification — asked from opposite ends of one machine.

The two columns are not answered by the same thing, and it would be convenient but false to suggest otherwise. **The left column is what key custody and content addressing answer**, and the rest of this paper is that answer. **The right column is answered today by the absence of customer code**, plus operator documentation and the agreement — not by cryptography. The symmetry is in the questions, not in the mechanisms.

What the two columns do share is why neither can be answered with an assurance. An assurance has to be extended to somebody, and these two parties have no basis for extending one to each other. Chapter 2 explains why they will not acquire one.

4.3 What each level guarantees

Level 0 — integrity only. The content is public. Integrity is enforced; nothing is hidden, and nothing is claimed to be.

Level 1 — confidentiality against the Edge Node *and* against the control plane, for as long as AES-256-GCM holds. The key lives in the browser and reaches it through the URL fragment, which by specification is never transmitted to any server.

Level 2 — confidentiality against anyone who cannot obtain a per-request access token, and against the Edge Node outside the window of a request that node is authorized to serve.

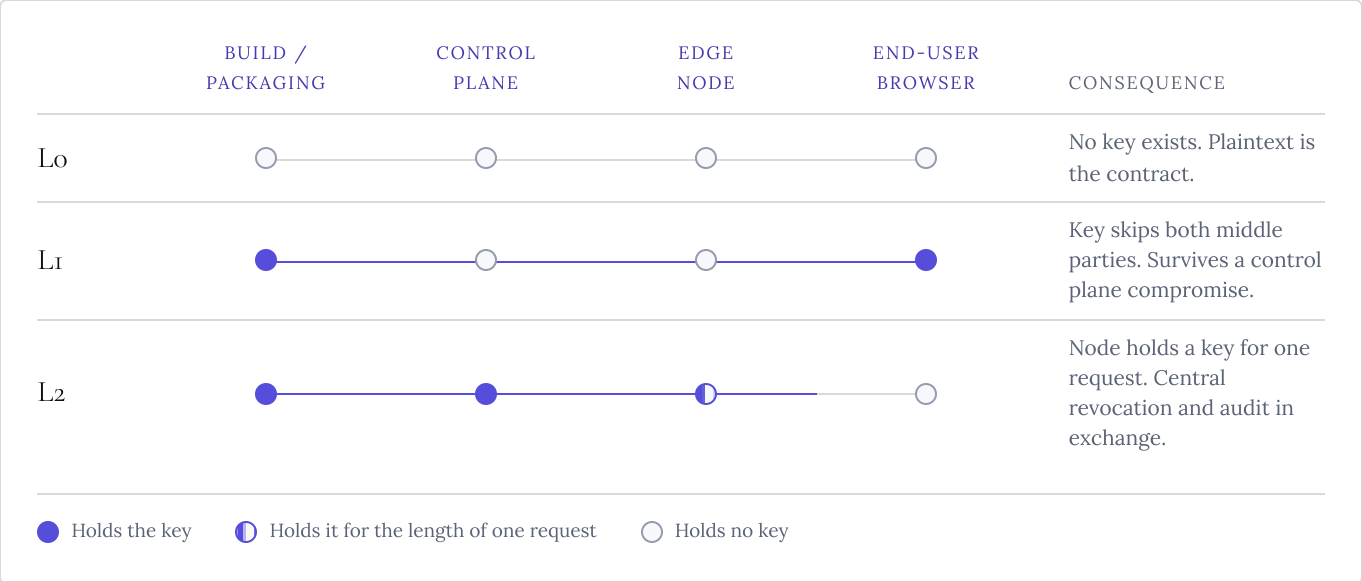

FIGURE 4 · Key custody across the three levels — the one control of the four that changes the trust story. Read each row left to right as the path a workload's key does, or does not, travel.

4.4 Four controls, not ten

It is conventional in this genre to enumerate as many security layers as can be made to sound like layers, and then to assert that an attacker must defeat all of them. We are not going to do that, because a reviewer who counts will find that some of the layers are not controls.

Compression is not a control. It is a side channel, and Section 9.3 treats it as one. Erasure coding provides availability, not confidentiality. Content expiry is a retention policy.

The controls that actually carry the trust model are four:

01	02	03	04
AES-256-GCM over content	AES-256-GCM over the index	Per-request access tokens Level 2 only	Key custody The only one that changes the trust story

Four honest controls are worth more in front of a security reviewer than ten inflated ones.

05 The Three Encryption Levels

Every customer who uploads content picks a level. The level is a property of the workload, fixed at packaging time, and recorded in the container header.

5.1 Level 0 — public content

The content is Brotli-compressed and stored as opaque blocks, but it is not encrypted. The Edge Node serves it after decompression. There are no keys, no tokens and no access decisions.

Level 0 is for content you would put on a public CDN anyway: marketing images, static JavaScript, a documentation site. Its integrity is protected — the content hash and per-block checks in Chapter 6 apply at every level — but anyone holding the file can read it, and we do not want a customer arriving at that discovery later.

5.2 Level 1 — client-side encryption

This is the strongest mode and the one we recommend for anything a customer would describe as sensitive.

The customer publishes URLs of this shape:

```
https://customer.example.com/ee/<content-hash>/file.bin#key=
<base64url(secret||salt)>
```

Everything after the # is the URL **fragment**. By HTTP specification a compliant client never transmits the fragment to any server — not to the customer's own origin, not to the Edge Node, not to the control plane. It exists in the user's browser and nowhere else on the path.

In the browser, the Evolving Edge SDK uses the Web Crypto API to:

1. Parse the secret and salt out of the fragment
2. Derive an AES-256 content-decryption key with HKDF-SHA256
3. Fetch the encrypted blocks from the Edge Node, which arrive as AES-256-GCM ciphertext
4. Decrypt and decompress in memory, in the page

The Edge Node serves ciphertext blocks and an encrypted index. It never invokes a decrypt path for a Level 1 workload, because it has nothing to decrypt with.

> If an operator copies an entire Level 1 `.ee` file off their disk and walks away with it, they have AES-256-GCM ciphertext bound to a key that exists only inside browsers that received a URL fragment.

There is a real cost to this, and it is worth naming: because the Edge Node cannot decrypt the index, it cannot address individual files inside the workload. It serves raw blocks and the SDK reassembles. Level 1 buys the strongest guarantee in the system by giving up server-side file addressing.

5.3 Level 2 — gateway-side, with audit and revocation

Some workflows need central revocation — licensed media when a subscription lapses, a paid download where the purchase is validated server-side, anything where "cut this person off now" has to be enforceable from one place. Level 2 exists for those.

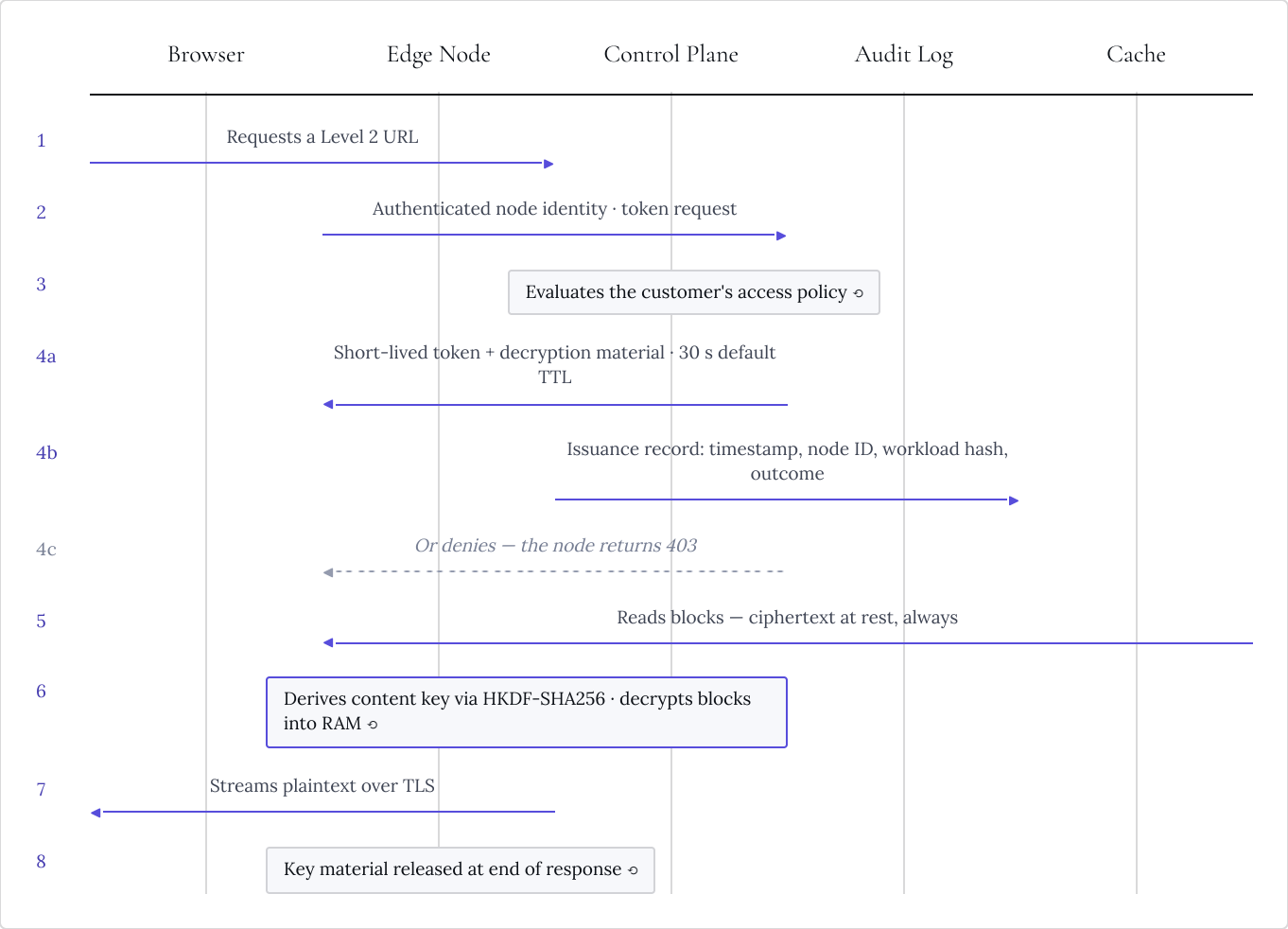

FIGURE 2 · The Level 2 request flow. The cache holds ciphertext at rest, always; plaintext exists only in the node's memory, for the length of one request. Step 4c is the denial path.

The flow:

1. A browser requests a Level 2 URL from an Edge Node.
2. The Edge Node holds the encrypted `.ee` file in its cache — the same AES-256-GCM ciphertext a Level 1 workload would be.
3. The node requests a short-lived access token from the control plane, presenting its own authenticated identity (Chapter 7).
4. The control plane consults the customer's access policy and either issues a token carrying decryption material, valid for **30 seconds by default** and enforced server-side, or denies the request, in which case the node serves a 403.
5. On approval, the node derives the per-workload content key with HKDF-SHA256, decrypts and decompresses the blocks into RAM for the lifetime of that one HTTP request, and streams the plaintext out over TLS.

6. Every key issuance is recorded on the control plane — timestamp, node ID, workload hash, outcome. Customers pull their own audit feed through the portal API.

Three plain statements about where plaintext exists in this flow:

- **At rest on the Edge Node:** never. The cache file is ciphertext.
- **On the wire to the end user:** yes, eventually — that is what delivery means. The channel is TLS-protected.
- **In operator RAM:** yes, during the request. Chapter 8 discusses what that actually means for a customer.

Revoking a Level 2 workload

Flip the access policy on the control plane. The next time any Edge Node requests a token for that workload, the request is refused and the node serves a 403. Tokens already issued expire on their own within the TTL.

For a harder cut-off, rotate the workload secret, which makes outstanding tokens useless. One caveat applies to both this and the stronger form described in Section 9.4, and it is load-bearing: the control plane currently caches workload secrets in memory with no expiry, so a rotation does not take effect on a running control plane instance until that instance restarts. Until that cache is bounded, neither revocation path should be described as immediate.

06 The .ee Format

6.1 Container layout

A workload is a single file with three parts: a fixed 512-byte cleartext header, the index, and the content blocks. At Level 1 and Level 2 both the index and the blocks are encrypted; at Level 0 they are compressed but not encrypted. The header carries the offset and size of the index, so a reader locates the index from the header rather than from a fixed position.

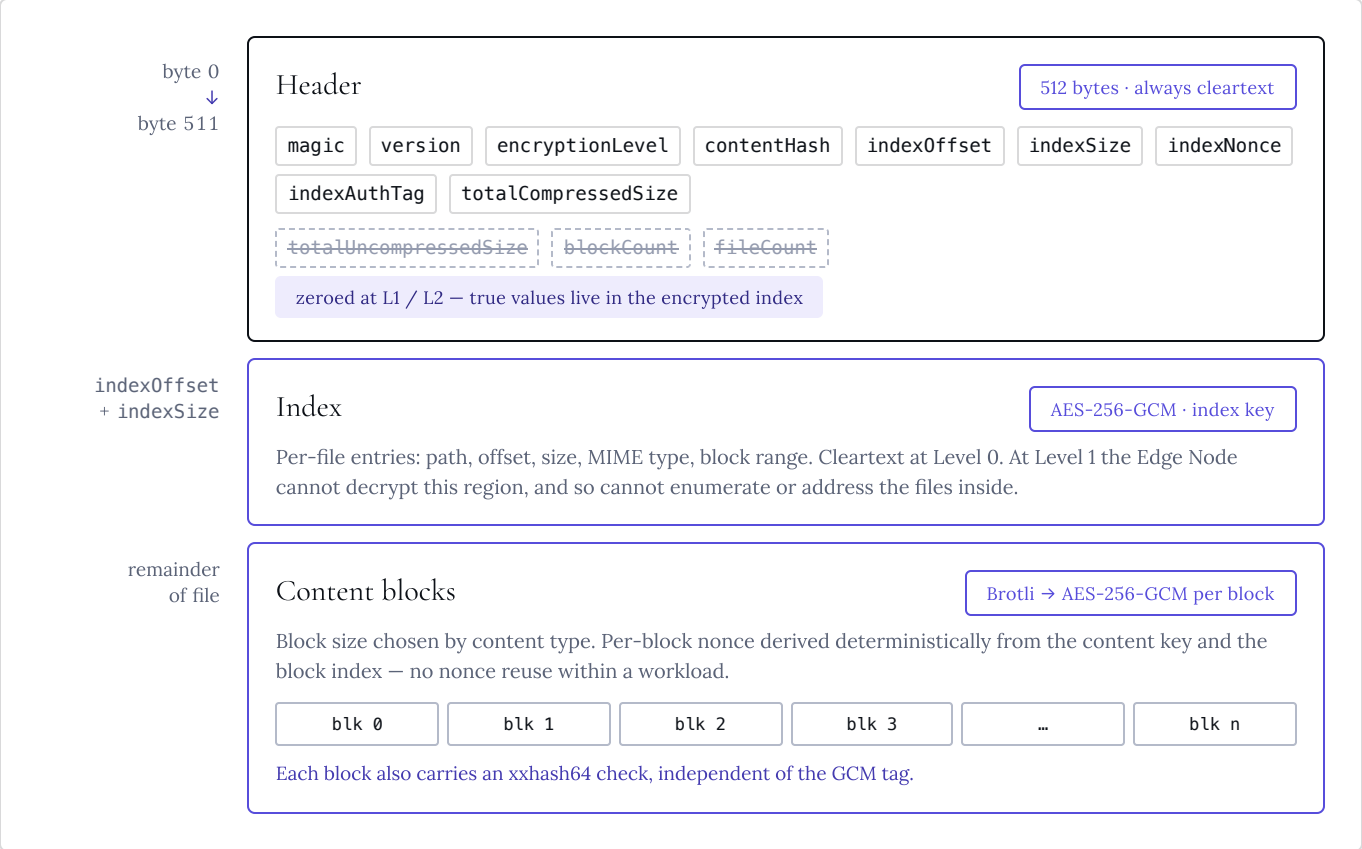

FIGURE 3 · The `.ee` container. The header must be cleartext — a reader needs the index offset, size and nonce before it can decrypt anything at all — so every field in it is treated as published. Struck fields are zeroed at Levels 1 and 2.

The header **must** be cleartext. A reader needs the index offset, the index size and the index nonce before it can decrypt anything at all. There is no way around that, so we treat every field in the header as published.

6.2 What the header reveals, and what it withholds

Because the header is cleartext at every level, the design question is not whether these fields are visible but which fields exist.

Withheld at Level 1 and Level 2 — zeroed in the header, with the true values carried inside the encrypted index:

- `totalUncompressedSize` — this is the exact plaintext length
- `blockCount` and `fileCount` — block structure and the number of files in the workload

Retained at every level:

- `contentHash`, `indexOffset`, `indexSize`, `indexNonce`, `indexAuthTag` — required to read the file at all
- `totalCompressedSize` — approximately the object size, which any observer already has

At Level 0 everything is retained, because the content is public.

> This was a real finding, not a hypothetical. Until it was fixed, the exact plaintext length of every encrypted workload could be read out of the first 512 bytes with no key and no work.

We include that because a reader evaluating this document should be able to see that the format has been looked at adversarially by people who were willing to publish what they found.

6.3 Compression

Compression is Brotli, applied at block level, with block size chosen by content type. It runs **before** encryption.

That ordering is the conventional one and it is the right one for delivery, but it has a consequence that Section 9.3 states in full: ciphertext length becomes a function of plaintext content. We would rather put that in the security analysis as a limitation than leave it for a reviewer to find.

6.4 Key derivation and custody

Each workload gets a 32-byte secret and a 32-byte salt, both generated fresh at build time. HKDF-SHA256 derives, from the secret and salt together with the content hash:

- a **master key**
- an **index key**, which encrypts the index
- a **content key**, which encrypts the blocks
- **per-block nonce material**, derived deterministically from the content key and the block index

Because nonces are derived per block from the content key and the block's own index number, there is no nonce reuse across the blocks of a workload. Because the secret and salt are per workload, there is no key reuse across workloads or customers.

Custody by level:

- **Level 1** — the key reaches the browser through the URL fragment. The Edge Node never receives it and cannot decrypt the index, so it cannot address individual files. It serves raw blocks; the SDK does the rest.
- **Level 2** — the node fetches key material per request, derives the content key, decrypts, and releases the key material when the response ends. Intermediate keying material is cleared immediately after derivation.

There is no HSM in this system and no threshold key ceremony. We mention that only because such things are sometimes described in documents of this type without being present in the corresponding code.

6.5 Content addressing and integrity

The BLAKE3 hash of the canonical content is in the header, and the file's name in storage is that hash. Substituting different bytes under the same address is not possible without producing a hash mismatch.

Integrity is checked twice, by design:

- **Per-block xxhash64.** A cheap non-cryptographic check that catches corruption early, before the AEAD is invoked. A mismatch aborts the serve.

- **AES-256-GCM authentication tags.** AEAD, so any tampering with a block's ciphertext is detected on decrypt and no plaintext is used before the tag verifies.

The file's name in storage is `<content-hash>.ee` and nothing else. Customer identity, intended audience, pricing and access policy are control plane state, not properties of the cached artifact. An operator holding the file holds a hash-named blob with no metadata attached to it.

6.6 Cryptographic primitives

Everything below is a standard, well-reviewed primitive implemented through a mainline library — Go's `crypto/*` packages on the node and control plane, the Web Crypto API in the browser. We have written no cipher of our own.

PURPOSE	ALGORITHM	WHY
Content hashing	BLAKE3	Fast, collision-resistant, parallelizable
Content encryption	AES-256-GCM, per block	AEAD; confidentiality and integrity together; widespread hardware acceleration
Index encryption	AES-256-GCM	Same primitive, independently derived key
Key derivation (workload)	HKDF-SHA256	Standard extract-and-expand over secret, content hash and salt
Key derivation (node's on-disk secrets)	PBKDF2-SHA256, 100,000 iterations	Slow-by-design KDF, appropriate for stretching machine entropy into an AES key
Block integrity	xxhash64, in addition to the GCM tag	Cheap early corruption check
Node transport	TLS 1.2+, ECDSA P-256 certificate	Standard mTLS
Compression	Brotli	Strong ratio across the typical delivery mix; native browser support

Cross-implementation test vectors

The same key derivation runs in three independent implementations — Go, TypeScript on Node, and the browser. Three implementations of one derivation is three chances to drift apart silently, and silently is the dangerous part: a divergence

in HKDF output does not produce a visible failure, it produces content that one implementation can read and another cannot.

We maintain a fixed set of cross-language test vectors that all three implementations must match exactly. Given a fixed secret, content hash and salt, the vectors pin the master key, the index key, the content key and the first two per-block nonces. Any implementation that drifts fails on the next CI run.

07 Node Identity and Enrollment

An Edge Node is a machine we do not own, asking to join a network and be handed customer content. How it proves what it is matters.

A new node joins as follows.

01	Pairing	The operator pairs a node with their account through the operator portal. The portal issues a one-time enrollment token of the form `ee-enroll-v1.<base64url(control-plane-URL)>.<encrypted-payload>`.
02	Enrollment	The node presents the token to the control plane.
03	Verification	The control plane verifies the token signature against an HKDF-derived key over the enrollment secret, and verifies that the token's nonce has not been used before. A replayed enrollment token is refused.
04	Issuance	The control plane mints a node ID and a fresh node API key. The raw key is returned to the node once; the control plane stores only its hash. Enrollment, owner, region and capacity are recorded.
05	Sealing	The node persists its node ID, node key, control plane URL, region and heartbeat interval to an encrypted config file.

Credentials are bound to the machine that enrolled them

The node's config file is AES-256-GCM ciphertext with a corresponding salt. The encryption key is derived with PBKDF2-SHA256 at 100,000 iterations from machine-specific entropy — a hash of stable hardware identifiers combined with the hostname — together with that salt.

The practical consequence is what matters: copy the config file and the salt to a different machine and the result is undecryptable. Long-lived node credentials cannot be lifted off one box and replayed from another without a further compromise of the original machine. The node's TLS private key is stored the same way.

Ongoing authentication

Every heartbeat authenticates in one of two ways:

- **Long-lived node key**, presented in a header over TLS. The control plane hashes the presented key and compares against the stored hash.
- **OIDC machine identity.** For operators using service accounts, the node refreshes a short-lived bearer token ahead of each heartbeat. This is the path for operators who want no long-lived secret sitting on the node at all.

Mutual TLS can be layered on either, using an ECDSA P-256 client certificate that the node auto-renews when it comes within seven days of expiry.

A node whose heartbeat is rejected — for instance, after a control plane rebuild that lost state — re-registers automatically. This is the only automatic path back into the fleet, and it still requires an enrollment record matching that machine's entropy.

08 What an Edge Node Operator Can See

This is the chapter security reviewers turn to first, and the one where a white paper is most tempted to be vague. The claims below are stated at a level of detail that lets a customer check them, and an operator verify them on their own hardware.

8.1 On disk

The Edge Node binary keeps a single persistent directory. Inside it:

The content cache. Files named `<content-hash>.ee`, and nothing else. There is no manifest on disk mapping a hash to a customer, a product, a URL or a pricing tier. Without control plane credentials, which an operator does not have, a cached file cannot be attributed to a customer.

- *Level 0 workloads* contain Brotli-compressed blocks. An operator with the format specification can extract and read them. This is by design — Level 0 is the public tier, and those are the same bytes any end user downloads.
- *Level 1 and Level 2 workloads* contain AES-256-GCM ciphertext blocks and an encrypted index. Reading the file produces random-looking bytes. The keys are not on the machine: for Level 1 they exist in end-user browsers, and for Level 2 they are issued on demand and never written to disk.

The encrypted node config and its salt. The node's enrollment record — node ID, node API key, control plane URL — sealed with the machine-bound scheme in Chapter 7.

TLS material. The node certificate, which is public, and its private key, encrypted with the same machine-bound scheme. The CA chain used to verify the control plane, which is public information.

Pin state. A small file listing which workload hashes are pinned against cache eviction. Content hashes only, no customer metadata.

Logs. Operational events — heartbeats, fetches by hash, evictions, errors. Logs do not carry customer-identifying metadata for protected workloads.

8.2 In process memory

Level 0	Level 1	Level 2
Decompressed plaintext is in RAM during the serve. This is public content.	Nothing. The node handles only ciphertext; decryption happens in the end user's browser.	During an authorized request, the node holds the content key and decrypted block bytes in RAM. Both released when the response ends — released is not zeroed, and Section 9.5 states what that costs.

8.3 On the wire

An operator running a packet capture on their own node sees:

- **Inbound from end users:** HTTPS, terminating at the node. Hash-addressed request paths are visible at the HTTP layer after termination. For Level 1, the payload is ciphertext.
- **Upstream to the control plane:** TLS, optionally with mTLS on top. The control plane accepts only authenticated requests from enrolled nodes.
- **Upstream from object storage:** HTTPS. For Level 1 and Level 2, this is end-to-end ciphertext from the bucket to the operator's box.

So yes, an operator can capture their own traffic. For Level 1 and Level 2 what they capture is hash-named requests and ciphertext payloads.

8.4 The limit we do not design around

THE LOAD-BEARING LIMIT

Level 2 content exists in plaintext in RAM on an operator's machine for the duration of a request. An operator with root, a kernel module, eBPF, or a debugger attached to the node process can extract it.

No software-only defense stops that. Not on hardware the adversary owns. We are not going to claim a mitigation that does not exist, so here is what we actually have, and what it is worth:

What is cryptographic: nothing, at Level 2, against this specific attacker. The plaintext is in their address space.

What is procedural and operational:

- Operator participation is opt-in and gated by a written agreement.
- Every access token issuance is audit-logged by node ID, workload hash and timestamp. A node requesting an unusual number of distinct keys in a short window is visible on the control plane, and it is visible to the customer in their own audit feed, not only to us.
- Token TTL — 30 seconds by default, configurable downward — bounds how long a captured key remains usable.
- The window is one request, on whichever node that request happened to land.

What actually solves it: Level 1. A customer who considers operator compromise part of their threat model should not be running that content at Level 2. Level 1 exists precisely for this, and it removes the problem rather than narrowing it.

That recommendation is the honest one even though Level 2 is the more capable mode, and we would rather a customer land on the right level at design time than discover the distinction during a security review.

8.5 The same chapter, read from the other side

Everything above answers a customer's question: what can the person holding the machine see? Read it again as an operator and it answers a different one: what did this software put on my machine, and what is it doing?

The inventory is the same inventory. A cache directory of hash-named files the operator cannot read and did not choose. An encrypted credential file sealed to that specific machine. A certificate, a small pin-state file, and operational logs. A process that fetches, verifies, serves, and — for Level 2 requests the control plane authorizes — decrypts a stranger's content in the operator's memory for the length of one request.

Nothing in Sections 8.1 to 8.4 is withheld from the operator, and nothing in them is a capability the operator lacks. The sections are written at a level of detail that lets an operator verify each claim on their own hardware rather than take our word for it, which is the only form of assurance available to someone who was never promised one.

Two things belong to the operator alone and are not security properties at all:

- **Bandwidth and power.** The operator's real ongoing cost exposure. A commercial matter, settled in operator terms rather than in a file format, and outside this paper.
- **The ability to stop.** Whether and how quickly an operator can withdraw their machine is a property of the node software and the agreement, not a cryptographic one.

Both are answered in operator documentation rather than here, and they are named so that an operator reading this paper knows which of their questions it does not answer.

09 Security Analysis

Each claim below is stated, given its basis, and paired with what checks it.

9.1 Claim: ciphertext integrity

STATEMENT	An adversary cannot modify a `.ee` file without detection.
BASIS	AES-256-GCM is an AEAD. The authentication tag is verified before any plaintext is used. Separately, each block carries an xxhash64 check independent of the GCM tag.
TEST	A bit is flipped at every byte position in the file in turn. Every one is rejected.

9.2 Claim: Level 1 Edge Nodes cannot read content

STATEMENT	An adversary with root on an Edge Node serving a Level 1 workload learns nothing about the plaintext content itself. What such an adversary does still learn — approximate object size, request timing, and the fact that a given hash-addressed artifact is present — is set out in Sections 9.3 and 9.5.
BASIS	The index is encrypted with a key the node never receives. Note carefully what kind of property this is: it holds because no code path delivers the key to the node. It is a property of the deployment, not of the arithmetic.
TEST	A structural test walks the serve handler's abstract syntax tree and asserts that every call capable of obtaining key material sits inside a branch gated on the encryption level.

Structural rather than behavioral, on purpose. A behavioral test proves that today's request path is safe. This one fails on the *addition* of an ungated path — which is how this guarantee would actually be lost: silently, in a change that still serves correctly. It also fails if the watched calls are renamed away, so it cannot pass by protecting nothing.

9.3 Acknowledged leakage: object size correlates with content

This is a limitation, and it is stated as one.

Compression runs before encryption, so ciphertext length is a function of plaintext content. Two plaintexts of equal length can produce ciphertexts differing by more than an order of magnitude — measured at 27.5× in our own test suite. An observer

distinguishes them from size alone, without going anywhere near the encryption.

It is sometimes claimed that compression *enhances* security by increasing entropy or frustrating known-plaintext attacks. It does not. AES-GCM output is already indistinguishable from random; there is no entropy for compression to add. What compression adds is this side channel.

Consequence for linkability. Size, plus timing, is enough to group objects. Clustering on size alone recovers task grouping at 100% purity against a 17% baseline in our suite. A claim that files cannot be linked would be false, and we are not making one.

What is true. The per-build random secret and salt mean that the same plaintext deployed twice produces different ciphertext, a different hash and a different object name. Unlinkability holds *across deployments*. It does not hold *across nodes holding one artifact*, and content addressing is the reason — it is also what makes caching, deduplication and integrity work, so it is a deliberate trade rather than an oversight.

Mitigation. Length padding, as an explicit opt-in with a stated bandwidth cost. It is not a default and we are not claiming it until it ships.

9.4 Crypto-shredding — not forward secrecy

Each build generates its own secret. Deleting the stored secret renders every cached copy of that workload undecryptable on every node in the fleet, without having to reach any of them. The node cannot decrypt without asking the control plane, and the control plane then has nothing to give it.

This is a genuinely useful property for a distributed cache. It is the reason revocation does not require chasing copies.

It is not forward secrecy. Forward secrecy is a property of key agreement — ephemeral keys such that compromising a long-term key does not expose past sessions. Nothing here provides that. An adversary who captured a file earlier and later obtains the key can decrypt it. Content expiry is not forward secrecy either, and describing it that way would be wrong in two directions at once.

A caveat, and it is load-bearing. The control plane caches workload secrets in memory with no expiry, so deleting the stored secret does not take effect on a running instance until that instance restarts. Until that cache is bounded, crypto-shredding is a guarantee about *durable* storage, not about the live fleet. We are treating this as a defect to fix rather than a caveat to keep, and this paper will not describe the effect as immediate until it is.

9.5 Residual exposure, listed rather than omitted

- **Object size and request timing.** Section 9.3.
- **Level 2 key and plaintext in node memory** for the duration of a request. Released afterward, but released is not the same as zeroed: buffers return to the runtime's memory manager rather than being guaranteed overwritten, and the HTTP transport holds buffers we do not control at all. The exposure is narrowed to the request window, not eliminated within it.
- **Control plane compromise** exposes Level 2 content. Section 4.2.
- **Content-addressed names** let two nodes confirm they hold the same artifact.
- **Customer-side URL hygiene.** A Level 1 URL is only as protected as the page it appears on. If a customer serves it over plain HTTP, writes it into server-side logs or analytics, or lets it escape in a referrer header, the fragment carrying the key can leak. The design assumes HTTPS end to end on the customer's site, and the SDK documentation says so.
- **Side channels in third-party implementations** — timing, cache, power. We use mainline libraries and inherit both their guarantees and their limits.
- **Physical attacks** on operator hardware — cold-boot memory extraction, JTAG, bus probing. Out of scope for a software system, and in the same category as Section 8.4.

9.6 Verifying these claims without our cooperation

A customer evaluating the platform can check the following independently.

01 **Inspect a Level 1 download in browser DevTools.** The Network tab shows the Edge Node response as opaque block ciphertext, not your content. Decryption appears in the page, in a Web Crypto call.

02 **Confirm the fragment is never transmitted.** Inspect the outgoing request. Only the path before the **#** is on the wire.

03 **Hash-check a cache artifact.** Pull the raw `.ee` file from any Edge Node, hash the canonical content with BLAKE3, and compare against the in-header hash. A match means the bytes were not silently substituted.

04 **Replay-test enrollment.** Attempt to enroll a used token a second time. The control plane refuses it.

05 **Read your own audit log.** For Level 2 workloads, every access-token issuance for content you own appears in your portal audit feed with timestamp, node ID and outcome.

10 Choosing a Level

The level is a design-time decision and it is the most consequential one a customer makes on this platform. Each option below carries its trust statement from Section 4.3, so the choice can be made on the property rather than on the label.

Static and media delivery → Level 0

Integrity only

Marketing sites, documentation, static assets, public media. *Trust statement: integrity only. Content is public.* Choose this when the content would sit on a public CDN regardless.

Licensed, paywalled or per-user content → Level 1

Recommended

Premium downloads, per-customer assets, anything where the customer wants zero server-side trust. *Trust statement: confidential against the Edge Node and against the control plane, as long as AES-256-GCM holds.* Choose this whenever operator compromise or provider compromise is in your threat model. Accept that the node cannot address individual files inside the workload.

Per-request authorized delivery → Level 2

Read §8.4 first

Licensed media with subscription enforcement, paid downloads validated server-side, anything needing central revocation and a per-access audit trail. *Trust statement: confidential against anyone who cannot obtain a per-request token, and against the node outside the window of a request it is authorized to serve.* Choose this when you need revocation and audit more than you need protection against a privileged operator — and read Section 8.4 before you decide that.

> If the two requirements conflict — you need central revocation *and* you cannot accept transient plaintext on operator hardware — the honest answer today is that this platform does not resolve that conflict for you. Level 1 with revocation handled in your own application layer is the closest available shape.

II Limits We State Plainly

We trust the control plane. A compromised control plane breaks Level 2 confidentiality. It does not break Level 1. The control plane is operated by Evolving Edge with conventional cloud controls — SSO, audit logging, infrastructure as code, least privilege. Those are good practices, not a guarantee.

We trust each operator's local kernel. A technical operator with root on their own machine can extract Level 2 plaintext from RAM during a request. No software-only system prevents that. Section 8.4 covers the mitigations and their real weight.

We trust customer URL hygiene. Level 1 fragments are safe over HTTPS and only if customers keep them out of logs, analytics and referrer headers.

Crypto-shredding is not yet immediate on the live fleet. Section 9.4. Tracked as a fix.

No customer-managed keys yet. Level 2 key material lives with the control plane. There is no path today for a customer to hold Level 2 keys in their own KMS or HSM. Customers who need that separation should use Level 1, where the key never reaches us in the first place. Customer-managed keys for Level 2 are on the roadmap and are not claimed as a current capability.

No formal verification. The cryptographic code is reviewed and tested against shared cross-language vectors. It has not been formally verified in the sense of a machine-checked proof. We use mainline standard libraries specifically to minimize the surface where our own implementation errors could live.

No length padding today. Section 9.3 describes the side channel and names padding as the mitigation. It is not shipping and is not claimed.

No customer workload execution, and so nothing to protect at runtime. Edge Nodes cache and serve files. Evolving Edge does not yet run customer code on operator hardware, so there is no runtime-protection property to claim — not a weak one, not any. Encrypted micro-VM isolation is the intended answer for later roadmap phases and does not exist in production. Sections 8.4 and 9.5 together describe what a privileged operator can reach during a request, and for content delivery as it ships today, Level 1 is the only mode that removes that question rather than bounding it.

Compliance certifications are in progress, not held. SOC 2 Type II and HIPAA readiness work is underway. Neither is complete as of this writing, and nothing in this paper should be read as an attestation. Compliance is an audit outcome, not a property of a file format: what the `.ee` format provides is technical safeguards — encryption at rest and in transit, AEAD integrity, per-request access control, and auditable key custody — which are inputs to an audit rather than a substitute for one.

12 Conclusion

At Level 1, Evolving Edge runs customer content through operator-owned hardware without making operators trustees of plaintext.

For sensitive content, Level 1 is the default and the recommendation. The Edge Node, the control plane and the storage origin see only AES-256-GCM ciphertext bound to a key that exists in the end user's browser and nowhere else on the path. A control plane compromise does not reach it. An operator with root does not reach it.

Where a customer needs central revocation and a per-access audit trail, Level 2 provides them, and plaintext appears briefly in operator RAM during a request in exchange. That window is bounded by a 30-second default token TTL and recorded in an audit feed the customer can read directly. It is a real exposure and we would rather a customer weigh it than discover it.

Public content flows in the clear at Level 0, because that is the contract.

Long-lived operator credentials are sealed to the machine that enrolled them. Content is addressed by BLAKE3 hash and authenticated per block. Keys are derived per workload with no reuse across customers.

And where the trust model has limits — privileged kernel access on operator hardware being the load-bearing one — those are written down here, in the same document as the claims, so that a customer can pick the level that matches their threat model rather than the level that matches our marketing.

This is the distribution substrate, and it is the part of the architecture designed to survive the platform's move from content delivery toward inference and general compute. Which parts of it actually survive unchanged is set out in Section 3.3, along with the part that does not: the encryption levels as they stand assume a node that serves files rather than runs them. Keeping a workload unreadable while it runs is a different problem, no customer code runs on Edge Nodes today, and that problem will be answered in a different document when there is something running to answer it about.

Talk to us

Security teams evaluating the platform	Every verification step in Section 9.6 is available to you without our involvement. Start there.
Customers with a compliance requirement	Tell us the control framework you are being audited against and what evidence your auditor expects. That conversation is more useful than a certification badge.
Researchers	Vulnerabilities can be reported to security@evolvingedge.ai. We aim to acknowledge within one business day and will work with you on coordinated disclosure.

A Glossary

AEAD — Authenticated Encryption with Associated Data. An encryption mode providing confidentiality and integrity together, so tampering is detected on decryption.

AES-256-GCM — The block cipher and mode used for all content and index encryption in this system. An AEAD construction.

BLAKE3 — The cryptographic hash function used for content addressing.

Block — The unit of compression and encryption inside a `.ee` container. Block size is chosen by content type.

Content addressing — Naming an object by the hash of its content, so the name changes if the bytes change.

Control plane — The centrally operated Evolving Edge service that authenticates customers, issues credentials, brokers Level 2 access decisions and holds workload secrets. Holds no customer content.

Crypto-shredding — Rendering data undecryptable by destroying the key rather than the data. See Section 9.4.

Edge Node — The Evolving Edge caching and serving software, running on hardware owned by an independent operator.

Encryption level — A per-workload property, fixed at packaging time, determining key custody. Level 0, 1 or 2.

HKDF-SHA256 — The key derivation function used to expand a workload secret into master, index, content and nonce material.

Index — The structure inside a `.ee` container describing the files it holds: path, offset, size, MIME type and block range. Encrypted at Level 1 and Level 2.

Operator — The independent individual or business running an Edge Node on their own hardware.

PBKDF2-SHA256 — The deliberately slow key derivation function used to stretch machine entropy into the key that seals a node's on-disk credentials.

Per-request access token — Short-lived credential issued by the control plane authorizing a single Level 2 decryption. 30-second default TTL.

URL fragment — The portion of a URL after **#**. Never transmitted to a server by a compliant client. Carries the Level 1 key.

Micro-VM — A lightweight virtual machine. Encrypted micro-VMs are the intended basis for keeping a workload unreadable while it runs, in later roadmap phases. Not implemented, not designed, not tested; see Section 3.3.

Workload — One packaged, content-addressed, independently keyed unit: a single `.ee` file. Throughout this paper a workload is a *file*, not a running process. Its contents today are customer content; the container format itself is not specific to content.

xxhash64 — A fast non-cryptographic checksum applied per block, independent of the GCM authentication tag.

B Format field reference

FIELD	REGION	LEVEL 0	LEVEL 1	LEVEL 2
Magic / version	Header, cleartext	Present	Present	Present
Encryption level	Header, cleartext	Present	Present	Present
`contentHash` (BLAKE3)	Header, cleartext	Present	Present	Present
`indexOffset`	Header, cleartext	Present	Present	Present
`indexSize`	Header, cleartext	Present	Present	Present
`indexNonce`	Header, cleartext	Present	Present	Present
`indexAuthTag`	Header, cleartext	Present	Present	Present
`totalCompressedSize`	Header, cleartext	Present	Present	Present
`totalUncompressedSize`	Header, cleartext	Present	**Zeroed**	**Zeroed**
`blockCount`	Header, cleartext	Present	**Zeroed**	**Zeroed**
`fileCount`	Header, cleartext	Present	**Zeroed**	**Zeroed**
File entries (path, offset, size, MIME, block range)	Index	Cleartext	AES-256-GCM	AES-256-GCM
Content blocks	Blocks	Brotli only	Brotli + AES-256-GCM	Brotli + AES-256-GCM
Per-block xxhash64	Blocks	Present	Present	Present

Header is a fixed 512 bytes. Values withheld at Level 1 and Level 2 are carried inside the encrypted index.

C References

CRYPTOGRAPHIC STANDARDS

- NIST SP 800-38D — *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*
- FIPS 197 — *Advanced Encryption Standard (AES)*
- RFC 5869 — *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*
- RFC 8018 — *PKCS #5: Password-Based Cryptography Specification Version 2.1* (PBKDF2)
- RFC 7932 — *Brotli Compressed Data Format*
- RFC 3986 §3.5 — *Uniform Resource Identifier (URI): Generic Syntax*, fragment identifier semantics
- RFC 8446 — *The Transport Layer Security (TLS) Protocol Version 1.3*
- RFC 5246 — *The Transport Layer Security (TLS) Protocol Version 1.2*
- BLAKE3 specification — O'Connor, Aumasson, Neves, Wilcox-O'Hearn

IMPLEMENTATION

- Go standard library `crypto/aes`, `crypto/cipher`, `crypto/tls`
- W3C Web Cryptography API

Comparative claims in Section 2.4 are stated qualitatively and without figures. Any comparison added to that section — numeric or not — requires a citation before publication.

CLASSIFICATION	Public
STATUS	Describes the shipping system. Section 9.4 identifies one defect under active remediation and labels it as such. Section 3.3 states the roadmap boundary: Evolving Edge does not execute customer code on Edge Nodes, and this paper covers workload distribution only.
SECURITY CONTACT	security@evolvingedge.ai