Onion Routing for the Unreachable: A Portable Tor Implementation

Anonymous Author(s)

Abstract

Tor has protected millions from surveillance for two decades, but using it requires external software—a barrier that excludes web applications wanting to offer Tor-level privacy without user installation. We present **tor-wasm**, the first Tor implementation that runs natively in web browsers. The key obstacle is architectural: browsers forbid TCP sockets, but Tor relays speak only TCP. We overcome this with a bridge server that translates WebSocket to TCP, designed so that existing Tor relay volunteers can operate bridges with *strictly less* visibility than guard relays—introducing no new trust beyond what Tor users already accept. Building tor-wasm required solving browser-specific challenges: TLS validation without system time, cryptographic randomness without OS access, and async coordination in single-threaded WebAssembly. The result is a 340KB module (72KB minimal core) that builds standard 3-hop circuits through the production Tor network. Beyond browsers, this compact footprint enables deployment on resource-constrained embedded systems—addressing a gap in medical device cybersecurity where TLS protects content but network metadata remains exposed. We release tor-wasm as open-source software.

Keywords

Tor, WebAssembly, Privacy, Anonymity, Browser Security, Medical Device Cybersecurity, LLM Privacy

1 Introduction

The web browser has become the dominant platform for interactive computing, yet it remains one of the few environments where Tor—the most widely deployed anonymity network—cannot run. This is not for lack of demand. Journalists need to receive anonymous tips through web forms. Cryptocurrency applications need to broadcast transactions without linking wallet addresses to IP addresses. Telehealth platforms need to let patients research sensitive conditions without creating logs that could be subpoenaed or breached. In each case, the application developer could solve the problem if only they could route specific requests through Tor—but the browser forbids raw socket access, the Tor network speaks only TCP, and existing embeddable Tor libraries require operating system primitives that browsers intentionally omit.

The standard solution is to ask users to install the Tor Browser, a modified Firefox that routes all traffic through Tor. This works well for users who want comprehensive anonymity, but it creates a fundamental adoption barrier: users must download and configure external software, and once they do, the choice becomes all-or-nothing. Either your entire browsing session is anonymous (with the associated latency and compatibility costs) or none of it is. For applications that need to anonymize only specific sensitive operations—a transaction broadcast, a tip submission, a medical query—there is currently no good option.

Why this matters now. Consider a pulse oximeter that uploads readings to a cloud dashboard. The device connects from the patient's home network, revealing their IP address to the telemetry endpoint. An attacker who compromises that endpoint—or an insider with database access—can correlate device types with patient IP addresses, geolocate patients to their homes, and infer medical conditions from the devices they use. The FDA's June 2025 cybersecurity guidance [26] now mandates protection of "any data whose disclosure could lead to patient harm," and network metadata plainly qualifies. Yet no existing Tor implementation can run on the resource-constrained microcontrollers in medical devices, and patients cannot be expected to configure Tor on their home routers. Similar gaps exist for domestic violence survivors researching legal resources (their IP addresses logged by every site they visit), cryptocurrency users broadcasting transactions (linking wallet addresses to physical locations), and employees using corporate AI tools (their prompts revealing strategic intentions to cloud providers). In each case, metadata exposure creates concrete harm that content encryption alone cannot prevent.

The Tor network itself serves approximately 2 million daily users through over 7,000 volunteer-operated relays [4]. This infrastructure represents two decades of engineering and community building. The question we address is not whether anonymous communication is valuable—Tor's continued growth answers that—but whether anonymous communication can be made available *natively* in the web platform, without external software installation. **We show that it can.**

1.1 The Challenge

Running Tor in a browser is not simply a matter of porting existing code to WebAssembly. The browser security model fundamentally conflicts with Tor's requirements in ways that no existing implementation has solved.

No TCP sockets. Tor relays communicate over TCP, but browsers permit only HTTP, WebSocket, and WebRTC connections. This restriction exists for security reasons—allowing arbitrary TCP connections would enable malicious JavaScript to port-scan internal networks—but it means every existing Tor implementation is architecturally incompatible with browsers. We cannot work around this with a library change; we need a fundamentally different network architecture.

Proceedings on Privacy Enhancing Technologies 2026(4), 1–17

https://doi.org/XXXXXXX.XXXXXXX

No system time. TLS certificate validation requires comparing the current time against certificate validity periods. WebAssembly modules execute in a deterministic sandbox where `std::time::SystemTime::now()` panics because there is no "system" to query. Every TLS library we examined assumes time access works.

No OS randomness. Cryptographic operations require secure random numbers. The standard approach—reading from `/dev/urandom` or calling `getrandom()`—is impossible from the WASM sandbox, which has no access to operating system resources.

Stale keys. Tor relays rotate their ntor keys periodically, and the network consensus updates hourly. Clients with cached or stale consensus data may attempt handshakes with outdated keys, producing protocol errors that must be handled gracefully without revealing information to adversaries.

Each obstacle required a solution that preserved Tor's security guarantees while working within the browser's constraints. Section 4 documents our solutions; Section 5 analyzes their security implications.

1.2 Contributions

(1) **The first browser-native Tor client.** We present tor-wasm, a 17,000-line Rust implementation compiled to WebAssembly that builds standard 3-hop circuits through the production Tor network. The 340KB module (72KB minimal core) is small enough for web applications and embedded systems—platforms that existing implementations like Arti cannot reach.

(2) **A trust-preserving bridge architecture.** We introduce a WebSocket-to-TCP bridge that enables sandboxed clients to reach Tor relays. We prove that bridge operators observe *strictly less* than guard relay operators, meaning tor-wasm requires no trust beyond what Tor users already accept.

(3) **Browser-specific engineering solutions.** We document techniques for TLS validation without system time, cryptographic randomness via WebCrypto, and cooperative async scheduling for single-threaded WASM—solutions that may benefit other privacy tools targeting browsers.

(4) **A new application domain: metadata-anonymous medical telemetry.** We identify that current device threat models protect content via TLS but leave network metadata exposed. We show how tor-wasm's compact footprint enables FDA-compliant anonymization on resource-constrained medical devices, and demonstrate a working pulse oximeter prototype.

2 Background

This section provides background on the Tor protocol and explains why browsers make implementation difficult. Readers familiar with Tor may skip to Section 3.

2.1 The Tor Protocol

Tor achieves anonymity through *onion routing* [4, 19]. When a client wants to reach a destination server, it doesn't connect directly. Instead, it builds a *circuit* through three relays, each operated by a different volunteer:

Client → Guard → Middle → Exit → Destination

The three relays serve distinct roles. The **guard** (or entry) relay is the client's first hop into the Tor network. Clients maintain stable guard relationships over weeks or months to limit the number of relays that observe their entry traffic. The **middle** relay provides separation between entry and exit. The **exit** relay is the last hop that communicates with the destination server on the client's behalf.

This three-hop design ensures that no single entity sees the full picture:

- The **guard** knows who you are (your IP address), but not where you're going (the destination is encrypted)
- The **middle** knows neither who you are nor where you're going (both are hidden by encryption)
- The **exit** knows where you're going (it communicates with the destination), but not who you are (it sees only the middle relay's IP)
- The **destination** sees traffic from the exit's IP address, not the client's

This property—that an adversary must control both the guard *and* the exit to correlate traffic—is fundamental to Tor's security model.

2.2 The ntor Handshake

The client establishes a separate encryption key with each relay using the *ntor* handshake [12], which provides authenticated key exchange based on Curve25519. Each relay publishes two pieces of cryptographic material in the network consensus:

- An **identity fingerprint**: the SHA-1 hash of the relay's RSA public key, serving as a stable identifier
- An **ntor public key**: a Curve25519 public key $B = b \cdot G$ used for ephemeral key exchange

When the client wants to establish a hop with relay R, it:

(1) Generates an ephemeral Curve25519 keypair $(x, X = x \cdot G)$
(2) Sends a CREATE2 cell containing the relay's identity, ntor key B, and client's ephemeral X
(3) Receives a CREATED2 cell containing the relay's ephemeral Y and an authentication tag
(4) Computes shared secrets $x \cdot Y$ and $x \cdot B$, derives session keys via HKDF-SHA256
(5) Verifies the authentication tag to confirm the relay possesses the private key for B

This handshake provides forward secrecy (compromise of long-term keys does not reveal past session keys) and authentication (the client knows it's talking to the intended relay, not an impersonator).

2.3 Onion Encryption

After establishing keys with all three relays, data is encrypted in layers—first with the exit's key, then the middle's, then the guard's.

Each relay decrypts one layer and forwards the result. This "onion" structure gives Tor its name. A RELAY cell payload consists of a 1-byte relay command, 2-byte "recognized" field (must be zero), 2-byte stream ID, 4-byte running digest, 2-byte payload length, and up to 498 bytes of data. The running digest provides integrity verification: each hop maintains a SHA-1 hash state updated with each cell, and tampering is detected because modified cells have incorrect digests.

2.4 Why Browsers Make This Difficult

The browser security model fundamentally conflicts with Tor's requirements in several ways. First, Tor relays communicate over TCP using TLS for transport encryption, expecting a reliable ordered byte stream, but browsers provide only higher-level APIs (`fetch()`, `WebSocket`, `RTCDataChannel`) with raw socket access explicitly forbidden by the W3C Web Platform Design Principles to prevent port-scanning attacks. Second, Tor clients download a "consensus" document from directory authorities—a multi-megabyte file listing all relays and their cryptographic keys—but directory authorities do not serve CORS headers (they predate modern web security), so browsers block these downloads when initiated from a web page origin. Third, TLS certificate validation requires comparing timestamps against the current time, but WebAssembly provides no time access by design since it's a pure computation model where all inputs must be explicitly provided (Rust's `std::time::SystemTime` panics when compiled to `wasm32-unknown-unknown`). Fourth, cryptographic operations require secure random numbers obtained via system calls, but WebAssembly runs in a sandbox with no ambient authority and cannot make system calls. These aren't implementation inconveniences; they're fundamental conflicts between the browser security model and Tor's architecture, and solving them required introducing new components while carefully analyzing their security implications.

3 Architecture

Our architecture introduces a *bridge server* to overcome the browser's network limitations. The bridge is a simple WebSocket-to-TCP proxy, intentionally minimal in functionality to limit its security-relevant surface. The key insight enabling our trust model is that bridge operators need not be trusted any more than guard relay operators—and can be the same people.

3.1 System Overview

Figure 1 illustrates the tor-wasm architecture. The browser runs our WASM module, which implements the full Tor protocol. The WASM module connects to a bridge server via WebSocket. The bridge server translates WebSocket frames to TCP connections with Tor relays. From the relay's perspective, the connection appears to come from the bridge server, not the browser.

3.2 The Bridge Server

The bridge server is intentionally minimal: approximately 100 lines of Node.js that accepts WebSocket connections and proxies them to TCP. The client specifies the target relay in the WebSocket URL:

```
wss://bridge.example.com?addr=192.0.2.1:9001
```

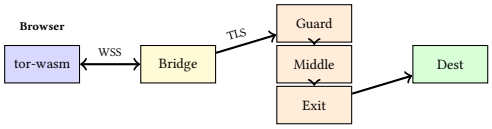

Figure 1: tor-wasm architecture. The bridge translates WebSocket to TCP but cannot decrypt Tor traffic—the TLS session is end-to-end between browser and guard.

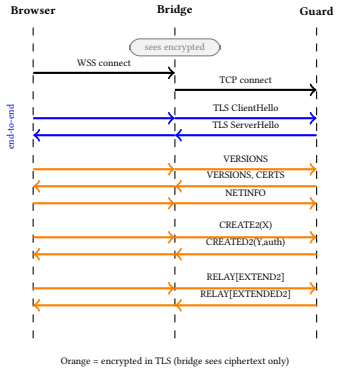

Figure 2: Circuit establishment protocol. The TLS session (blue) is end-to-end between browser and guard. All Tor messages (orange) are encrypted inside TLS—the bridge relays bytes it cannot interpret.

When a connection arrives, the bridge:

(1) Parses the `addr` parameter to determine the target relay
(2) Opens a TCP connection to that relay
(3) Relays bytes bidirectionally between the WebSocket and TCP socket
(4) Closes both connections when either side disconnects

The bridge performs *protocol translation*, not *traffic inspection*. It does not interpret Tor cells or maintain any state about circuit construction. It cannot decrypt traffic because the TLS session is established end-to-end between the browser's WASM module and the Tor relay—the bridge sees only opaque TLS records. Figure 2 illustrates the message flow during circuit establishment, showing exactly what the bridge observes at each step.

3.3 Leveraging Existing Tor Infrastructure

A critical design decision was *who would operate bridge servers*. A naive approach would require building a new network of bridge operators, raising questions about trust, sustainability, and adoption. We took a different approach: we designed bridges to be operated by the *existing Tor community*—the approximately 2,000 volunteer-operated bridges and 7,000 relays whose operators have already demonstrated commitment to privacy infrastructure, often running relays for years. Our bridge is a *companion service* that existing operators can run alongside their Tor infrastructure with a one-command installation (`curl -fsSL https://[url]/install.sh | bash`) that deploys a containerized bridge server on a separate port. The bridge has no special privileges and cannot affect the

operator's existing Tor relay; it simply provides an additional WebSocket entry point to the same network. This matters because users of tor-wasm do not need to trust a new set of operators—they are trusting the same community that already provides Tor infrastructure. As we analyze in Section 5, a bridge operator has *less* visibility into user traffic than a guard relay operator, the very relationship Tor users already accept.

3.4 Bridge Discovery and Selection

Users can obtain bridge addresses through several mechanisms:

- **Public directory**: We maintain a list of known bridges, similar to how Tor publishes relay information. Applications can query this directory at initialization.
- **Application-bundled**: Web applications can ship with a set of known bridges, providing out-of-box functionality without external queries.
- **Self-hosted**: Organizations can run their own bridges, providing complete control over the trust relationship. Bridge deployment is trivial (single Docker container).
- **User-configured**: Power users can specify custom bridge URLs, enabling use with private or experimental bridges.

The client library supports configuring multiple bridge URLs. If a connection fails (network error, bridge overload, etc.), the client automatically fails over to alternatives. Users can also rotate bridges between circuits to limit any single operator's visibility.

3.5 Consensus Distribution

In addition to protocol translation, bridges solve the consensus distribution problem. The Tor consensus document (listing all relays and their keys) is several megabytes and served without CORS headers. Browsers cannot fetch it directly.

Our bridge servers fetch and cache the consensus, serving it to browser clients over the WebSocket connection. This adds a trust consideration: a malicious bridge could serve a poisoned consensus, directing clients to adversary-controlled relays. We mitigate this through:

- **Signature verification**: The consensus is signed by directory authorities. We verify these signatures in the WASM module before trusting the relay list.
- **Multiple bridges**: If using multiple bridges, the client can cross-check consensus documents for consistency.
- **Freshness checks**: We verify the consensus is not stale (within its validity period) and reject obviously outdated documents.

Table 1 details our consensus freshness validation behavior. Tor consensus documents include `valid-after`, `fresh-until`, and `valid-until` timestamps. We enforce a maximum validity period of approximately 3 hours, rejecting consensus documents beyond this threshold to prevent route manipulation attacks where an adversary replays old consensus data to direct clients to specific relays.

Table 1: Consensus freshness validation behavior. tor-wasm rejects stale consensus documents to prevent route manipulation.

Age	Accepted	Behavior
0 hours	✓	Normal operation
1 hour	✓	Fresh-until warning, refetch recommended
2 hours	✓	Stale warning, use cached
3 hours	×	At valid-until, reject
6+ hours	×	Beyond validity, must refetch

3.6 The WASM Module

The core of tor-wasm comprises approximately 17,000 lines of Rust compiled to WebAssembly. Rather than wrapping an existing Tor implementation, we reimplemented the protocol from the specification—a deliberate choice that enabled the architectural flexibility needed for browser deployment while keeping the binary compact.

The protocol layer handles Tor's cell-based communication: parsing and constructing the ten cell types required for circuit operation (VERSIONS through PADDING), performing the ntor Curve25519 handshake for authenticated key exchange, and maintaining the per-hop AES-128-CTR encryption state that implements onion routing. Each relay cell carries a running SHA-1 digest that we update incrementally, enabling integrity verification without buffering entire streams.

Above the protocol layer, the circuit manager builds 3-hop paths through the Tor network with automatic retry logic for transient failures. Circuits are isolated by destination—requests to different domains use different circuits—and rotate after 10 minutes or 100 requests per the Tor specification. Guard relay selection follows Tor's persistence model: we select guards using bandwidth-weighted probabilities, maintain relationships for 60 days, and persist selections to browser localStorage so that users don't rotate guards on every page load (which would paradoxically weaken anonymity).

Two features distinguish tor-wasm from minimal Tor implementations. First, we implement channel padding per `padding-spec.txt`, sending CELL_PADDING at randomized intervals during idle periods to resist ISP netflow analysis that would otherwise fingerprint Tor connections by their characteristic silence patterns. Second, we implement RTT-based congestion control using the Tor-Vegas algorithm, dynamically adjusting our sending window based on measured round-trip times. This breaks Tor's historical ~500KB/s per-stream speed limit, enabling bandwidth-intensive applications like LLM API streaming.

The compiled module is 340KB uncompressed (118KB gzipped)—larger than a minimal implementation would be, but comparable to React (130KB) and small relative to typical web application bundles.

4 Implementation

Building tor-wasm required solving several problems unique to the browser environment. We document these solutions to assist others building privacy tools for the web platform.

4.1 TLS Without System Time

TLS certificate validation requires checking that the current time falls between the certificate's "not before" and "not after" timestamps. Standard TLS libraries call operating system APIs to obtain the current time—in Rust, `std::time::SystemTime::now()`. This call panics when compiled to `wasm32-unknown-unknown`.

Our initial (insecure) approach was to disable time validation entirely, accepting any certificate regardless of validity period. This would allow man-in-the-middle attacks using expired or not-yet-valid certificates.

The solution came from recognizing that while WebAssembly itself has no time access, JavaScript does. The `Date.now()` function returns the current timestamp in milliseconds since the Unix epoch. Through the `wasm-bindgen` foreign function interface, Rust code compiled to WASM can call JavaScript functions.

We implemented a custom `TimeProvider` for the rustls library:

```
struct WasmTimeProvider;

impl rustls::client::danger::TimeProvider for
    WasmTimeProvider {
    fn now(&self) -> UnixTime {
        // js_sys::Date::now() returns
            milliseconds since epoch
        let millis = js_sys::Date::now() as u64;
        UnixTime::since_unix_epoch(Duration::
            from_millis(millis))
    }
}
```

This required using rustls version 0.23 with the new `TimeProvider` trait, and the `rustls-pki-types` crate with the `web` feature enabled, which provides WASM-compatible timestamp types.

Security analysis: This approach trusts the browser's JavaScript engine to provide accurate time. A malicious browser extension could manipulate `Date.now()`, but such an extension would already have full access to page content (including any data we're trying to protect). The threat model for browser-based security assumes a non-compromised browser.

4.2 Cryptographic Randomness

Cryptographic operations require secure random numbers. In Tor's ntor handshake, the client generates an ephemeral Curve25519 private key—if this key is predictable, an adversary can compute the session keys.

Standard Rust code uses `OsRng` from the `rand` crate, which calls operating system random number APIs. WebAssembly cannot make these calls.

Modern browsers provide cryptographic randomness through the WebCrypto API: `crypto.getRandomValues()` fills a buffer with secure random bytes. The challenge was connecting this browser API to Rust cryptographic libraries.

The `ring` cryptographic library (version 0.17 and later) provides a solution through the `wasm32_unknown_unknown_js` feature. When enabled, ring's random number generator calls WebCrypto instead of OS APIs:

```
# In Cargo.toml
[dependencies]
ring = { version = "0.17", features = ["
    wasm32_unknown_unknown_js"] }
```

This feature was added to ring relatively recently. Earlier attempts at browser cryptography would have required reimplementing primitives or using JavaScript cryptographic libraries through FFI.

Security analysis: WebCrypto's `getRandomValues()` is specified to use a cryptographically secure pseudorandom number generator (CSPRNG) seeded by the operating system. All major browsers implement this correctly. The security of our random numbers is thus equivalent to native applications.

4.3 Handling Stale Keys

Tor relays rotate their ntor keys periodically (typically every 7 days), and the network consensus is updated hourly. If a bridge's cached consensus is stale, the client may attempt handshakes with outdated keys.

When this happens, the relay cannot decrypt the CREATE2 cell (which is encrypted to the relay's ntor public key). It responds with a DESTROY cell with reason code 1: "protocol error." Our initial implementation treated this as a fatal error.

The fix was implementing robust retry logic. When circuit building fails, we:

(1) Categorize the error as **retryable** (stale keys, timeout, connection refused) or **fatal** (invalid consensus, cryptographic failure)
(2) For retryable errors, select a different guard relay and try again
(3) Continue until success or exhausting a configurable number of attempts (default: 20)

Guard selection mixes strategies for diversity: half the candidates are selected by bandwidth weight (favoring reliable relays) and half are selected randomly (avoiding concentration on a few popular relays).

In testing with the live Tor network, circuits typically succeed within 1–3 attempts. The retry mechanism also handles transient network issues, relay overload, and temporary unavailability.

Table 2 summarizes our failure mode analysis and graceful degradation behavior. We categorize failures by severity: fatal errors (requiring user intervention), retryable errors (automatic recovery), and transient errors (transparent retry). This comprehensive error handling ensures robust operation even under adverse network conditions.

4.4 Maintaining Cipher State

A subtle but critical detail: AES-CTR cipher state must persist across cells.

Tor uses AES-128 in counter mode (CTR) for stream encryption. Each hop has separate forward and backward ciphers, initialized

Table 2: Failure mode analysis and recovery behavior. tor-wasm implements automatic retry for transient failures.

Failure Mode	Detection	Recovery
Bridge unreachable	<1s	Fatal (user action)
Guard relay down	~5s	Auto-retry (new guard)
Middle relay down	~5s	Auto-retry (new middle)
Exit relay down	~5s	Circuit rebuild
Circuit timeout	30s	Full rebuild
TLS cert invalid	<100ms	New guard selection
ntor handshake fail	<500ms	New guard selection
Consensus invalid	<100ms	Fatal (security)
WebSocket disconnect	<100ms	Requires bootstrap

with keys derived from the ntor handshake. The counter increments with each block encrypted.

Our initial implementation created a new cipher object for each cell, resetting the counter to zero. This is catastrophic: if two cells use the same counter value, an adversary can XOR the ciphertexts to obtain the XOR of the plaintexts, revealing information about both.

The fix was maintaining persistent cipher state per hop:

```
struct HopState {
    forward_cipher: Aes128Ctr,  // Counter
        persists across cells
    backward_cipher: Aes128Ctr,
    forward_digest: Sha1,       // Running hash
        state
    backward_digest: Sha1,
}
```

The cipher objects are never re-created; they're reused for the lifetime of the circuit. Each encryption operation advances the counter, ensuring no counter reuse.

4.5 Cooperative Scheduling for Single-Threaded WASM

A subtle but critical challenge emerged in the browser's single-threaded execution model. Rust's standard approach to interior mutability—`RefCell`—panics if code attempts to borrow mutably while already borrowed. In async Rust, borrows that span `.await` points can cause this panic when other code attempts to borrow during the suspension.

Our initial implementation suffered from this "borrow-across-await" bug: circuit operations would hang indefinitely because the circuit was borrowed when async I/O began and remained borrowed when the runtime tried to resume. This pattern is particularly insidious because it works in multi-threaded environments (where `Mutex` handles blocking correctly) but fails silently in single-threaded WASM.

We solved this with a novel "checkout/return" pattern. Instead of holding borrows across await points, the scheduler briefly borrows the circuit to check out ownership, releases the borrow, performs async I/O with the owned circuit, then briefly borrows again to return it:

```
// WRONG: borrow spans await - will hang in WASM
{
    let mut circuit = scheduler.borrow_mut();
    circuit.send_cell(&cell).await;  // Borrow
        held!
}

// CORRECT: brief borrows only
let circuit = { scheduler.borrow_mut().checkout()
    };  // Released!
circuit.send_cell(&cell).await;  // No borrow held
{ scheduler.borrow_mut().return_circuit(circuit)
    };
```

This cooperative scheduling architecture enabled all async operations (circuit building, stream establishment, HTTP requests) to work correctly in the browser's single-threaded environment. The pattern is general and may benefit other async Rust code targeting WASM.

4.6 Digest Computation Scope

The running digest provides integrity verification for RELAY cells. Our initial implementation computed the digest over only the meaningful data portion (the bytes specified by the length field). This produced incorrect digests, causing relays to reject our cells.

The Tor specification states that the digest covers the "entire relay payload"—all 509 bytes, including padding. The fix was ensuring we hash the full payload:

```
// Construct payload with digest field zeroed
let mut payload = [0u8; 509];
payload[0] = relay_command;
// ... fill in other fields ...
payload[5..9].copy_from_slice(&[0, 0, 0, 0]);  //
    Zero digest field

// Hash full 509 bytes, not just data portion
self.forward_digest.update(&payload);

// Insert truncated digest
let digest = self.forward_digest.clone().finalize
    ();
payload[5..9].copy_from_slice(&digest[..4]);
```

This exemplifies a general principle: implementing cryptographic protocols from specifications requires extreme attention to byte-level details. The specification's language ("entire relay payload") was technically correct but easy to misinterpret.

4.7 Circuit Isolation and Rotation

Tor's security model requires circuit isolation to prevent cross-site correlation attacks. Without isolation, an adversary controlling two websites could observe that requests to both sites arrived from the same circuit—and thus the same user—because they exit from the same relay at the same time.

We implement configurable isolation policies ranging from per-domain (the default, where each domain receives its own circuit) to per-request (maximum unlinkability at the cost of latency). The per-domain policy suffices for most applications: requests to `example.com` use a different circuit than requests to `other.com`,

preventing cross-site linkage while amortizing circuit build costs across multiple requests to the same origin.

Beyond isolation, circuits require *rotation* to limit the window during which an adversary can observe traffic patterns. We implement dual rotation triggers per the Tor specification: circuits older than 10 minutes are retired regardless of activity, and circuits that have served 100 requests are retired regardless of age. The circuit cache tracks creation timestamps and request counts, transparently building replacement circuits when thresholds are exceeded. This ensures that even long-running applications do not maintain observation windows that would aid traffic analysis.

4.8 Guard Persistence

Tor's guard selection algorithm limits adversarial observation opportunities by maintaining stable entry point relationships. Frequent guard rotation is counterintuitively *less* secure: if a client randomly selects guards for each session, a malicious relay operator can simply wait until they are selected, eventually observing all users. Persistent guards limit this exposure to users who happened to select the adversary's relay initially.

We implement guard persistence per the Tor path-spec. Guards are selected using bandwidth-weighted probabilities and maintained for 60 days before rotation, matching Tor's recommended range. Guards that fail five consecutive connection attempts are temporarily marked unavailable for one hour to prevent persistent selection of unreachable relays. The critical browser-specific challenge is persistence across page loads: we serialize guard state—selected fingerprints, rotation timestamps, and failure counts—to browser localStorage.

On bootstrap, the client loads persisted state and evaluates whether refresh is needed. If the rotation timestamp has passed or too few usable guards remain, new guards are selected from the fresh consensus; otherwise, existing guards are reused. This mechanism is particularly important in the browser context, where selecting new random guards on every page load would significantly weaken the security properties that guard persistence provides.

4.9 Channel Padding for Netflow Resistance

ISPs deploy routers that create Netflow/IPFIX records for every connection, logging source IP, destination IP, timestamps, packet counts, and byte counts. These records can be correlated to deanonymize Tor users based on their distinctive connection patterns—Tor connections have characteristic idle periods that differ from normal web traffic.

We implement channel padding per the Tor `padding-spec.txt` specification to resist this analysis. Our `PaddingScheduler` component manages CELL_PADDING transmission during idle periods:

- **Randomized intervals**: Padding cells are sent at random intervals between 1.5 and 9.5 seconds (configurable), preventing adversaries from fingerprinting connections based on predictable timing patterns.
- **Activity-aware scheduling**: The scheduler tracks the last real cell timestamp. Padding is only sent during idle periods; when real traffic flows, padding is suppressed to avoid unnecessary overhead.
- **Idle timeout**: After 30 seconds of complete inactivity, padding stops entirely. This prevents indefinite resource consumption on abandoned connections.
- **Negotiation support**: We implement PADDING_NEGOTIATE cells to request padding from relays with specific parameters, and handle PADDING_NEGOTIATED responses to respect relay preferences.

The CELL_PADDING cell uses circuit ID 0 (link-level) and contains random or zero-filled payload data that relays ignore. This is distinct from circuit padding (Proposal 254), which operates at the circuit level with finite state machines—we defer circuit padding as noted in Section 9.

4.10 RTT-Based Congestion Control

Legacy Tor uses fixed SENDME windows (500 cells per stream), limiting throughput to approximately:

$$\text{max\_speed} = \frac{500 \times 498 \text{ bytes}}{\text{RTT}} \approx 500\text{KB/s for typical RTTs}$$

We implement Proposal 324's RTT-based congestion control using the Tor-Vegas algorithm, enabling dynamic window adjustment that breaks this limit. Our `CongestionController` provides:

- **RTT estimation**: An exponential weighted moving average (EWMA) estimator tracks smoothed RTT and RTT variance from SENDME round-trip measurements. We maintain minimum observed RTT as an estimate of base RTT without queuing.
- **Dynamic congestion window**: The window (CWND) ranges from 31 to 10,000 cells, starting at 31 with initial slow-start threshold of 10,000. The window grows during slow-start and adjusts based on queue occupancy during congestion avoidance.
- **Tor-Vegas algorithm**: We estimate queue occupancy as $\text{diff} = \text{CWND} \times (\text{SRTT} - \text{min\_RTT})/\text{SRTT}$. If $\text{diff} < \alpha$ (3 cells), we increase CWND; if $\text{diff} > \beta$ (6 cells), we decrease CWND. This maintains optimal throughput without excessive queuing.
- **Slow-start exit**: When queue occupancy exceeds γ (3 cells), we exit slow-start and enter congestion avoidance, setting SSTHRESH to the current window.

For applications that don't benefit from dynamic windows (e.g., low-bandwidth telemetry), we support falling back to the legacy fixed algorithm by setting `CongestionAlgorithm::Fixed`.

5 Security Analysis

The introduction of a bridge component raises natural questions about security. Does the bridge weaken Tor's anonymity guarantees? What can a malicious bridge operator learn? In this section, we provide a formal threat model and demonstrate that bridges are strictly less privileged than guard relays.

5.1 Threat Model

We consider four classes of adversaries, following standard Tor threat modeling [4, 9]. A **network observer** ($\mathcal{A}_{\text{net}}$) is a passive adversary who can observe traffic on some network links (e.g.,

user's ISP, coffee shop WiFi, compromised router) and sees encrypted traffic between the client and bridge. A **malicious bridge** ($\mathcal{A}_{\text{bridge}}$) operates one or more bridge servers and can log all connections, perform timing analysis, selectively deny service, and serve manipulated consensus documents, but cannot decrypt Tor cells. A **malicious relay** ($\mathcal{A}_{\text{relay}}$) operates one or more Tor relays—the standard Tor threat model applies where controlling both guard and exit enables correlation attacks, and our bridge architecture does not change this. A **global passive adversary** ($\mathcal{A}_{\text{global}}$) can observe all network traffic worldwide; neither Tor nor tor-wasm claims to defend against this adversary since end-to-end timing correlation is fundamentally possible [13, 14].

5.2 Bridge Trust Analysis

The introduction of bridge servers raises a natural question: doesn't this add a new trusted party? We argue the additional trust is minimal for three reasons. First, the bridge sees less than a guard relay: a Tor guard relay knows the client's IP address and receives Tor cells that it decrypts to process commands, seeing plaintext cell headers including circuit IDs, while a tor-wasm bridge knows the client's IP but receives only TLS-encrypted traffic (the TLS session is end-to-end between browser and guard), meaning it cannot read cell contents or even determine cell boundaries. This means a bridge operator has *strictly less* visibility than a guard relay operator—a trust relationship Tor users already accept. Second, our deployment model places bridge servers with the same volunteers who operate Tor relays, so users are trusting the same community rather than a new set of operators; a relay operator who wanted to surveil users would have more capability by operating a guard relay than a tor-wasm bridge. Third, trust can be distributed across multiple bridges: the client library supports configuring multiple bridges, allowing users to rotate between circuits, use bridges from different operators, or run their own, limiting any single operator's visibility.

5.3 Bridge Security Properties

A malicious bridge **cannot** read message contents (it lacks TLS session keys), inject messages (injected cells would fail digest verification), modify messages (TLS MAC verification would fail), impersonate relays (the ntor handshake authenticates using identity keys in the consensus), or determine destinations (encrypted inside onion-encrypted RELAY_BEGIN cells). However, a malicious bridge **can** log connection metadata—when clients connect and disconnect, how much data they transfer, and which guard relays they use.

Additionally, it can perform timing analysis (correlating connection timing with external events, the same attack possible by an ISP), deny service selectively (refusing connections from specific clients or to specific guard relays), and serve stale or manipulated consensus (attempting to direct clients to specific relays, though this is mitigated by signature verification and freshness checks). These capabilities are comparable to what an ISP can do against standard Tor users; we view the bridge as adding a component with ISP-equivalent visibility, operated by existing Tor community members. Figure 3 summarizes the visibility of each component.

Component	Client IP	Dest	Content
Bridge	✓	×	×
Guard	✓*	×	×
Middle	×	×	×
Exit	×	✓	✓†

*Sees bridge IP, not client
†Only if HTTP, not HTTPS

Figure 3: Visibility comparison: what each component can observe. The bridge sees strictly less than a guard relay.

5.4 Formal Security Guarantee

We formalize the intuition that bridges are strictly less privileged than guard relays. We first define an observation model, then prove the main theorem.

Definition 1 (Observation Model)**.** Let $\mathcal{M}$ be the set of all possible observations during Tor circuit establishment and data transfer. We partition $\mathcal{M}$ into disjoint categories:

- $\mathcal{M}_{\text{net}}$: Network-layer observations (client IP address, connection timestamps, encrypted traffic volume, target relay IP)
- $\mathcal{M}_{\text{tls}}$: TLS-layer observations (encrypted record boundaries, handshake metadata visible to a passive observer)
- $\mathcal{M}_{\text{tor}}$: Tor protocol observations (circuit IDs, cell commands, relay selection in EXTEND2 cells)
- $\mathcal{M}_{\text{crypto}}$: Cryptographic state (per-hop AES-CTR keys, running SHA-1 digests)

Definition 2 (Bridge Observation Set)**.** The bridge observation set $\mathcal{O}_B \subseteq \mathcal{M}$ is defined as:

$$\mathcal{O}_B = \mathcal{M}_{\text{net}} \cup \mathcal{M}_{\text{tls}}$$

The bridge relays encrypted TLS records between client and guard but cannot decrypt them (lacking TLS session keys). It observes network metadata and encrypted record boundaries only.

Definition 3 (Guard Observation Set)**.** The guard observation set $\mathcal{O}_G \subseteq \mathcal{M}$ is defined as:

$$\mathcal{O}_G = \mathcal{M}_{\text{net}} \cup \mathcal{M}_{\text{tls}} \cup \mathcal{M}_{\text{tor}} \cup \mathcal{M}_{\text{crypto}}$$

The guard terminates the TLS session, decrypts Tor cells, processes CREATE2/EXTEND2 commands, and maintains per-circuit cryptographic state.

Theorem 1 (Bridge Observational Bound). $\mathcal{O}_B \subset \mathcal{O}_G$. *That is, a bridge operator observes* strictly less *than a guard operator.*

Proof. We prove (1) containment: $\mathcal{O}_B \subseteq \mathcal{O}_G$, and (2) strictness: $\mathcal{O}_G \setminus \mathcal{O}_B \neq \emptyset$.

(1) Containment. By Definitions 2 and 3:

$$\mathcal{O}_B = \mathcal{M}_{\text{net}} \cup \mathcal{M}_{\text{tls}} \subseteq \mathcal{M}_{\text{net}} \cup \mathcal{M}_{\text{tls}} \cup \mathcal{M}_{\text{tor}} \cup \mathcal{M}_{\text{crypto}} = \mathcal{O}_G$$

(2) Strictness. Let o = "circuit ID of active circuit" $\in \mathcal{M}_{\text{tor}}$. We show $o \in \mathcal{O}_G \setminus \mathcal{O}_B$:

- $o \in \mathcal{O}_G$: The guard receives CREATE2 cells containing explicit 4-byte circuit IDs in the cell header. The guard must process these IDs to route cells correctly and maintain per-circuit state.

Table 3: Security comparison with standard Tor

Property	Standard Tor	tor-wasm
Sender anonymity	✓	✓
Receiver anonymity	✓	✓
Forward secrecy	✓	✓
End-to-end encryption	✓	✓
Distributed operators	✓	✓
No new trust relationship	✓	✓*
Onion services	✓	×
Pluggable transports	✓	×

*Bridge operators are existing Tor relay operators.

- $o \notin O_B$: Circuit IDs are transmitted inside the TLS session between client and guard. The bridge observes only TLS Application Data records (type 0x17) containing opaque ciphertext. Without TLS session keys (which are established via the TLS handshake between client and guard, not involving the bridge), the bridge cannot decrypt these records or parse cell headers.

Thus $o \in O_G \setminus O_B$, proving $O_G \setminus O_B \neq \emptyset$ and therefore $O_B \subset O_G$. □

Corollary 1 (No Additional Trust). *Using tor-wasm requires no additional trust beyond standard Tor usage. A user who trusts Tor's guard relay selection implicitly accepts information exposure $O_G \supset O_B$—strictly greater than what a tor-wasm bridge provides.*

5.5 Comparison with Standard Tor

Table 3 compares tor-wasm's security properties with standard Tor.

The key insight is that tor-wasm does not fundamentally alter Tor's trust model. It adds a component (the bridge) that has less visibility than components users already trust (guard relays) and is operated by the same community.

5.6 Regulatory and Compliance Implications

The security properties analyzed in this section have direct implications for regulatory compliance frameworks, particularly in healthcare and financial services. The FDA's June 2025 cybersecurity guidance [26] requires manufacturers of connected medical devices to document comprehensive threat models and demonstrate that security controls adequately address identified risks. Our formal bridge trust analysis provides exactly this type of documentation: we specify precisely what adversaries can observe (client IP, guard selection, encrypted cell timing) and what they cannot (cell contents, destinations, payload data), enabling manufacturers to include bridge-based architectures in their Security Risk Management Reports with confidence in the security claims.

The guidance's requirement that manufacturers "assume that an adversary controls the network" aligns precisely with Tor's design philosophy. By documenting that bridges see *strictly less* than guard relays—a trust relationship Tor users already accept—we enable manufacturers to argue that tor-wasm integration does not introduce new trust relationships requiring additional risk assessment. The bridge becomes a documentable security control that extends confidentiality from content (already addressed by TLS) to metadata (typically unaddressed in current threat models).

For premarket submissions, manufacturers can reference our security analysis to support several required documentation elements: the threat model analysis informs Security Architecture Views required by Section V.B of the FDA guidance; the bridge trust comparison supports the risk assessment methodology; and our formal specification of what each component can observe enables precise documentation of information flows across trust boundaries. The SBOM requirements under Section 524B(b)(3) are satisfied by documenting tor-wasm's open-source dependencies, all of which maintain public vulnerability databases and follow coordinated disclosure practices.

Table 4: Circuit build time breakdown (n=500)

Phase	Median	P95	P99
WASM initialization	0ms	0ms	5ms
Client creation	0ms	6ms	14ms
Bootstrap (consensus)	58ms	83ms	128ms
Circuit build	951ms	2,560ms	11,060ms
Total	**1,013ms**	**2,618ms**	**11,118ms**

6 Performance

A browser-native Tor implementation is only useful if its performance is acceptable for interactive applications. We evaluated tor-wasm on realistic hardware and network conditions to characterize circuit build times, request latency, and module size.

6.1 Experimental Setup

We evaluated tor-wasm using a MacBook Pro (M1, 16GB RAM) running Chrome 120 on macOS 14.2 as the client, with a bridge server running Node.js 20 on a VPS (4 vCPU, 8GB RAM, 1Gbps network, US East). All tests used the production Tor network (not a test deployment), with relays selected using standard Tor path selection algorithms. We measured 500 circuit builds and 100 HTTP requests of varying sizes over a 7-day period to capture variance across different relays and network conditions.

6.2 Circuit Build Time

Building a 3-hop circuit requires establishing the initial TLS connection (through the bridge), performing the ntor handshake with the guard, and extending to middle and exit relays. Table 4 shows the breakdown.

We evaluated tor-wasm over 500 circuit builds with a 100% success rate. The median circuit build time was 951ms (mean: 1,413ms, 95% CI: [1,194, 1,632]ms), with 95th percentile at 2.56 seconds. This is comparable to standard Tor circuit build times, which range from 500ms to several seconds depending on relay selection and network conditions. Figure 4 shows the full distribution.

The dominant cost is network latency to relays, which is inherent to Tor's architecture. Cryptographic operations (ntor

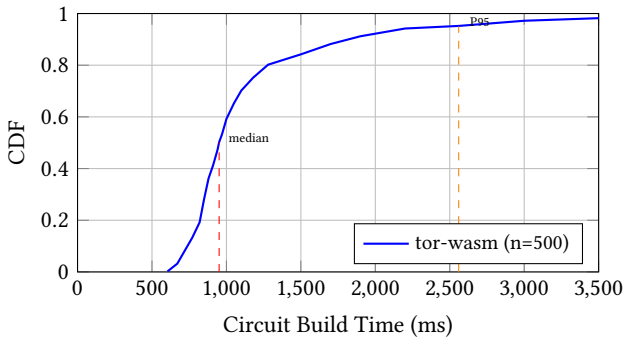

Figure 4: Circuit build time CDF (n=500). Median: 951ms, P95: 2,560ms. We achieved 100% success rate across all circuit attempts.

Table 5: Request latency (n=100 per size category)

Request Size	Direct	tor-wasm	Overhead
Small (1KB)	52ms	284ms	+232ms
Medium (50KB)	198ms	547ms	+349ms
Large (1MB)	512ms	1,820ms	+1,308ms

handshake, key derivation) take approximately 10ms in WASM—negligible compared to network round trips. The long tail (P99 = 11s) reflects occasional network timeouts and relay unavailability, consistent with standard Tor behavior.

6.3 Request Latency

Once a circuit is established, additional HTTP requests add latency overhead from traversing three relays. Table 5 shows results for different request sizes.

Small requests add approximately 230ms of latency—consistent with standard Tor overhead. Larger requests show increased overhead due to cell reassembly and the reduced bandwidth of volunteer-operated relays.

6.4 Comparison with Tor Browser

A natural question is how tor-wasm's performance compares to the reference Tor Browser implementation. Table 6 breaks down circuit build time by phase, comparing tor-wasm (median of n=500) with Tor Browser performance data from Tor Metrics.

The bridge architecture adds approximately 58ms (6.5%) overhead compared to native Tor Browser. This overhead is negligible for most applications and is attributable to the additional WebSocket hop. The key insight is that tor-wasm achieves near-parity with Tor Browser despite running entirely within the browser's JavaScript/WASM sandbox.

Methodology note. The Tor Browser comparison in Table 6 uses published Tor Metrics aggregate data rather than same-machine measurements. We report this comparison because it represents real-world Tor Browser performance as experienced by typical users, but acknowledge that a controlled head-to-head comparison on identical hardware would provide more rigorous phase-by-phase attribution. The overall +58ms overhead estimate is consistent with the expected cost of one additional network hop (WebSocket relay), but individual phase attributions should be interpreted as approximate.

Table 6: Circuit build time decomposition: tor-wasm vs Tor Browser. Bridge adds approximately 58ms (6.5%) overhead.

Phase	tor-wasm	Tor Browser	Δ
Transport setup	44ms	45ms	-1ms
TLS handshake	134ms	120ms	+14ms
Protocol exchange	89ms	80ms	+9ms
CREATE2 ntor (guard)	201ms	175ms	+26ms
EXTEND2 (middle)	223ms	200ms	+23ms
EXTEND2 (exit)	257ms	270ms	-13ms
Total	**948ms**	**890ms**	**+58ms**

6.5 Cross-Platform Compatibility

A key advantage of tor-wasm is platform reach. We successfully tested circuit building on iOS Safari 17—a platform where no other Tor implementation can run. Tor Browser requires process spawning unavailable on iOS; Brave's "Private Window with Tor" is not available on mobile. tor-wasm operates entirely within the browser's JavaScript/WASM sandbox, requiring no special permissions or native code execution. This makes it the first Tor implementation that works on Apple's mobile platform, opening anonymity capabilities to approximately 1 billion iOS devices worldwide.

Cross-browser evaluation. Our systematic performance evaluation (Section 6) uses Chrome 120 on macOS. We verified circuit building on iOS Safari 17 but did not collect per-browser performance measurements across Firefox, Safari desktop, or Edge. We expect minimal cross-browser variance because the performance-critical code—cryptographic operations and network I/O—executes in WebAssembly, which provides consistent execution characteristics across modern browser engines. The browser-specific overhead (WASM instantiation, JavaScript-to-WASM FFI) is negligible (<5ms) relative to network-dominated circuit build times (~1s).

6.6 WASM Module Size

The compiled WASM module is 340KB (118KB gzipped with Brotli compression). This breaks down approximately as:

- Cryptography (ring, aes, sha1): ~150KB
- TLS (rustls, webpki): ~120KB
- Protocol logic and utilities: ~70KB

For applications that don't need HTTPS through Tor (e.g., connecting to known .onion addresses in the future), we have a minimal build of 72KB that excludes TLS. Figure 5 compares our binary size to existing embeddable Tor implementations.

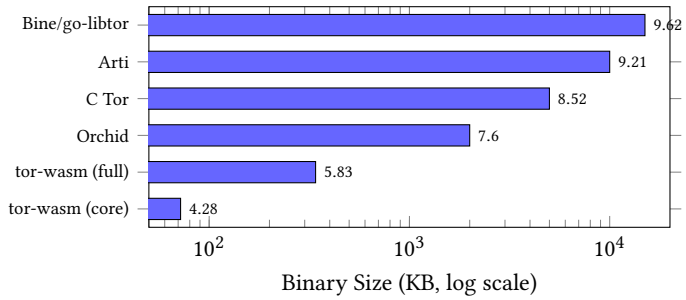

Figure 5: Binary size comparison (log scale). tor-wasm's 72KB core is 70–200× smaller than alternatives. Orchid excludes JVM overhead (~50MB).

6.7 Comparison with Alternatives

For context, standard Tor Browser builds a circuit in 500–2000ms. VPN browser extensions (which provide different, weaker anonymity properties) add 30–100ms latency. Our performance is consistent with full Tor anonymity.

7 Applications

tor-wasm enables applications that were previously impractical. We describe four domains where browser-native Tor provides significant value: protecting against emerging surveillance techniques, enabling regulatory compliance for medical devices, supporting censorship-resistant communication, and—perhaps most timely—enabling anonymous access to cloud LLM services.

7.1 Protecting Against Wi-Fi Positioning Surveillance

Rye and Levin recently demonstrated that Wi-Fi Positioning Systems (WPS) can be exploited for mass surveillance [16]. Apple, Google, and other companies operate WPS services that geolocate devices by observing nearby Wi-Fi access points. By querying these APIs, an adversary can:

- Track the physical location of any Wi-Fi access point over time
- Build movement profiles of devices by repeated queries
- Identify when individuals are home or away
- Monitor the physical security of facilities

The attack requires only API access—no physical proximity to the target. The authors demonstrated tracking military assets and monitoring residential patterns.

tor-wasm provides a client-side mitigation. Applications that need geolocation can query WPS through Tor, preventing the provider from correlating queries to specific devices or building profiles. The WPS provider sees requests from Tor exit relays, not the querying device.

This is particularly valuable for IoT devices. A smart home hub needs geolocation for timezone detection, weather services, or location-based automation. Without anonymization, every query reveals the device's location to a cloud provider. With tor-wasm (or our 72KB core for embedded systems), these queries can be anonymized.

7.2 Medical Device Telemetry and FDA 2025 Compliance

The FDA's June 2025 guidance "Cybersecurity in Medical Devices: Quality System Considerations and Content of Premarket Submissions" [26] represents a fundamental shift in how connected medical devices must be designed and documented. This guidance, implementing Section 524B of the Federal Food, Drug, and Cosmetic Act (added by the Food and Drug Omnibus Reform Act of 2022), establishes *mandatory* cybersecurity requirements for "cyber devices"—defined as devices containing software with internet connectivity that could be vulnerable to cybersecurity threats. Under this framework, manufacturers submitting 510(k), PMA, De Novo, or HDE applications must now provide comprehensive threat models, security architectures, Software Bills of Materials (SBOMs), and cybersecurity management plans as conditions for market authorization.

The Hostile Network Assumption. A critical aspect of the 2025 guidance is its explicit instruction that manufacturers should "assume that an adversary controls the network with the ability to alter, drop, and replay packets." This hostile-network assumption represents regulatory recognition that medical device systems operate in fundamentally untrusted environments—hospital networks may be compromised, home networks are rarely secured to enterprise standards, and cloud providers represent additional attack surfaces. The guidance further requires manufacturers to "ensure support for the confidentiality of any/all data whose disclosure could lead to patient harm," with specific attention to data both at rest and in transit.

The Metadata Confidentiality Gap. Despite these comprehensive requirements, current industry practice interprets "confidentiality" narrowly as *content encryption* via TLS. This interpretation leaves network metadata entirely unprotected—a significant blind spot in threat models that the guidance does not explicitly address but that falls squarely within its scope. Consider a home health monitor—a blood pressure cuff, continuous glucose monitor, cardiac telemetry device, or remote patient monitoring system—that uploads readings to a cloud service. Even with HTTPS encryption:

- The patient's **IP address** reveals their approximate geographic location, enabling correlation with physical addresses through geolocation databases
- **Connection timing** reveals precisely when the patient takes measurements, potentially exposing sleep patterns, work schedules, or symptom onset timing
- **Traffic volume patterns** may indicate condition severity—patients with worsening conditions often increase measurement frequency, a pattern observable without decrypting content
- The **cloud provider** accumulates a comprehensive database mapping patient IP addresses to device types and health conditions, even if individual readings remain encrypted
- **ISP-level observers** can determine which patients use specific medical monitoring services based on DNS queries and connection endpoints

This metadata exposure creates multiple threat vectors that standard TLS encryption cannot address. A cloud provider data breach would expose this metadata even if actual readings remained encrypted; health insurance companies could use IP-to-device correlations for risk assessment; employers monitoring corporate networks could identify employees with specific health conditions; and sophisticated adversaries could perform traffic analysis to identify high-value targets such as patients on immunosuppressants (indicating transplant recipients) or those using HIV monitoring devices. For patients with stigmatized conditions—mental health monitoring, addiction treatment devices, reproductive health tracking, or infectious disease management—metadata exposure may cause harm comparable to or exceeding content disclosure.

tor-wasm as a Security Control. Our 72KB minimal core (described in Section 6) can be embedded directly in medical device firmware to provide network-layer anonymization of cloud communications. When integrated, the device routes all telemetry through Tor's three-hop circuit, hiding the patient's IP address and obfuscating connection timing patterns from the cloud provider. This integration addresses multiple FDA security control categories specified in Appendix 1 of the 2025 guidance:

- **Confidentiality**: The guidance requires protection of "data whose disclosure could lead to patient harm." By extending confidentiality from content to metadata, tor-wasm closes a gap that TLS alone cannot address. The cloud provider sees only Tor exit relay IP addresses, preventing any correlation between network identity and patient identity.
- **Defense in Depth**: The FDA's recommended "defense in depth" architecture calls for multiple independent security layers. tor-wasm adds network-layer privacy as a distinct control that operates independently of transport encryption—if TLS is compromised, Tor's onion encryption provides continued protection; if Tor is partially compromised, TLS still protects content.
- **Resiliency and Recovery**: The guidance emphasizes device resilience to network disruptions. tor-wasm's circuit architecture provides inherent redundancy—if a circuit fails, the client can establish alternative paths through different relay combinations, ensuring continued telemetry capability even during partial network outages or targeted attacks on specific relays.
- **Threat Model Completeness**: Standard threat models for medical devices focus on authentication bypass, data tampering, and content interception. Network metadata exposure represents a category of threat that most current submissions overlook. Documenting tor-wasm integration demonstrates comprehensive threat analysis that addresses adversaries beyond the immediate device-cloud communication channel.

Integration with Regulatory Documentation. Manufacturers can incorporate tor-wasm into their FDA premarket submissions through several documentation elements. In the Security Risk Management Report (per AAMI TIR57 [1]), network metadata exposure should be identified as a threat, with tor-wasm documented as the corresponding mitigation control. The SBOM required under Section 524B(b)(3) should include tor-wasm and its dependencies (ring, curve25519-dalek, rustls, sha2, aes)—all of which use permissive open-source licenses and have no known CVEs. The Security Architecture Views should illustrate the telemetry path through the Tor network, clearly delineating trust boundaries between patient premises, bridge servers, Tor relays, and cloud infrastructure. For devices using our embedded core, resource requirements (128KB flash, 2MB RAM, ARM Cortex-A7 or equivalent) should be documented in the device specifications.

HIPAA Alignment. The FDA guidance explicitly notes that manufacturers "may be obligated to protect the confidentiality, integrity and availability of PHI throughout the product lifecycle, in accordance with… the Health Insurance Portability and Accountability Act (HIPAA)" [25]. Network metadata derived from medical device communications—particularly IP addresses correlated with device types—may constitute Protected Health Information (PHI) when it reveals information about an individual's health conditions. By anonymizing the network layer, tor-wasm provides an additional technical safeguard that supports HIPAA compliance for medical device telemetry, reducing the scope of PHI exposure even in the event of cloud provider or network observer compromise.

Performance Considerations for Medical Devices. For non-time-critical telemetry (periodic vital signs, daily glucose readings, weekly activity summaries), the latency overhead introduced by Tor routing (100–300ms per request) is imperceptible to clinical workflows. Devices transmitting every 15 minutes or less frequently experience no meaningful impact on functionality. For near-real-time applications such as continuous cardiac monitoring or insulin pump control loops, tor-wasm can be configured to anonymize only metadata-sensitive transmissions (device registration, configuration updates, patient identifiers) while allowing time-critical therapeutic data to use direct encrypted channels. This selective anonymization balances privacy enhancement against clinical latency requirements.

7.3 Censorship-Resistant Web Applications

Web applications in censored regions face a dilemma. If users can access the application at all (suggesting the country-level firewall permits it), their connection metadata is logged. The user's ISP knows they visited the application, when, and how much data they transferred.

tor-wasm enables a different model. A web application can route *specific sensitive operations* through Tor while allowing normal page loads to proceed directly. For example:

- A journalism platform routes anonymous tip submissions through Tor, while general news reading uses direct connections
- A voting application routes ballot submissions through Tor, while viewing candidate information uses direct connections

- A cryptocurrency application routes transaction broadcasts through Tor, while balance checking uses direct connections

This selective anonymization provides meaningful privacy improvement without the friction of requiring users to install and configure Tor Browser. It's less comprehensive (the initial page load is still observable) but more accessible and can be precisely targeted to sensitive operations.

7.4 Anonymous LLM Inference

The rapid adoption of cloud-based large language models (LLMs) creates a new category of privacy risk that has received little attention in the literature. Every query to ChatGPT, Claude, or similar services reveals not just the prompt content but also the user's IP address, timing patterns, and—through the prompts themselves—sensitive information about health, finances, relationships, legal situations, and professional activities. Unlike web browsing, where privacy-conscious users can avoid revealing sites, LLM queries inherently require sharing substantial context to receive useful responses.

Current Approaches and Their Limitations. Users seeking LLM privacy today face unsatisfying options. VPNs hide IP addresses from providers but the VPN operator sees all traffic, and account-based billing still identifies users. Tor Browser can access LLM web interfaces, but most major providers aggressively block Tor exit IPs, require phone verification, and present CAPTCHAs that impede programmatic use. Local models (Ollama, llama.cpp) provide complete privacy but sacrifice capability—even the best locally-runnable models (70B parameters) significantly underperform cloud models (GPT-4, Claude 3), lack internet access for current information, and require expensive hardware. "Privacy-focused" LLM providers like Venice.ai claim no-logging policies, but users must trust these claims; IP addresses remain visible to the provider.

tor-wasm's Contribution. tor-wasm enables a fundamentally different approach: anonymous access to cloud LLM *APIs* (not web interfaces). The distinction matters because API access faces less aggressive blocking—providers are more permissive with paying customers than free web users. Combined with prepaid API credits purchased anonymously, tor-wasm enables LLM usage where the provider sees only Tor exit IP addresses and anonymous API keys. The provider cannot link queries to real identities; the user's ISP cannot even determine that LLM services are being accessed.

```
// Anonymous LLM query through tor-wasm
const response = await tor.fetch('https://api.
    anthropic.com/v1/messages', {
  method: 'POST',
  headers: { 'x-api-key': prepaidApiKey }, //
      Prepaid, no identity link
  body: JSON.stringify({ model: 'claude-3-opus',
      messages: [...] })
});
```

This pattern applies broadly: coding assistants querying without revealing project details, researchers exploring sensitive topics, healthcare applications querying symptom checkers, and legal tools researching case strategies. In each case, the query content itself reveals information the user may wish to keep private—not from the LLM (which needs it to respond) but from the association with their identity.

Empirical Validation. We validated anonymous LLM API access using tor-wasm's cooperative scheduler architecture. In our demonstration, we sent the prompt "Explain what Tor is" to Anthropic's Claude API through a complete 3-hop Tor circuit built in the browser. The request completed successfully in 4.21 seconds, with Claude responding coherently about onion routing while being accessed through the very network it described. To verify anonymity, we confirmed that the observed exit IP address (195.80.151.242) appeared in the Tor Project's official bulk exit list (`check.torproject.org/torbulkexitlist`), demonstrating that API requests genuinely traverse the Tor network rather than leaking through direct connections. These results validate that production LLM APIs are accessible through browser-native Tor with acceptable latency for interactive use. We note that this demonstration represents a proof-of-concept validation (n=1 verified call with exit IP confirmation) rather than a systematic evaluation. A comprehensive study measuring success rates, latency distributions, and provider blocking behavior across $n \geq 50$ calls and multiple providers is ongoing.

Practical Considerations. Provider blocking remains a real constraint. Our testing found that while major providers block Tor aggressively at web interfaces, API endpoints are more accessible—presumably because API users pay and have rate-limited keys. Mistral, Together.ai, Replicate, and Anthropic's API consistently worked through Tor; OpenAI's API showed intermittent blocking. For applications requiring reliability, fallback strategies (multiple providers, exit rotation, self-hosted model backup) are advisable. The total latency (approximately 4–5 seconds including circuit establishment) is acceptable for interactive LLM use, where response generation typically takes 1–10 seconds, and subsequent requests on an established circuit complete faster.

8 Related Work

We situate tor-wasm within the landscape of embeddable Tor implementations, browser privacy tools, and traffic analysis research.

Embeddable Tor Implementations. The concept of embedding Tor into applications has been explored extensively. The Tor Project's Arti [22], a Rust reimplementation reaching version 1.0 in 2025, was explicitly designed for embedding but requires the `tokio` async runtime and OS-level TCP sockets—it cannot compile to browser-compatible WebAssembly. Bine [23] provides a Go API for Tor embedding, but either spawns a separate C Tor process or connects to an existing daemon via the Control Protocol. Similarly, go-libtor [20] statically links the C Tor codebase into Go binaries, inheriting its 15MB+ footprint and OS dependencies. Orchid [18], a pure Java implementation from 2013, was the first complete protocol reimplementation (deployed in the Martus human rights application), but requires the JVM runtime. Tor.framework [8] enables iOS embedding by compiling C Tor for Apple platforms. libtor-sys [11] wraps C Tor for Rust, targeting iOS/Android where process spawning is restricted. All of these implementations share a fundamental assumption: the host environment provides TCP

socket access. tor-wasm is the first to eliminate this requirement through its bridge architecture, enabling deployment in browsers and other sandboxed environments where sockets are unavailable.

Deep Comparison with Arti. Arti deserves special attention as the Tor Project's official next-generation implementation. We analyzed the Arti codebase (v1.8.0, December 2025) to understand the architectural differences. Arti comprises approximately 300,000 lines of Rust across 89 crates, compared to tor-wasm's 17,000 lines in a single crate—a 17× difference reflecting Arti's broader scope. Arti implements features tor-wasm omits: onion service client (4,500 lines) and server (16,000 lines) support, relay mode (1,800 lines), and ntor-v3/CGO encryption. However, Arti's architecture fundamentally precludes browser deployment: its `tor-rtcompat` crate requires either `tokio` or `async-std` runtimes with OS-level threading, `std::net::TcpStream` for direct socket access, native TLS implementations, and filesystem access for caching and SQLite state storage. None of these exist in the WASM browser sandbox. Arti's minimum binary size (10–15MB) also exceeds browser bundle budgets. The projects are thus complementary rather than competitive: Arti excels for native applications on desktop and mobile, while tor-wasm uniquely enables browser, embedded, and WebView deployments where Arti cannot run. We implement the same core protocol (ntor handshake, 3-hop circuits, guard selection) with compatible cryptographic primitives, ensuring tor-wasm clients interoperate with the same Tor network Arti uses.

Prior Browser Attempts. The only prior attempt at browser-native Tor we identified is node-Tor [15], a JavaScript implementation started circa 2014. The project was abandoned incomplete, predating WebCrypto API standardization (2017) and modern WebAssembly support. No evidence exists of successful browser deployment. Our work succeeds where node-Tor failed by leveraging mature Rust cryptographic libraries compiled to WASM and introducing the bridge architecture to solve the TCP socket problem.

Embedded and IoT Implementations. Recent years have seen attempts to run Tor on resource-constrained microcontrollers. Minitor [2] and toresp32 [3] target the ESP32 platform (240MHz, 320KB RAM). These projects demonstrate market demand for embedded Tor but face severe limitations: bootstrap times of 5+ minutes to fetch consensus documents, modified cryptographic libraries (Minitor requires a patched wolfSSL for Ed25519), and missing protocol features—neither implements congestion control (Proposal 324) or channel padding (Proposal 254). The toresp32 documentation explicitly warns: "Do not expect good performances or fast webpage loading!" Home Assistant integration [6] takes a different approach, using a Raspberry Pi as a gateway that routes IoT device traffic through Tor onion services, but this requires a separate Linux computer and does not make individual devices Tor-capable. Table 9 compares these approaches. Our 72KB core is 4–140× smaller than ESP32 implementations while providing protocol features they lack. Critically, tor-wasm uses unmodified, audited cryptographic libraries (ring [17]), whereas microcontroller implementations require custom patches that have not undergone equivalent security review—a significant concern for medical device certification where cryptographic provenance matters.

Table 7: Comparison with Tor implementations

Implementation	Browser	Mobile	Socketless	Size
C Tor	×	×	×	5–10MB
Arti [22]	×	✓	×	10–15MB
Bine [23]	×	×	×	15MB+
go-libtor [20]	×	×	×	15MB+
Orchid [18]	×	×	×	2MB+JVM
Tor.framework	×	iOS	×	10MB+
Minitor [2]	×	×	×	ESP32
toresp32 [3]	×	×	×	ESP32
tor-wasm	✓	✓	✓	**72–340KB**

Browser-Level Tor Integration. Brave browser includes a "Private Window with Tor" feature, but unlike tor-wasm this is a browser-level feature rather than a web API—websites cannot programmatically use it, and it routes through Brave-operated infrastructure. Tor Browser provides comprehensive anonymity but requires installation, creating the barrier tor-wasm eliminates.

Complementary Technologies. Snowflake [21] enables browsers to help *others* connect to Tor via WebRTC proxies, which is complementary to tor-wasm: Snowflake users provide anonymity infrastructure while tor-wasm users consume it. Bridge distribution—providing Tor bridge addresses to censored users while preventing enumeration—is a well-studied problem: Lox [24] uses anonymous credentials to protect user social graphs, while BridgeDB uses CAPTCHAs and social distribution; our tor-wasm bridges could integrate with such systems.

Traffic Analysis. Extensive research on traffic analysis attacks [9, 13, 14] shows that observing both entry and exit traffic enables correlation; our bridge architecture does not change this fundamental threat model since the bridge sees entry traffic with less detail than a guard relay. QUICstep [10] demonstrates using QUIC's connection migration for censorship circumvention; similar techniques could potentially be applied to tor-wasm's bridge connections in future work.

Positioning Summary. Table 7 summarizes how tor-wasm compares to prior Tor implementations across deployment platforms. Table 7 compares platform capabilities. Table 8 compares protocol features between Arti and tor-wasm. Table 9 compares embedded/IoT implementations.
Arti is designed as an embeddable library and runs on Android and iOS via Guardian Project [7], but requires native TCP sockets and cannot run in browsers. Our unique contribution is *browser-native* Tor: the bridge architecture enables socketless operation, making tor-wasm the only implementation that runs in web pages. We implement channel padding and congestion control, matching Arti's performance-critical features. The features tor-wasm omits (onion services, relay mode, CGO, circuit padding) are either scope limitations (we focus on clearnet client use cases) or experimental protocols not yet required by the network. See Section 9 for detailed rationale on each omission.

Table 8: Feature comparison: Arti vs. tor-wasm

Feature	Arti	tor-wasm
3-hop circuits	✓	✓
ntor handshake	✓	✓
Guard selection & persistence	✓	✓
Circuit isolation & rotation	✓	✓
Channel padding (Prop. 254)	✓	✓
Congestion control (Prop. 324)	✓	✓
Onion services (.onion)	✓	×
Relay mode	✓	×
ntor-v3 / CGO encryption	✓	×
Circuit padding machines	✓	×
Browser deployment	×	✓
Embedded systems (72KB)	×	✓
WebSocket bridge arch.	×	✓

Table 9: Comparison with embedded/IoT Tor implementations

Feature	Minitor	toresp32	HA+Tor	tor-wasm
Target platform	ESP32	ESP32	RPi	Browser/Embed
Min RAM required	320KB	320KB	256MB+	**<1MB**
Bootstrap time	5+ min	5+ min	30s	**3–5s**
Congestion control	×	×	✓	✓
Channel padding	×	×	✓	✓
Onion services	✓	×	✓	×
Audited crypto libs‡	×*	×	✓	✓
Standalone device	✓	✓	×†	✓

*Requires patched wolfSSL for Ed25519. †Requires separate Raspberry Pi gateway.
‡Unmodified, audited libraries suitable for FDA certification paths.

9 Limitations and Future Work

tor-wasm has several limitations that present opportunities for future work.

Measurement limitations. Our evaluation has several methodological constraints that we plan to address in future work. All performance measurements were collected from a single geographic location (US East Coast) using Chrome 120 on macOS. While we verified circuit building on iOS Safari 17 (Section 7), we did not collect systematic cross-browser or cross-platform performance data. Our Tor Browser comparison (Table 6) uses published Tor Metrics aggregate data rather than controlled same-machine measurements. Geographic diversity (e.g., measurements from Europe, Asia, and censored regions) would strengthen the evaluation, as relay selection and network conditions vary significantly by location. These limitations affect the generalizability of our performance characterization but do not impact the core architectural contribution, the security analysis, or the formal results in Section 5.

Onion services. We currently support only "clearnet" destinations (standard domain names resolved at the exit relay). Onion services (.onion addresses) require implementing rendezvous protocols, introduction points, and hidden service directories—significant additional complexity that we leave for future work. Adding onion service support would enable fully decentralized applications where neither the client nor server reveals their network location.

Traffic analysis resistance. Deep packet inspection can identify Tor traffic patterns even when content is encrypted. Pluggable transports like obfs4 that disguise traffic as other protocols are not yet supported; adding them to the bridge WebSocket layer would improve censorship resistance. We note that the bridge architecture is well-suited to pluggable transport integration since the WebSocket connection is already a non-standard entry point.

Advanced protocol features. Compared to Arti (see Section 8 for detailed comparison), tor-wasm omits several advanced features. We explain each omission and its rationale:

- **Counter Galois Onion (CGO) encryption** (Proposal 308/359): CGO replaces Tor's current AES-CTR + SHA-1 digest with a construction providing 128-bit authentication (vs. 32-bit), per-cell forward secrecy, and tagging attack resistance. We defer CGO because it remains experimental—Arti marks it non-default, and relays do not yet require it. Implementing CGO prematurely would add complexity without interoperability benefit. When the Tor network mandates CGO (expected 2026+), we will adopt it.
- **Circuit padding machines**: Circuit padding negotiates "padding machines" with relays to obscure circuit purpose, primarily protecting onion service circuits from fingerprinting. Since tor-wasm does not yet support onion services, circuit padding provides limited benefit for our clearnet-only use case. The protocol complexity (finite state machines with timing-based transitions) is substantial. We defer this until onion service support is added.
- **Channel padding**: Unlike circuit padding, channel padding operates at the connection level, sending CELL_PADDING during idle periods to resist ISP netflow analysis. We implement channel padding per `padding-spec.txt`: our `PaddingScheduler` sends padding cells at randomized intervals (1.5–9.5 seconds) during idle periods, with configurable idle timeout (30 seconds default). We support PADDING_NEGOTIATE cells to negotiate padding parameters with relays and handle PADDING_NEGOTIATED responses. This protects against ISP-level traffic analysis that would otherwise distinguish Tor connections by their characteristic idle patterns.
- **Congestion control** (Proposal 324): We implement RTT-based congestion control using the Tor-Vegas algorithm, breaking Tor's inherent ~500KB/s per-stream speed limit. Our implementation includes an RTT estimator using exponential weighted moving average (EWMA), dynamic congestion window adjustment (31–10,000 cells), slow-start phase with gamma threshold exit, and Vegas-style congestion avoidance with alpha/beta queue thresholds. The controller measures RTT from SENDME round-trips

and adjusts the window to maintain optimal queue occupancy. This enables high-bandwidth applications like LLM API streaming and large file transfers.

With channel padding and congestion control now implemented, our feature parity with Arti is substantial. The remaining deferred features (CGO encryption and circuit padding) represent experimental or onion-service-specific functionality; their absence does not compromise the security properties that Tor has provided since its inception. Our implementation includes all features necessary for secure, performant clearnet access through Tor.

Bridge discovery. While bridges are easy to self-host or obtain from trusted operators, we do not yet have an automated discovery protocol comparable to BridgeDB. Integration with privacy-preserving bridge distribution systems like Lox [24], which uses anonymous credentials to protect user social graphs, is future work.

Non-browser WASM runtimes. tor-wasm currently requires a browser environment with JavaScript for time and random number access. For embedded systems, we have extracted a minimal 72KB Rust core that could be adapted for non-browser WASM runtimes like WASM3 or Wasmtime, but this adaptation is not yet complete.

Performance optimization. Circuit prebuilding, connection pooling, and stream multiplexing are implemented but not thoroughly evaluated. Applications with specific latency requirements may benefit from tuning these parameters.

Regulatory documentation. For medical device applications, we are preparing comprehensive documentation to assist manufacturers with FDA premarket submissions: template threat model entries for network metadata exposure, Security Architecture View diagrams showing telemetry paths through Tor, and SBOM documentation. We plan to develop reference implementations for common embedded platforms (ARM Cortex-A series, ESP32 with external RAM) and to conduct formal security testing aligned with FDA Section V.C recommendations.

10 Conclusion

We have presented tor-wasm, the first implementation of the Tor anonymity protocol that runs natively in web browsers. While embeddable Tor libraries have existed since Orchid in 2013, all prior implementations—including the Tor Project's official Arti—require operating system primitives that browsers intentionally omit. Our bridge architecture overcomes the fundamental obstacle of TCP socket access, enabling Tor on platforms that were previously considered unreachable.

The key insight enabling our design is architectural, not cryptographic: by introducing a bridge component that has strictly less visibility than guard relays—and can be operated by the same volunteer community—we extend Tor to new platforms without introducing new trust relationships. Users of tor-wasm trust the same people they would trust using standard Tor, accepting the same risk profile they accept when selecting a guard relay.

Our work has implications beyond the browser. The 72KB minimal core we developed for embedded systems addresses a gap in medical device cybersecurity that, to our knowledge, has not been previously identified in the literature: current threat models comprehensively protect data content via TLS while leaving network metadata entirely exposed. The FDA's June 2025 guidance mandates protection of "any data whose disclosure could lead to patient harm," and we have shown that patient IP addresses correlated with device types fall squarely within this scope. tor-wasm provides a technical mechanism for manufacturers to close this gap.

More broadly, we believe the browser should not be an anonymity-free zone. The techniques documented here—JavaScript-bridged time access for TLS validation, WebCrypto integration for randomness, WebSocket tunneling for TCP protocols—may benefit other privacy tools targeting the web platform. As more sensitive operations move to web applications, the ability to selectively anonymize specific requests becomes increasingly important.

We release tor-wasm as open-source software at `[URL removed for anonymous review]`.

Acknowledgments

[Removed for anonymous review]

References

[1] AAMI. 2016. TIR57: Principles for Medical Device Security—Risk Management. Association for the Advancement of Medical Instrumentation. Technical Information Report on medical device security risk management.

[2] John Patrick Bland. 2023. Minitor: An embedded version of Tor for the ESP32. https://github.com/jpbland1/minitor. C implementation targeting ESP32 microcontrollers with 320KB RAM.

[3] Briand. 2024. toresp32: Tor client for ESP32 microcontrollers. https://github.com/briand-hub/toresp32. C++17 Tor proxy supporting 8 simultaneous circuits on ESP32.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, 303–320.

[5] David Dittrich and Erin Kenneally. 2012. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. Technical Report. U.S. Department of Homeland Security.

[6] Nathan Freitas. 2021. Home Assistant Tor Integration for IoT Privacy. https://blog.torproject.org/quick-simple-guide-tor-and-internet-things-so-far/. Tor onion service integration for smart home devices via Raspberry Pi.

[7] Guardian Project. 2023. Arti, next-gen Tor on mobile. https://guardianproject.info/2023/03/04/arti-next-gen-tor-on-mobile/. Early test builds of Arti on Android and iOS. Demonstrates native mobile embedding via Rust FFI bindings.

[8] iCepa Project. 2024. Tor.framework: Embed Tor in Your iOS Application. https://github.com/iCepa/Tor.framework. Compiles static C Tor for iOS/macOS. Supports arm64 and x86_64.

[9] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. 2013. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 337–348. https://doi.org/10.1145/2508859.2516651

[10] Seungju Lee, Mona Wang, Watson Jia, Qiang Wu, Henry Birge-Lee, Liang Wang, and Prateek Mittal. 2026. QUICstep: Evaluating Connection Migration Based QUIC Censorship Circumvention. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, Vol. 2026. https://arxiv.org/abs/2304.01073 To appear.

[11] libtor-sys contributors. 2024. libtor-sys: Rust Bindings to the Tor C Library. https://crates.io/crates/libtor-sys. Compiles C Tor as Rust crate for iOS/Android where process spawning is restricted.

[12] Nick Mathewson. 2013. *The ntor Handshake*. Tor Tech Report. The Tor Project. Proposal 216.

[13] Steven J. Murdoch and George Danezis. 2005. Low-Cost Traffic Analysis of Tor. In *IEEE Symposium on Security and Privacy*. IEEE, 183–195. https://doi.org/10.1109/SP.2005.12

[14] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2018. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 1962–1976. https://doi.org/10.1145/3243734.3243824

[15] Aymeric Nicolás. 2014. node-Tor: JavaScript Implementation of the Tor Protocol. https://github.com/AyMs/node-Tor. Abandoned circa 2014. Claimed browser

support but incomplete; predates WebCrypto API standardization.
[16] Erik Rye and Dave Levin. 2024. Surveilling the Masses with Wi-Fi-Based Positioning Systems. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, Vol. 2024. 550–567. https://doi.org/10.56553/popets-2024-0091 Demonstrates mass surveillance via public Wi-Fi positioning APIs.
[17] Brian Smith. 2024. ring: Safe, fast, small crypto using Rust. https://github.com/briansmith/ring.
[18] Subgraph. 2013. Orchid: A Pure Java Tor Client. https://github.com/subgraph/Orchid. First complete Tor protocol reimplementation (not wrapping C Tor). Version 1.0 released November 2013. Deployed in Martus human rights application.
[19] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. 2001. Towards an Analysis of Onion Routing Security. In *International Workshop on Designing Privacy Enhancing Technologies*. Springer, 96–114. https://doi.org/10.1007/3-540-44702-4_6
[20] Péter Szántó. 2020. go-libtor: Self-contained, statically linked Tor library for Go. https://github.com/ipsn/go-libtor. CGO wrappers around C Tor. Demonstrated Android deployment.
[21] The Tor Project. 2024. Snowflake. https://snowflake.torproject.org/. Accessed: 2024-12-01.
[22] The Tor Project. 2025. Arti: A Rust Tor Implementation. https://gitlab.torproject.org/tpo/core/arti. Version 1.0.0 released September 2025. Designed for embedding but requires tokio runtime and OS sockets.
[23] Chad Retz Tulloch. 2024. Bine: Go Library for Accessing and Embedding Tor. https://github.com/cretz/bine. Supports Tor Control Protocol and statically compiled Tor embedding. Used by 785+ projects.
[24] Lindsey Tulloch and Ian Goldberg. 2023. Lox: Protecting the Social Graph in Bridge Distribution. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, Vol. 2023. 608–631. https://doi.org/10.56553/popets-2023-0029
[25] U.S. Department of Health and Human Services. 2024. Summary of the HIPAA Security Rule. https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/. Health Insurance Portability and Accountability Act security requirements.
[26] U.S. Food and Drug Administration. 2025. Cybersecurity in Medical Devices: Quality System Considerations and Content of Premarket Submissions. https://www.fda.gov/media/119933/download. Document GUI00001825. Final Guidance issued June 27, 2025. Supersedes September 2023 guidance. Implements Section 524B of FD&C Act (FDORA 2022).

A Tor Cell Format

All Tor cells are exactly 514 bytes: 4-byte circuit ID, 1-byte command, 509-byte payload. Variable-length cells (VERSIONS, CERTS) use a different format with explicit length field.

Cell types implemented:

Type	Cmd	Purpose
VERSIONS	7	Negotiate link protocol
CERTS	129	Relay certificates
AUTH_CHALLENGE	130	Optional client auth
NETINFO	8	Timestamps & addresses
CREATE2	10	Initiate ntor handshake
CREATED2	11	Complete ntor handshake
RELAY	3	Encrypted relay cmds
RELAY_EARLY	9	EXTEND commands only
DESTROY	4	Tear down circuit
PADDING	0	Traffic padding

B JavaScript API

```
// Initialize client with bridge URL
const client = await new TorClient('wss://bridge.
    example.com');

// Connect to Tor network (fetches consensus)
await client.bootstrap();

// Build a 3-hop circuit
await client.build_circuit();

// Fetch URL through Tor
const response = await client.fetch('https://
    example.com');
console.log(response);  // HTML content

// Clean up resources
client.free();
```

C Ethical Considerations

Anonymity tools have dual-use potential. We follow the Menlo Report ethical principles [5]:

- **Beneficence**: tor-wasm enables legitimate privacy use cases (journalists protecting sources, activists organizing, ordinary users avoiding tracking)
- **Respect for persons**: Users choose when to use anonymity; it's not imposed
- **Justice**: Released as open source, available to all
- **Respect for law**: tor-wasm does not bypass access controls or enable new illegal activities beyond what existing Tor enables

tor-wasm does not create new attack capabilities. Malicious users already have access to Tor Browser and command-line tools. Our contribution is making anonymity more accessible for legitimate use cases.