

Groundwire: Decentralized Sybil Resistance for a Dead Internet

Evan Forman `evan@groundwire.io`
Jake Hamilton `jake@groundwire.io`
Trent Steen `trent@groundwire.io`

Groundwire Foundation

Abstract

Public-key infrastructures have hitherto assured us that a message comes from whoever it says it does, but we contend that this does not give us secure, sybil-resistant networking at the current level of LLM capability and proliferation, let alone post- or even near-AGI conditions. Once bots represent a majority of internet traffic we have to ask, who is trying to send us this message? As LLMs become capable of sophisticated social engineering attacks we must ask, is the sender of this message really who they say they are? And as AI drives the price of information production to near-zero, we ask: at what cost? This article introduces a decentralized, sybil-resistant identity and peer-to-peer networking stack which we design for an internet where the overwhelming majority of participants are autonomous AI agents.

Contents

1 Introduction

2

2	Mnemonyms	5
2.1	Multi-line layouts	7
2.2	Wordlist construction	8
3	Groundwire ID	9
3.1	Personally identifiable information	10
3.2	PKI metadata	10
3.3	PKI evidence	11
3.4	Urbtc implementation	11
3.5	Confidential comets	13
3.5.1	Name commitment	14
3.5.2	Onchain attestation	15
3.5.3	Offchain revelation	16
4	Wire URIs	17
4.1	Unsigned wires	18
4.2	Signed wires	18
5	Future Work	21
5.1	Named data networking	21
5.2	Post-quantum signatures	23
5.3	Hierarchical identity	25
6	Conclusion	26
7	Appendices	26
7.1	Ossification	26
7.2	Consensus	30

1 Introduction

Spam prevention efforts by legacy web platforms have failed to prevent the popular adoption of “dead internet theory”—that most visible internet activity is from bots—and an incipient retreat from the public web into closed (if not exactly private) online spaces. Everywhere you go online, there you are: ticking a box to prove you are human. Public discourse and culture at large suffer as the information environment is increasingly beset by engagement-farming bot accounts and increasingly

automated, agentic pipelines for state and non-state manipulation of target demographics (Lumbaca, 2025).

Spam prevention on these platforms is stymied by fundamental conflicts of interest: as AI agents become more capable, it becomes harder to automatically discriminate between human and nonhuman input, so human users suffer more onerous verification burdens. This runs counter to the platforms' incentive to attract and retain as many users for as much time-spend as possible, as well as their incentive to report as many impressions to creators and advertisers as possible.

How these platforms would adapt to an internet where an increasing supermajority of participants are AI agents is anyone's guess: SIM verification and photo ID laws such as those in Japan and Australia just increase operating expenses in the seller's market for fake accounts. Fake accounts tied to phone numbers in those markets currently sell for an average \$4.93 (Japan) and \$3.24 (Australia) per account in the same bulk volumes as cheaper \$0.10 (UK) and \$0.26 (US) accounts, as sellers pass SIM costs to buyers, with prices for accounts on Telegram and WhatsApp rising up to 15% in the lead-up to national elections (University of Cambridge, 2025).

The spam resistance stack we outline here is general purpose: any platform could implement some or all of it. Our interest lies not in redeeming Web 2.0 from its malign incentives, nor do we seek a return to a prelapsarian "open web" whose very openness exposed it to predation, centralization, and enclosure (Davis, 2022).

This article introduces three separable but complementary layers of a new decentralized identity and peer-to-peer networking stack:

1. Mnemonyms §2, a BIP-39-style (2013) mnemonic system for reasonably short, human-meaningful references to public keys.
2. Groundwire ID §3, a protocol for unique, persistent cryptographic identity across key rotations, tied to public signatures.
3. Wire §4, a URI scheme for signing data on arbitrary namespaces, securing it against the value of the signer's

Groundwire ID.

In the Future Work section §5 we outline near- to medium-term projects including named data networking secured against Groundwire ID, post-quantum signatures, and derivative identities for AI agents and networked devices.

Appendices §7 detail our thinking around two key issues for the software ecosystem we anticipate: security by planned ossification, and trustless consensus mechanisms.

Groundwire is in part an argument about structure: we contend that the collective cultivation of a global filesystem-like (or, “tree-shaped”) namespace where end-users are cryptographically sovereign root nodes can deliver a more effective sybil resistance regime than all mainstream solutions—CAPTCHA, SIM/ID/Email verification, blacklisting IP ranges, etc.—which are isolated, interactive, “conversation-” or “connection-oriented” exchanges, as opposed to our “data-oriented” (as in Named Data Networking, Blackman et al., 2026) approach to sybil resistance.

Materially, Groundwire’s first implementation §3 requires a one-time proof-of-work action from the creator of a self-custodied keypair to attest to owning that keypair on the Bitcoin blockchain. This sybil resistance “challenge” only happens once, and is cashed out in every network interaction that involves the user sending an effectful write or an access-controlled read §5.1. Server operators may layer further automated checks on top of this, but no manual sybil resistance checks should be necessary to prove that the end-user is not a disposable botnet identity. If servers want to verify that the user is a human and not a benign AI agent, that’s their business; the protocol does not discriminate.

A client may reconcile the namespace from multiple sources if it is not already unified in-situ. In the sybil resistance case they may apply policy to logical subtrees, analyse node provenance across space (permissions, derivative agent identities §5.3) and time (key rotations, verifiable onchain history §3.5.2). When this exact same filesystem addresses data storage, financial transactions, and networking traffic §5.1, the design space for simple and efficient (so, decentralizable) sybil

resistance tooling is wide open. Any subtree in the namespace can be summed into some quantity of trust-value over and above the total financial, computational, and/or thermodynamic cost of creating the nodes in that subtree, where every node in the subtree is ultimately secured against the proof-of-work value of the keypair that created it.

We take care from the outset to support interoperability between software that uses Mnemonyms, Groundwire ID, and Wire URIS, preferring to leave composition of that software to developers and end-users.

The sections below are introductory snapshots of works in progress, and not full specifications. The Groundwire Foundation has implemented these protocols in public alpha software, and everything described below is subject to change before production launch.

2 Mnemonyms

The mnemonic scheme is a general-purpose method for generating mnemonic references to binary strings. We use these as reasonably short, human-meaningful references to public keys on a PKI such as Bitcoin, Nostr, and Groundwire ID.

The mnemonic design is greatly inspired by BIP-39. Like BIP-39, a mnemonic wordlist is an alphabetically-sorted list of 2,048 words. Each word in the string represents 11 bits in a bit-string of entropy ENT with an appended checksum of $bitlength(ENT/32)$.

Two key differences from BIP-39 mnemonics:

- We do not expect mnemonic wordlists to be embedded in crypto hardware wallets in the base case; as a result, we grant a proliferation of wordlists for various languages, communities, and personal use-cases.
- Mnemonic strings represent an entire series of leading null words by omission; non-leading null words are represented normally, with whichever word is at index 0 in the list.

By omitting leading zeroes, we allow for reasonably short

mnemonyms (or “nyms”) to be distributed to users according to whichever scheme the underlying PKI permits. Such schemes may include simple first-come-first-serve domain registration, digital feudalism (Yarvin, 2010a), or proof-of-work mining. In a centralized, permissioned, or feudal system the length of the nym is useless as a market signal in and of itself: you must understand its provenance to find out how it was attained and at what cost, if any. In a permissionless proof-of-work system, the length of the nym approximately illustrates the thermodynamic cost at which the keypair must have been discovered and acquired, whether the holder of the nym mined it on their own machine or purchased it from a miner.

Our initial English wordlist §2.2 consists exclusively of iambs. For example, the pubkey...

```
00000000000ba839b38bcfd68037176c
```

would be encoded as...

```
.preserves.astounds.regrets.ensoul
.repeats.about.receive.repulsed
```

We reserve five visual representations of nyms at the protocol level:

- Bare nyms, e. g.,
 abet.baboon.caffeine.denounce.escape
- One-dot nyms, e. g.,
 .abet.baboon.caffeine.denounce.escape
- Two-dot nyms, e. g.,
 ..abet.baboon.caffeine.denounce.escape
- “Foreshortened” nyms of a string’s first, second, penultimate, and final words, e. g.,
 .abet.baboon..denounce.escape
 ..abet.baboon..denounce.escape
- “Abridged” nyms of just its first and last words, e. g.,
 .abet...escape
 ..abet...escape

Applications must reserve these as follows:

- One-dot for nyms which they have verified as being on the relevant PKI.

- Two-dot for nyms which they have either confirmed are not on the PKI, or for disposable references they do not expect to correspond to user identities on the PKI (e. g. block hashes, content hashes, receiving addresses, etc.).
- Bare nyms are reserved for intermediate contexts where a sender and receiver may or may not respect the same PKI (e. g. Wire URIS §4).

“Foreshortened” and “abridged” nyms may be used however the application developer sees fit, though abridged nyms must not be used in contexts where user funds may be at risk, to mitigate against address poisoning.

These are the only nym representations we formally bless. Informally, we grant that pseudonymous users may come up with various colloquial, context-sensitive short-name conventions to refer to themselves and each other, such as “.abet.escape”, “caffeine.chanteuse”, “.eclipse”, “eclipse”, etc. We expect developers will layer conventional “contact book” or “display name” features on top of mnemonyms at some point, obviating their use in online social contexts beyond first contact or peer-to-peer financial transactions.

The iambic English wordlist and libraries in C, Python, and Hoon are available at [gwbtc/mnemonyms](https://github.com/gwbtc/mnemonyms). Our C library is implemented in [gwbtc/comet-miner](https://github.com/gwbtc/comet-miner). The iambic English wordlist is still a work-in-progress and likely to change.

2.1 Multi-line layouts

Application developers may lay the nym out across several lines, e. g. in a UI context specifically designed to help the user memorize a nym. We encourage the following layouts specifically:

Twelve-word nym on three lines of four words:

```
.eclipse.betray.escape.delight
.mankind.alone.delay.ignite
.implode.elect.enclose.alas
```

Ten-word nym on two lines of five words:

```
.tattoo.giraffe.kazoo.champagne.balloon  
.auteurs.ablaze.admire.untamed.display
```

Nine-word nym on three lines of three words:

```
.raccoon.beret.lagoon  
.typhoon.guitar.chateau  
.concur.exempt.vacate
```

Eight-word nym on two lines of four words:

```
.disgrace.expired.infest.reject  
.doubloon.retells.entice.retreat
```

2.2 Wordlist construction

A nym wordlist may conform to any particular aesthetic or none at all. Wordlists should aspire to make resultant nyms as memorable, legible, and easy to work with as possible.

In our iambic English wordlist, we achieve this like so:

- Every word is a two-syllable iamb, which enhances memorability in that it reduces the number of words that could possibly follow another without reference to the wordlist.
- Words have no punctuation or diacritics in them.
- We limit the number of words that share common four-letter prefixes like “over-”, “semi-”, and “tran-” for the sake of autocomplete features in text inputs.
- No two words are exact homophones.
- We err on the side of words which are relatively common.
- Many words are plurals or verbs that end with ‘s’, so we limit words beginning with ‘s’ to avoid confusion in speech.
- We err on the side of words which result in absurd, concretely visual combinations inspired by memory palace techniques, though it should be just as easy to come up with a nym that is beautiful as one that is humorous.

3 Groundwire ID

Groundwire ID is a platform-agnostic protocol for persistent decentralized identities belonging to humans and autonomous AI agents.

The user is assumed to hold a BIP-32 (2012) HD crypto wallet that can derive an Ed25519 keypair whose pubkey will be encoded as a mnemonic. (Groundwire Foundation software uses the iambic English wordlist, but no wordlist is specified at the protocol level. Applications may use whichever wordlist they choose or let the user configure one.)

In practice, Groundwire ID is a protocol for tweaking pubkeys with publicly-verifiable evidence of their sybil resistance value, where we take the position that “sybil resistance value” is mostly financial/thermodynamic cost plus supplementary analytics subject to some policy configured by the user that determines who can send packets their way §3.5.3.

This tweak has two components: PKI metadata, and a pointer to publicly-verifiable evidence on that PKI. As we outlined in UIP-0136 (2026), the tweak is a “bidirectional cryptographic commitment between onchain and offchain attestations to prevent replay, spoofing, and double-spawning attacks. If there wasn’t an offchain anchor pointing to onchain data, then anyone could come in and attest to ownership of any comet and %ord-watcher would naively believe them.”

```
// pseudocode to produce a Groundwire ID tweak
(concatenate [(length-encode 'urbtc')
              [(length-encode 8)
               (length-encode
                (serialize pki-evidence))]]])
```

Where `(concatenate ...)` concatenates an array of `[bitwidth bitstring]` pairs. This produces a bitstring which is hashed into the pubkey to couple it to the PKI evidence.

```
// pseudocode to apply the tweak to a pubkey
(sha256 pubkey
 (concatenate [(length-encode 'urbtc')
               (length-encode
                (sha256
                 (concatenate [(length-encode 'urbtc')
                               [(length-encode 8)
                                (length-encode
                                 (serialize pki-evidence))]]])
                 pubkey)]])
```

```
[(length-encode 8)
 (length-encode
  (serialize pki-evidence))]]))
```

The first two items in this array are the PKI metadata §3.2, and the third a link to PKI evidence §3.3.

3.1 Personally identifiable information

Notably, Groundwire ID does not imply that a verified nym correlates to a human person. (We assume an internet where 99.9% of users are autonomous AI agents, an addressable market easily served by the 256-bit namespace that Ed25519 nyms imply and plausibly served by a 128-bit namespace §5.3.)

It *could* verifiably correlate to a human, and we are interested in exploring permissionless ZK solutions to couple Groundwire IDs to personally identifiable information without exposing that PII to anyone. A bridge towards PII in the general case and government-backed ID in particular is one direction in a greater design space: What can we cryptographically attest about an identity? And how can we compose those predicates into subprotocols useful for a variety of autonomous network actors within the bounds of a simple, freezable format?

3.2 PKI metadata

A PKI may be one of several on the same technical substrate, e. g. the same blockchain. The tweak names this PKI by a string of `^[a-z][a-z0-9-]*$`. In the `(length-encode 'urbtc')` example above, the name of the PKI is “urbtc”, which is Groundwire’s Urbit PKI on Bitcoin. Verifiers will resolve this name however appropriate: in urbtc, the PKI name is also the name of the local application an Urbit node will use to verify identities on that PKI, which may be one of several PKIs they follow.

The other piece of PKI metadata included in the tweak is `(length-encode 8)`, which in this case tells us that the pubkey was tweaked against urbtc version 8K. PKIs may support some kelvin versions §7.1 and not others, and may have their own control flow for handling tweaks on old versions.

One purpose of this is that it namespaces Groundwire IDs across PKIs: the same inputs will not generate the same Groundwire ID on two different PKIs, even if they're on the same substrate. The probability of the same Groundwire ID (that is, the same verified nym) appearing on two PKIs is astronomically tiny. This improbability leaves open a design space for cross-PKI interoperability, e.g. “teleburning” Ethereum assets onto Bitcoin Ordinal inscriptions (Rodarmor, 2026), but at time of writing we have no plans to pursue this.

3.3 PKI evidence

The second part of the tweak is some publicly-verifiable data on the PKI for a verifier to do two things: 1) verify the existence of a peer on the PKI and 2) judge the sybil resistance value of that data, according to some policy hard-coded in the verifier or set by the user.

The content of the pki-evidence in the (length-encode (serialize pki-evidence)) pseudocode above will vary by PKI. For example, in urbtc §3.4 the PKI evidence is the relative ordinal number (Rodarmor, 2022) of a sat in a UTXO belonging to the Bitcoin wallet that generated the Groundwire ID in question. We avoid using absolute ordinal numbers for performance reasons.

Notably, we do not include key rotations here or anywhere in the tweak, because doing so would change the resulting nym every key rotation. Reconciling the history of an identity across key rotations is the job of the PKI. Key rotations should generally not be considered to “reset” the sybil resistance value of an ID: on a blockchain where operations require transaction fees paid to infrastructure providers, an ID should accrue sybil resistance value over successive key rotations and miscellaneous transactions.

3.4 Urbtc implementation

The centerpiece of our public alpha is urbtc, a protocol for maintaining an Urbit PKI on the Bitcoin blockchain. Urbtc only supports Urbit “comets”—nodes in Urbit’s self-signed 128-bit

namespace—which can be spawned by anyone and have no speculative value. Ordinarily, comets are reliant on one of Urbit’s 256 “galaxy” root nodes to do peer discovery and routing, and the comet’s galaxy is hard-coded by the last 8 bits of their ID. Groundwire allows any comet to serve as a root node by attesting to a stable IP address onchain.

An urbtc tweak looks like this:

```
// pseudocode
(concatenate [(length-encode 'urbtc')
              [(length-encode 8)
               (length-encode
                (serialize spawn-sont))]]])
```

Where the `spawn-sont` is a tuple of `{txid vout offset}`:

- `txid`: ID of a Bitcoin transaction whose outputs contain the relevant UTXO.
- `vout`: the relevant UTXO’s position in a 0-indexed array of the “vector of outputs” from this transaction.
- `offset`: relative ordinal number of a sat in this UTXO; the Groundwire ID is pinned to this sat, which we will follow through successive UTXOs.

To facilitate comets attesting to their identities via urbtc, we introduced Groundwire’s notion of multiple PKIs to Jael, along with vane tasks for registering and suspending PKIs. Each corresponds to the name of a Gall agent permissioned to update information for one PKI (pending userspace permissions work currently underway). We additionally introduced a “Suite C” cryptographic core utilized by Groundwire comets, detailed in [UIP-0136 \(2026\)](#).

When a ship receives an introductory “self-attestation” packet from a Suite C comet, the comet includes the PKI data needed for verification. Ames then passes this data to Jael and subscribes for a verdict on whether the self-attestation is valid. Jael passes the data to the corresponding agent for that PKI, and upon receiving a positive update sends a fact to Ames along the relevant subscription, and Ames completes the handshake appropriately. From here, the userspace agent can continue to

watch the PKI substrate for further key rotations for that comet.

Due to the confidential nature of key rotations on urbtc, userspace agents only know upon seeing a key rotation that that identity's old keys are now out-of-date; they do not know what the new networking keys are until receiving a new self-attestation packet from that identity. The Urbit ship must, then, ignore all new packets from that identity except for self-attestation packets. We introduce the %snob task to Ames (as a counterpart to %snub) which is used to put an identity into this limbo state.

In addition to your sponsor—which maintains contact with your ship for relay and route rediscovery from peers if your node is behind a WiFi router—we extended the routing hint options in \$point:jael with a \$fief, an IPv4, IPv6, or DNS static transport address that may be used for route selection in the kernel and which we attest in onchain routing state on Bitcoin. Any node that has such an address can straightforwardly sponsor and route for comets running on mobile or a home server, and optionally supply value-add services, so we're inclined to grow this set early to enable a robust network topology and incentivize development of QoS protocols.

Urbtc is implemented across gwbtc/urbit, gwbtc/vere, and gwbtc/groundwire. We aim to upstream our Arvo and Vere work into the corresponding urbit/urbit and urbit/vere repos.

3.5 Confidential comets

Our initial tweak was designed for a version of the urbtc protocol that used ordinal inscriptions (Rodarmor et al., 2026). Inscriptions serialize data as chained OP_PUSH opcodes of a utxo's Taproot script in a “commit” transaction, then a “reveal” transaction makes that hashed onchain data legible as a plaintext MIME response. This required all Urbit nodes to index all Bitcoin blocks to find urbtc reveal transactions and use those to maintain their local PKI state.

In testing we found that the computational burden of this indexing was trivial for Groundwire nodes on a commercial vps and the developers' own laptops. The main issue was that

nodes could take hours to catch up with their peers, which we ameliorated with a stopgap trusted snapshot system for onboarding. This helped testers by skipping an initial sync from our chosen block 943,140 to the present, but we couldn't help them speed up recovery from unplanned downtime because Urbit doesn't allow users to overwrite PKI state outwith Jael's API. This snapshot system was just a convenience, and injecting a null value where the snapshot should have been would trigger re-indexing of the chain from block 943,140.

However, this question of using Bitcoin as arbitrary data storage has been a source of long-running debate in the Bitcoin community, citing a concern for keeping the burden on Bitcoin node operators as low as possible (BIP-0110, 2025). Making all PKI state public onchain is also a privacy concern, even if the data is apparently trivial.

For these reasons, we implemented a system of “confidential comets” by, for a start, simply omitting the “reveal” transaction for attestations. Instead, we have comets reveal their PKI history offchain as part of the initial handshake with a new peer. As of 8K, the three data commitments that comprise a Groundwire ID attestation have the following shapes.

3.5.1 Name commitment

The keypair underlying the comet's name commits via Schnorr tweak (scalar addition) to [%urbtc spawn-son], which is a PKI tag and a unique chainstate pointer binding one name to one sat. (No other name may be bound to this sat.)

The %urbtc tag is a Hoon text constant that names our PKI in Urbit's Jael kernel module. The sat is addressed relative to its host UTXO at comet generation time rather than by a global sequence number as in the ordinals protocol, due to heavy indexing requirements, but we retain ordinals' first-in-first-out convention for ordering input sats over transaction outputs when following a bound sat through its subsequent spends.

3.5.2 Onchain attestation

Subsequent UTXOs containing the bound sat are used to permanently attest the comet name and various PKI state commitments beginning with the spawn transaction, which is a spend of the original host UTXO which we create following key generation and sat selection.

This bidirectional cryptographic binding between the offchain (name) and onchain (wallet) signers, plus the tweak's PKI separator prefix, prevents "double-spending" of a spawn transaction across PKIs, as well as some potential replay attacks within Bitcoin.

The base onchain "confidential attestation" is itself also a Schnorr tweak, in Pay-to-Taproot (P2TR) output pubkeys. The scalar we add to the wallet keypair to derive the P2TR key is a Merkle root committing to the serialized Nock noun containing the comet name, its life (key rotation) and rift (continuity-breaking key rotation), latest networking pubkey, routing hints, and at least one of a sponsor (Urbid ID) or a fief (IP/DNS).

For an output hosting one ID and bound sat, its PKI state is a Merkle leaf at axis 2 (first left child), and currently this is the only attestation type in the protocol. Batch custodial outputs containing multiple bound sats ("shared UTXOs") aren't yet supported on the alpha network, but we're exploring the design space particularly with respect to potential L2 architectures. So except in case of a breach (Urbid's continuity-breaking key rotation) the state hash is the root hash which is the scalar tweak. We reserve the right-hand subtree for future protocol expansion to support other attestation use-cases, such as display names and profile images, external links, web-of-trust links, code signatures, and proof of funds.

At spawn the initial networking key is the (preimage of the) name itself, but subsequent state commitments include both the name and a distinct latest messaging pubkey. Every time an output bearing the sat is spent, we require the new UTXO containing it to commit the current tip of its PKI log, even if the spend is simple custody-management e.g. transferring to a fresh address, and the committed state duplicates the previous

tip. Otherwise, the vocabulary of update types that we support is 1) rotate networking key 2) update routing info 3) publish an attestation and 4) continuity breach. Continuity breaches are signalled by the presence of a 0-valued second leaf at axis 3.

Plaintext publication of PKI data is an opt-in second layer “public attestation” in an added OP_RETURN output from the published state transaction. This serves two purposes: 1) permissionless advertising and discovery of public bootstrap sponsors, and 2) peer-recovery-of-last-resort for ships partitioned from each other by a mutual routing failure (e.g. extended sponsor downtime).

Note that we did not specify any tapscript encoding (Bitcoin Script in Merkle leaves that a P2TR spend can execute with an inclusion proof) of the protocol’s Merkle data above. That would be necessary if we wanted to publish data in the transaction witness to take advantage of the Segwit fee discount, as in the ordinals inscription mechanism. However, witness inscriptions bypass a Bitcoin node operator’s ability to filter the data without foregoing witness verification of signature operations as well, and the Segwit discount is actively harmful in the Groundwire setting since our sybil resistance mechanism *is* Bitcoin-denominated costliness.

Non-tapscript Merkle data commitments in the output chain are, instead, verifiably revealed by an OP_RETURN output that includes the internal (pre-tweak) keys of previous outputs and their signatures along with the state history. This is the same format as our offchain revelation in the peer attestation packet, detailed below.

In the taxonomy of Bitcoin Layer 2 constructs, urbtc is therefore a client-side validated (csv) protocol.

3.5.3 Offchain revelation

In Urbit’s peer-to-peer networking flow, comets introduce themselves offchain by exchanging attestation packets with a peer, and provide a refreshed packet when they broadcast new confidential attestations of PKI state onchain.

Legacy comets simply sign a minimal handshake, sufficient to negotiate the per-peer ECDH channel, using the same key of

which their ID is a straightforward hash. Groundwire IDs instead sign the packet with their latest messaging key, *and* must reveal the plaintext tweak data their name committed to, plus all data required for the recipient's PKI agent to verify the signature, the validity of the key, and the validity of the name. If the PKI tag is recognized by the receiver's Jael, it will forward the packet to the responsible PKI agent and chain client for validation. Otherwise, it will treat the sender as a legacy comet and require it to be at life 1 and sign with its identity key. For urbtc comets, the required information is outpoints and block heights for the sender's entire onchain history, and plaintext state commitments with Schnorr inclusion proofs for their current rift.

By the time we get to a production launch, ships should be able to configure their own sybil resistance policies including minimum acceptable values for costliness or proof-of-funds needed to accept a peer. That any ship may have their own secret policy obscures the requirements that a bad actor needs to meet to spawn a usable batch of fake accounts for any one ship or range of ships under one policy. Since a negative policy decision prevents peer discovery from happening at all, ships may only ever further constrain whichever policy is applied by their sponsor. Sponsors might advertise these policies as a point of differentiation. Sponsees may route around their sponsors' policies by accepting a new peer's keyfile out-of-band, e.g. by scanning a QR code or tapping an NFC tag in-person.

Since the attestation packet format specifies the content, location, and encoding of confidential attestations, and is itself broadcast in a public attestation, we can use a single kelvin version on the packet to cover both layers of the protocol, while freezing the ID tweak-data format straight to oK. The attestation packet's current kelvin version is 8K.

4 Wire URIs

The Wire URI scheme enables Groundwire IDs to make cryptographic claims on other namespaces with minimal integration

between Groundwire and that namespace’s ecosystem.

There are two forms of Wire URI (or “wire”): signed, and unsigned.

4.1 Unsigned wires

An unsigned wire is of the form:

```
wire://id/tag/path
```

as in these examples:

```
wire://aback.baboon.cabal.enough/https/groundwire.io  
wire://abyss.conceal.decrypt/geo/38.89278,-77.04583
```

An unsigned wire’s `id` is a Groundwire ID in the form of a bare nym, since we can’t assume the signer and verifier exist on the same PKI. The tag (a string `^[a-z][a-z0-9-]*$`) refers to some protocol that a client may or may not know what to do with, and the path is any `/`-separated path that corresponds to a resource the client can locate via that protocol.

An unsigned wire may be generated by the client as a human-readable reference to a signed wire stored elsewhere in the client. Unlike signed wires, they do not self-describe a version number, so a client parsing an unsigned wire without reference to a corresponding signed wire would have to handle a case if the `/tag/path` format ever changed during a kelvin version decrement in the signed wire.

4.2 Signed wires

Signed wires are of the form:

```
wire://id/base64url
```

as in this example:

```
wire://abate.banal.campaign.debate.elite/8BEBAC9odHRw  
cy9leGFtcGxlLmNvbRam2y5UY08BaCbZDGFToaToJugs8  
BX_uE0s3Hy7M95LhTn21s3_Mg-wH9WG-CsUxm2JGWjAbou  
m7waoMSWfgc
```

Signed wires serve a similar function to JSON Web Tokens (RFC 7519, 2015), in that we can use them to non-interactively verify that the owner of some public key signed a given string. We use Wire URIs and not JWTs in part because Wire URIs have a human-readable trust root and always represent a path in a signed namespace beneath that root.

Signed wires only have one segment after the id: a Base64URL string that encodes a header and a payload.

The 12-bit header has this structure:

- **First 1 bit:** A 0 or 1 flag that tells us if the signer has signed a hash of the `/tag/path` string (1), or the `/tag/path` string concatenated with the response we will receive from that path (0). This flag bit is not subject to kelvin versioning.
- **Next 3 bits:** kelvin version number. Wires launch at 7K, or 111. Everything to the right of this is version-dependent.
- **Next 8 bits:** The bytewidth of the decoded `/tag/path` string, which is up to 256 ASCII characters.

The payload has this structure:

- **First 16 bits:** Key rotation number, an abstraction over whatever key rotation mechanism the PKI substrate uses. May not be 0, we assume the signer's first keypair is 1.
- **Next n bytes:** The plaintext `/tag/path` string, where n is specified in the header
- **Next 64 bytes:** 7K's EdDSA signature of a 32-byte SHA-256 of either `key_rotation || tag_path` or `key_rotation || tag_path || content_hash`, where `content_hash` is itself a 32-byte hash of the octet stream a request on this path will receive. What exactly goes in the octet string depends on the `/tag` protocol; the signer may assume that a relevant client knows what kind of response to expect for a given protocol.

Clients can verify a signed wire that has no content on receipt. If they receive a signed wire with content, they should reconstruct the content hash on receipt and verify that against the signature before further processing.

Since Groundwire ID persists across key rotations, the client must verify the signature using the public key that corresponded to the wire's id at the key rotation signed in the payload. Key rotation events should not invalidate signed wires from previous key rotations, as verifiers should be able to get the keypair that corresponds to the wire from historical state. That historical state may be public (e.g. onchain) or privately distributed among peers, relays, or other network providers. (PKIs such as urbtc may implement a special, continuity-breaking key rotation event that *does* invalidate previous signatures, upon which peers should treat all historical state for that ID as having been wiped. If an ID signed a wire at keypair 2 before and after the continuity breach, only the post-continuity-break keypair 2 should be under consideration by the verifier. All signatures from before the breach should be considered invalid.)

Some example use-cases for Wire URIs are as follows:

- /mailto: A business or government agency signs their email address in a public link.
- /https: A business signs links in their emails such that a Groundwire-aware email client can filter out phishing scams without reference to a trusted third party.
- /tel: A government agency signs their public-facing telephone numbers; they sign numbers they may call from such that a Groundwire-aware phone/VoIP client can verify the number is legitimate before it rings.
- /bitcoin: A Groundwire ID requests payment to a Bitcoin address that may or may not hold the Groundwire ID's corresponding sat, enabling multi-wallet setups. The URI may also include a requested payment amount.
- /wss: A WebSockets server provider links a server they run to their Groundwire ID, without necessarily having to run that server on their Groundwire node.
- /ssh: A vps host signs their host key fingerprint, supplementing trust-on-first-use for users who want to ssh into that server.
- /geo: A Groundwire ID signs coordinates of a geocache.

- `/ipfs`: A Groundwire ID signs some distributed content by its hash, linking the content to the ID without the ID user having to run their own storage.

At time of writing there is no canonical reference implementation of the Wire URI scheme, but it is used in the `gwbtc/chorus` repo to store references to advertised content. This may be replaced with a Kademlia DHT implementation which stores references as content hashes rather than Wire URIs, mostly to improve upon Chorus’s gossip protocol.

5 Future Work

Sections §2, §3, and §4 describe code which is running in a public alpha release. This section discusses three future projects which are most pertinent to the ideas in this article: named data networking between Groundwire IDs, post-quantum signatures, and “child” identities derived from root-level Groundwire IDs.

5.1 Named data networking

Alongside kelvin versioning §7.1, the other part of Urbit that we aim to lift out of that system is Directed Messaging (Blackman et al., 2026), colloquially called “Mesa” by Urbit developers.

We think Mesa’s named data networking (NDN) behavior is a natural fit for any network of cryptographically sovereign root nodes, whether those correspond to separate physical machines in a peer-to-peer network or as virtual nodes on a cloud server.

NDN is sometimes described with terms like “data-oriented” as opposed to “connection-oriented” TCP/IP, and one can think of Groundwire as a data-oriented sybil resistance as opposed to connection-oriented sybil resistance (CAPTCHA, JS challenge, etc.).

To pull Mesa out of Urbit means removing some of the assumptions around network topology. For instance, Groundwire has no concept of a hierarchy of “galaxy”, “star”, and

“planet” nodes; sponsor-sponsee relationships are formed between service providers and their users, and any comet can sponsor another.

We don’t assume that Groundwire nodes are running Urbit OS, though our Mesa implementation should work for those nodes. One of the main goals of our Mesa project is seamless peer-to-peer networking between non-Urbit clients and Urbit servers.

Other than the caveats above and some other details, all headline features of Directed Messaging should be implemented in a Mesa library as protocol invariants:

- All messages should be requests or authenticated responses carrying named data.
- Authentication should happen on packets and not sessions.
- Source-independent single hop routing should be sufficient for networking, which minimizes bookkeeping required for nodes.

Other features of Mesa, such as exactly-once messaging, may be configurable per networking flow depending on the stack the peer in that flow is running Mesa on. Details like cache management and congestion control are implementation-specific.

This has the benefits of NDN as typically described, such as more resilient routing in unpredictable physical network conditions and less reliance on single points of control. Directed Messaging’s referentially-transparent namespace, with Groundwire IDs at the root, has its own use-cases for the distributed networking cases we’re interested in:

- Every path is signed by its publisher and has a version number, and there is currently no way to request data at a path without knowing its version number, so unversioned imports are impossible.
- Referentially-transparent paths under identities with key rotation enable more long-lived authorities over links, and Mesa’s caching and relay properties trivialize distributed archives and other CDN-like setups.
- Cheap, mechanically trivial CDNs distribute load away

from content publishers. Effectful writes must be signed by the Groundwire ID that originally sent them via HMAC. The likelihood that a write or access-controlled read comes from a disposable ID in a botnet can be priced at the moment of receipt of the first packet: either the peer writing to us is known and validated, or we ask for a self-attestation packet that includes the PKI evidence described in §3.3.

- Publishers and relays don't need to verify connections because there are no connections. Reads to access-controlled paths have similar caching properties as anonymous reads on public paths.

Our Mesa release will be a formal spec and at least one open-source reference implementation, likely in C. Many implementation details are left as exercises for network developers, including:

- Peer discovery mechanisms (bootstrap nodes, DHTs, pre-distributed PKIs, etc.).
- Transport layers (Mesa over TCP, UDP, SCTP, etc.).
- Caching policies.
- Peer-to-peer economic incentives (congestion control, relays, CDNs, etc.).
- Interaction with anonymous requesters.
- Reputation/uptime scoring.

Our Mesa work is in a preliminary `gwbtc/libmesa` repo, which will be made public once it's ready for a public alpha release.

5.2 Post-quantum signatures

At this point in time Groundwire's persistent cryptographic identity across key rotations is aimed at networks like Nostr, which have no affordance for transferring identity in the event of a compromised private key. Key rotation is presently a narrow use-case that even diligent users should rarely have to go through.

However, cryptographically-relevant quantum computing could make key rotation a much more salient issue, even in

normal circumstances. (PQ keypairs in stateful schemes like XMSS and LMS are finite, enumerated trees of single-use keypairs; the number of keypairs in a tree may be as low as 2^5 or up to 2^{25} (Cooper et al., 2020).) With Google recommending migration to post-quantum cryptography by 2029 (Google Quantum AI, 2026), technical communities like Bitcoin are urgently weighing their options for migration of legacy wallets to PQ.

We use “pre-quantum” signatures for urbtc and Wire URIS at their current kelvin versions because this is the status quo in both Urbit networking and Bitcoin consensus. In particular, the Schnorr cryptography we base our overlay of the comet namespace on is high among the signature features that will be difficult or impossible to achieve in the known PQ algorithms that are feasible for Bitcoin.

Our strategy for an early, low-overhead migration precisely mirrors the BIP-360 “Pay-to-Merkle-Root” quantum resistance project (BIP-0360, 2024):

- Abandon offchain key tweaking.
- Breach comet semantics from pubkey hashes to pure hashes of structured data that attest multiple signature algorithms.
- Implement a range of high-attention open PQ signature schemes drawn from Bitcoin research and other networks.

We expect that efforts in that third area, implementing new signature schemes, are likely to lead and drive timing for the migration as a whole, but a major goal will be to improve modularity of cryptosystem adoption in the future as well as upgrading the core crypto suites.

Once Bitcoin developers and hardware providers settle on a PQ migration plan, we will develop a migration plan from pre- to post-quantum kelvin versions which preserve continuity of identity and attestations to the fullest extent possible.

5.3 Hierarchical identity

Groundwire ID's namespace has no outer limit: a nym can have as many words as you need. Urbtc's namespace can accommodate 2^{128} identities because each must correlate to an Urbit comet. Urbtc identities must also correlate to Bitcoin sats, of which there will only ever be 2.1 quadrillion in existence, with far fewer in circulation. How could one Groundwire PKI like urbtc scale to name all humans, devices, and AI agents on and near Earth?

As it stands a Groundwire ID is an Ed25519 keypair generated by an HD wallet, which gives us an obvious way to correlate "sibling" identities owned by the same user wallet. But it also allows urbtc to fill the remainder of the 128-bit namespace by allowing the 2.1 quadrillion onchain root nodes to continue the next level of their HD derivation tree and derive many "child" identities, each of which can likewise derive their own children, and so on.

This hierarchical recursion of the HD wallet tree-structure gives us a natural way to represent relationships of ownership, responsibility, and granular access control amongst human-driven identities (e.g. multiple signing devices belonging to one human) and AI agent identities (e.g. multiple agents belonging to one human, multiple subagents belonging to one agent, etc.). Since child keys are derived by tweaking the parent with an arbitrary nonce, we can use this to commit to descriptors for the role, origin, authority, etc. of the child, cryptographic authorizations like a signature over permissions, or even a discretionary spending budget in the form of Chaumian ecash signatures.

We haven't explored the design space for this yet, but for our purposes a system for deriving and verifying identities must satisfy these requirements:

1. Derived identities are trivially verifiable as belonging to their ultimate root identities and any parent identities in between.
2. Deriving an identity is mechanically simple, computationally cheap, and almost instantaneous; an AI agent can trivially derive their own identities, having gained

approval from its root identity via manual sign-off or hard-coded policy.

3. Freshly derived identities have no sybil resistance value of their own, instead backed by the sybil value of their ultimate root identity and subject to whichever calculation a verifier chooses.
4. Parent/child relationships can be expressed in the namespace and children may be identified via their parents; e.g. if AI agent `..denounce...escape` belongs to human operator `.abet...baboon`, they might be identified in a UI as `.abet...baboon::denounce...escape`, or `::abet...escape`, or some other representation. Namespaces belonging to a derived identity may be viewed as subtrees of their root identity’s namespace.

6 Conclusion

Groundwire is cool and you should check it out.

7 Appendices

The following appendices are supplementary notes on software ossification and consensus mechanisms, detailing the thinking behind some of our design decisions above.

7.1 Ossification

One of the premises of open-source software is that “given enough eyeballs, all bugs are shallow”. Looking at recent large-scale cyberattacks—like the 2025 “Shai-Hulud” attacks distributed via node package manager (npm), and possibly written by an LLM (Moore, 2025)—we anticipate a future where the majority of those eyes are fleets of fully-automated cyberattack agents with near- or super-human ability to discover vulnerabilities and exploit targets via malicious code and social

engineering. All automatic software update pipelines are attack vectors, and dependency managers such as npm might import dozens of untrusted dependency versions—recursively—at build time for even the smallest projects.

Conversely, the developers of these dependencies find that updates become more fraught the greater the number of dependents, such that even desirable updates become impossible in practice. This kind of unplanned ossification is almost always bad. Two software communities in which developers embrace planned ossification are Urbit, which formally counts down to an immutable OS kernel, and Bitcoin, where an informally-defined stable state is seen as part of the investment thesis.¹

In a 2017 blog post on `urbit.org` called “Towards a Frozen Operating System” (Yarvin and Wolfe-Pauly, 2017), Curtis Yarvin and Galen Wolfe-Pauly contended that “there is almost never any good reason to automatically update dependencies with fresh versions of foreign code. In the case of a security update, the maintainer of an app which depends on a patched library needs to catch the patch and propagate the update in a new version of the app. An app shouldn’t be security-critical to begin with, and propagating updates is a basic function of a modern OS. Also, if the app is security-critical, it could have its own bugs, so it needs a maintainer and an update pathway anyway.” In a 2020 `urbit.org` post (Monk, 2020) Urbit developer Philip Monk wrote that “system modules should be designed to be frozen permanently; conversely, Urbit consists of that system software which admits of a Kelvin version.”

What is a kelvin version? The 2016 Urbit whitepaper (Yarvin et al., 2016) doesn’t define it. At time of writing, no formal specification has been signed off on by the Urbit Foundation or Urbit’s core developers. We do find one subsubsection of informal description in Yarvin’s 2010 blog post *Urbit: functional programming from scratch*, which publicized Urbit under the name “Urbit” for the first time: “the true, Martian way to perma-freeze a system is what I call Kelvin versioning. In Kelvin versioning, releases count down by integer degrees

¹Jameson Lopp argued against an ossifying Bitcoin in 2024 (Lopp, 2024), but gave a generous summary of the pro-ossification arguments he’d encountered.

Kelvin. At absolute zero, the system can no longer be changed. At 1K, one more modification is possible. And so on.” (Yarvin, 2010b)

Today, Urbit’s OS is distributed at 408K. This version number tags the `zuse.hoon` file which defines Urbit’s kernel module APIs; the modules themselves are not kelvin versioned. Zuse is the “outermost” layer of Urbit’s OS, and deeper layers may have their own kelvin versions. (The Hoon standard library at 135K, for instance.)

This raises the question of kelvin-versioned dependency management. In “Towards a Frozen Operating System,” Yarvin and Pauly wrote:

The right way for this trunk to approach absolute zero is to “telescope” its Kelvin versions. The rules of telescoping are simple:

1. If tool B sits on platform A, either both A and B must be at absolute zero, or B must be warmer than A.
2. Whenever the temperature of A (the platform) declines, the temperature of B (the tool) must also decline.
3. B must state the version of A it was developed against.
4. A, when loading B, must state its own current version, and the warmest version of itself with which it’s backward-compatible.
5. Of course, if B itself is a platform on which some higher-level tool C depends, it must follow the same constraints recursively.

The important effect of obeying rule 2: in a tall trunk, it’s sufficient to state your dependency on the platform directly below you. Suppose C sits on B, which sits on A and exports features of A. C can state only the version of B it’s written against, because when A chills, B must also chill.

In practice, Urbit core developers update Zuse and its dependencies at once: every Urbit release is either a new Zuse

kelvin or a non-kelvin patch to the current distribution of Zuse. Any change to any kelvin-versioned file below Zuse will be slated for release in the next Zuse kelvin release. Significant changes to unversioned kernel modules usually go in the next Zuse release, but this is at the discretion of the Urbit Core Guild.

Today, Urbit userspace applications must declare the Zuse version they were built against. If the user is running Zuse 408K, an application update built against 407K will be received and stored for later. Once the user is ready to update to Zuse 407K, the app update is applied in the latter stages of an *atomic* upgrade of Zuse and its dependents, which will cleanly fail if even one dependent has no pending update to 407K. If the user wishes to update Zuse anyway, they can “suspend” the application to prevent it from running but hang onto its code and data in the event that a 407K update ever arrives.

The maintenance burden this places on Urbit userspace developers is nontrivial: most Urbit userspace code available online will not compile against Zuse 408K. This fact runs counter to one of the stated goals of kelvin versioning, which is perfect forward compatibility. Quoting “Towards a Frozen Operating System”:

C is a very mature language. C has long since mastered backward compatibility. We can be sure that any C program which compiles on today’s correct C compilers will compile on any future correct C compiler.

But, since C is not frozen, we can’t be sure that today’s C compilers will compile any future C program. So C has not yet achieved perfect forward compatibility. By definition, a frozen system is compatible both forward and backward. So C is mature, but it could still get more mature.

The point of a frozen OS is to shoot for infinite maturity. Obviously a frozen platform is compatible forever in both directions.

The Urbit Foundation is doing prototype work on “kelvin

shimming”, which allows Urbit to run userspace applications built against previous kelvins in their own sandbox, but what form that solution will take is still an open question.

7.2 Consensus

Groundwire is a protocol and has no opinions about technical substrates. The Groundwire Foundation has such opinions, and our choice of the Bitcoin blockchain as the substrate for urbtc is highly motivated by the same concerns that inform our protocol design.

The most important feature of Bitcoin for our use-case is the “longest-chain rule” in its consensus protocol. Per the Bitcoin whitepaper (Nakamoto, 2008), “proof-of-work ... solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-per-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-cpu-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it.”

A Bitcoin node accepts the longest proof-of-work chain as the canonical record of Bitcoin’s transaction history, without question. If one finalized chain is overwritten by a longer one, so be it. This simple decentralized governance mechanism has survived many bitter disputes in Bitcoin’s history. Amongst competing Bitcoin forks, a single chain holds onto almost all nodes due to simple network effects and the game-theory of proof-of-work: in 2014 Vitalik Buterin wrote that “when contributing one’s work to the blockchain, a miner must make the choice of which of all possible forks to contribute to (or whether to try to start a new fork), and the different options are mutually exclusive. Double-voting, including double-voting where the second vote is made many years after the first, is unprofitable [sic] since it requires you to split your mining power among the different votes; the dominant strategy is always to put your mining power exclusively on the fork that you think is most likely to win.” (Buterin, 2014)

Technical debates amongst Bitcoin developers, such as that around BIP-110 (2025), hold it as a priority that independent

and hobbyist node operators using consumer hardware should be able to feasibly replay the history of the chain to arrive at its current state. This property is essential for decentralization and it falls out of proof-of-work: on initial sync a node operator may choose to trust or verify previously-committed blocks via an `assumevalid` flag, but in any case they are replaying the longest proof-of-work chain upon the genesis block which serves as the trust root.

Quoting `UIP-0136`, “Groundwire’s thesis is pragmatic: over the long term, Bitcoin is the only chain that provides an unambiguous, protocol-level canonical history (the heaviest valid proof-of-work chain) that a Kelvin o Urbit could resolve without social coordination or trusted checkpoints. This suggests a clean long-term anchor for the Urbit PKI.”

Compare this to Ethereum’s proof-of-stake model. If an Ethereum node receives two finalized blocks that contradict each other, there is no programmatic fork-choice rule to choose between them (`ethereum.org` contributors, 2023). The node’s version of chain history must be re-derived from a “weak subjectivity checkpoint”: some block height acquired from a trusted third party which they claim to be a truth universally acknowledged by all node operators.

`UIP-0136` continues: “It is too early to propose that that belief be enforced by [Urbit] kernel policy ... Thus while Bitcoin’s popular ordinal/inscription protocol is a functional and easy-to-reason-about initial choice of onchain mechanism, it would also be premature to commit the kernel to these abstractions. Instead, Urbit should provide a general mechanism whereby comets can attest keys to any PKI source (chosen by a Jael %listen task) and peers can verify these attestations.”

Having enabled Groundwire users to compose multiple PKI substrates—at least one of which is Bitcoin, and several of which may be other blockchains—we have fallen through a trapdoor and landed in the “blockchain interoperability problem”. Groundwire ID’s name commitment is supposed to namespace PKIs such that two PKIs should never hold the same ID, but if we nonetheless receive a self-attestation packet from an ID on Solana claiming to be an ID on Bitcoin we are about to send money to, who do we trust? Which of our PKI gateways

is broken or malicious? Which record of history is correct? What are the semantics by which history is recorded, and how do we reconcile those histories if two consensus mechanisms use different semantics?

At this point we may fall back on human judgement, but Groundwire assumes that 99.9% of internet users are not humans and cannot be held accountable for their judgements.

If we grant that even Bitcoin's longest-chain rule is inevitably backed by the human drama of social consensus, institutional governance, conflicting incentives, and spirited public debate, we think this is all the more reason to secure those out-of-band techno-social systems against a flood of highly capable bad actors with kelvin-versioned software and sybil-resistant identity and networking. By hosting the PKI onchain, Bitcoin could secure the host environment in which its own developers, node operators, and other stakeholders work in public and come to important decisions about its future. This holds for proof-of-stake chains which are even more dependent on these out-of-band consensus mechanisms.

So a programmatic symmetry-breaking rule is desirable where we cannot reconcile the histories of two blockchains. If the chains in question are both Ethereum L2s, we wait for the transactions to settle on Ethereum; if Ethereum produces two finalized blocks, we phone a friend. The opening of the Bitcoin whitepaper gives us such a programmatic rule, which we may apply recursively upwards through increasingly meta-level conflicts amongst consensus mechanisms: "we propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers."

May the longest proof-of-work chain win.

References

- Beast, Hunter, Ethan Heilman, and Isabel Foxen Duke (2024) “Pay-to-Merkle-Root (p2MR)”. Consensus (soft fork), status Draft, version 0.12.0. Consensus (soft fork), status Draft, version 0.12.0. URL: <https://github.com/bitcoin/bips/blob/master/bip-0360.mediawiki> (visited on ~2026.9.2).
- Blackman, Ted, Joe Bryan, Luke Champine, Pyry Kovanen, Jose Cisneros, and Liam Fitzgerald (2026). “Directed Messaging.” In: *Orbit Systems Technical Journal* 3.1. URL: <https://urbitsystems.tech/article/v03-i01/directed-messaging> (visited on ~2026.9.2).
- Buterin, Vitalik (2014) “Proof of Stake: How I Learned to Love Weak Subjectivity”. Blog post, Ethereum Foundation. Blog post, Ethereum Foundation. URL: <https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity> (visited on ~2026.9.2).
- Cooper, David A., Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller (2020). *Recommendation for Stateful Hash-Based Signature Schemes*. NIST Special Publication 800-208. National Institute of Standards and Technology. doi: 10.6028/NIST.SP.800-208. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-208.pdf> (visited on ~2026.9.2).
- Davis, N. E. (2022) “Immunology for the Internet Age”. Blog post. Blog post. URL: <https://urbit.org/blog/immunology-for-the-internet-age> (visited on ~2026.9.2).
- ethereum.org contributors (2023) “Weak Subjectivity”. Ethereum developer documentation, Ethereum Foundation. Ethereum developer documentation, Ethereum Foundation. URL: <https://ethereum.org/developers/docs/consensus->

- mechanisms/pos/weak-subjectivity/ (visited on ~2026.9.2).
- Google Quantum AI (2026) “Safeguarding Cryptocurrency by Disclosing Quantum Vulnerabilities Responsibly”. Google Research Blog. Google Research Blog. URL: <https://research.google/blog/safeguarding-cryptocurrency-by-disclosing-quantum-vulnerabilities-responsibly/> (visited on ~2026.9.2).
- Jones, Michael B., John Bradley, and Nat Sakimura (2015) “JSON Web Token (JWT)”. URL: <https://datatracker.ietf.org/doc/html/rfc7519> (visited on ~2026.9.2).
- Lopp, Jameson (2024) “On Ossification”. Cypherpunk Cogitations. Cypherpunk Cogitations. URL: <https://blog.lopp.net/on-ossification/> (visited on ~2026.9.2).
- Lumbaca, Jeremiah (2025). *Cognitive Warfare to Dominate and Redefine Adversary Realities: Implications for U.S. Special Operations Forces*. Occasional Paper, JSOU Report 25-23. Joint Special Operations University Press. URL: https://jsouapplicationstorage.blob.core.windows.net/press/578/Cognitive%20Warfare_FINAL.pdf (visited on ~2026.9.2).
- Monk, Philip C. (2020) “Precepts”. Blog post, Tlon Corporation. Blog post, Tlon Corporation. URL: <https://urbit.org/blog/precepts> (visited on ~2026.9.2).
- Moore, Justin (2025) ““Shai-Hulud” Worm Compromises npm Ecosystem in Supply Chain Attack”. Unit 42, Palo Alto Networks. Unit 42, Palo Alto Networks. URL: <https://unit42.paloaltonetworks.com/npm-supply-chain-attack/> (visited on ~2026.9.2).
- Nakamoto, Satoshi (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. Whitepaper. URL: <https://bitcoin.org/bitcoin.pdf> (visited on ~2026.9.2).
- Ohm, Dathon (2025) “Reduced Data Temporary Softfork”. Consensus (soft fork), status Closed. Consensus (soft fork), status Closed. URL: <https://>

- [//github.com/bitcoin/bips/blob/master/bip-0110.mediawiki](https://github.com/bitcoin/bips/blob/master/bip-0110.mediawiki) (visited on ~2026.9.2).
- Palatinus, Marek, Pavol Rusnak, Aaron Voisine, and Sean Bowe (2013) “Mnemonic Code for Generating Deterministic Keys”. Standards Track, Applications layer. Standards Track, Applications layer. URL: <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki> (visited on ~2026.9.2).
- Rodarmor, Casey (2022) “Ordinal Numbers”. Draft BIP, number unassigned, Informational, Applications layer. Draft BIP, number unassigned, Informational, Applications layer. URL: <https://github.com/ordinals/ord/blob/master/bip.mediawiki> (visited on ~2026.9.2).
- Rodarmor, Casey et al. (2026) “Inscriptions”. Ordinal Theory Handbook. Ordinal Theory Handbook. URL: <https://docs.ordinals.com/inscriptions.html> (visited on ~2026.9.2).
- Rodarmor, Casey (2026) “Teleburning”. Ordinal Theory Handbook. Ordinal Theory Handbook. URL: <https://docs.ordinals.com/guides/teleburning.html> (visited on ~2026.9.2).
- Steen, Trent, Michael Bensemann, Evan Forman, and Jake Hamilton (2026) “UIP-0136: Comet Attestation”. Standards Track, status Review. Standards Track, status Review. URL: <https://github.com/urbit/UIPs/blob/main/UIPS/UIP-0136.md> (visited on ~2026.9.2).
- University of Cambridge (2025) “Price of a Bot Army Revealed Across Hundreds of Online Platforms”. Research news. Research news. URL: <https://www.cam.ac.uk/stories/price-bot-army-global-index> (visited on ~2026.9.2).
- Wuille, Pieter (2012) “Hierarchical Deterministic Wallets”. Standards Track, Applications layer. Standards Track, Applications layer. URL: <https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki> (visited on ~2026.9.2).
- Yarvin, Curtis (2010a) “Introduction”. URL: <https://github.com/cgyarvin/urbit/blob/>

- 6ac688960687aa9c89d4da6fff49a3125c10aca1/
Spec/urbit/3-intro.txt (visited on ~2026.9.2).
- Yarvin, Curtis (2010b) “Urbit: functional programming from scratch”. URL:
<http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html> (visited on ~2024.1.25).
- Yarvin, Curtis, Philip Monk, Anton Dyudin, and Raymond Pasco (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation. URL:
<https://media.urbit.org/whitepaper.pdf> (visited on ~2024.1.25).
- Yarvin, Curtis and Galen Wolfe-Pauly (2017) “Toward a Frozen Operating System”. Blog post, Tlon Corporation. Blog post, Tlon Corporation. URL:
<https://urbit.org/blog/toward-a-frozen-operating-system> (visited on ~2026.9.2).