Groundwire Comet Cryptography

Trent Steen ~hanfel-dovned
Groundwire Foundation

Abstract

This article presents three coordinated changes to comet identity and networking on Urbit:

1. Ames may query Jael for a comet's current life and networking keys instead of requiring comet self-attestation;
2. Introduction of a cryptosuite with tweaked signing keys, binding keys to explicit provenance data while preserving Suite B compatibility; and
3. Optional fief-based routing, allowing comets to route directly by static IP or domain and lifting the assumption that sponsor chains must terminate at galaxies.

Collectively, comet identity becomes cryptographically coupled to the chosen PKI substrate while remaining substrate-agnostic at the protocol level.

Adapted from UIP-XXX.[1]

Contents

[1]The author would like to thank ~tinnus-napbus and ~bonbud-macryg for help documenting the original Groundwire codebase, and ~tondes-sitrym and ~sarpen-laplux for developing it.

Manuscript submitted for review.

Address author correspondence to ~hanfel-dovned.

1 Introduction

Urbit currently derives comet identity from self-attested networking keys whose public half hashes into the comet's `@p`. In practice, this creates an implicit coupling between a comet's keys and whichever source of truth a peer happens to trust at first contact.

Groundwire's proposal is deliberately modest and pragmatic. Over the long term, we believe Bitcoin is the only chain that defines canonical history at the protocol level (the heaviest valid proof-of-work chain) in a way that a zero-kelvin Urbit could verify without social coordination or trusted checkpoints. This makes Bitcoin an attractive long-term anchor for the Urbit PKI. At the same time, it is premature to elevate that belief to kernel policy or to bless a single on-chain abstraction (e.g., ordinals/inscriptions) in perpetuity. Different chains—and even different protocols on the same chain—offer distinct data and scripting affordances for identity; we should let those options be explored in the field.

Accordingly, we propose a substrate-agnostic mechanism whereby:

- Ames first consults Jael for comet keys and only falls back to comet self-attestation if Jael lacks a record.
- Networking keys may be *tweaked* using explicit provenance data, so the key material itself cryptographically commits to the on-chain (or off-chain) attestation it corresponds to.

- Comets may advertise routing information (a `$fief`) so packets can route directly without assuming a sponsorship hierarchy that must terminate at galaxies.

This leaves the memorable-name hierarchy (galaxies, stars, planets) on Azimuth untouched while upgrading the vast expanse of ephemeral identities for self-sovereign networking. Practically, Jael continues to subscribe to sources (via `%listen`) chosen by the operator; the kernel does not enshrine a single source of truth.

2 Design Overview

There are three concrete kernel changes entailed by the Groundwire proposal:

1. **Jael-first comet keys.** Ames replaces `+request-attestation` with `+fetch-comet-pki`, which asks Jael for the comet's latest life and corresponding keys. If a record exists, Ames validates messages against it; otherwise, it may fall back to self-attestation. The authoritative store remains Jael's map from `life` to `[crypto-suite=@ud =pass]` in `$point:jael`.

2. **Suite C tweaked keys.** We extend Suite B (`+crub`) to Suite C (`+cric`), which preserves Suite B and adds a case with tweaked signing keys. The untweaked signing public key and tweak data are included so any peer can reconstruct the tweaked key and verify it matches what the comet uses. In practice, current Groundwire comets commit to their ordinal; this prevents a wide class of replay and spoofing attacks and enables bidirectional provenance between on-chain and off-chain attestations.

3. **Fief-based routing.** We add an optional `$fief` field to comet entries in Jael defining a static IP/port or domain. Ames should attempt fief-based routing before sponsor-chain routing. It must not assume sponsor-chain termination at galaxies and must detect cycles.

The remainder of this article recaps elliptic-curve context and the current asymmetric cryptosuite interface used by Ames, then details the tweaked-key construction and routing implications. A prototype exists in Groundwire's forks of `urbit/urbit` and `urbit/vere` implementing `+fetch-comet-pki`, `+cric`, and `$fief`.

3 Elliptic Curves

> Each Urbit ship possesses two networking keypairs: one for encryption, and one for authentication. We often refer to these two keypairs as though they were a single keypair because they are stored as a single atom. Elliptic Curve Diffie-Hellman is used for encryption, while Elliptic Curve Digital Signature Algorithm is used for authentication. (Urbit docs, "Ames Cryptography")

An *elliptic curve* is a two-dimensional curve defined by the general form $y^2 = x^3 + ax + b$ with the special condition that $4a^3 + 27b^2 \neq 0$. As long as that special condition holds true for whatever constants you picked for a and b, you can define a special type of addition for pairs of points on this curve where you draw a line between them, find the singular third point on the curve that this line hits, c, and then flip that third point across the x axis, $-c$ (see Figure 1). Multiplication means doing this process repeatedly.

What is the benefit of this? Its irreversibility. A private key is just a number; a public key is just that number multiplied by some other commonly agreed on number (the "base point") using elliptic curve arithmetic. Regular multiplication would mean that one could trivially obtain someone's private key by dividing their public key by the base point, but by bouncing all our operations around with elliptic curve multiplication, we prevent division. This is less crude and more beautiful than

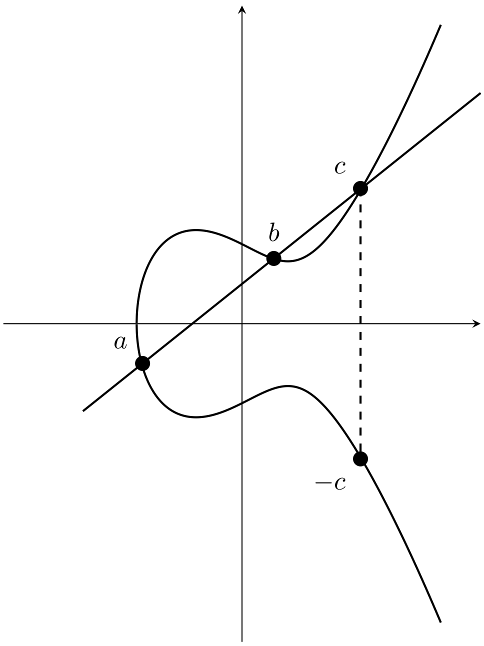

Figure 1: Elliptic curve point addition and multiplication, with the curve $y^2 = x^3 - x + 1$.

it might seem, because these keypairs have some special properties that are useful for cryptography.

If Alice multiplies her private key by Bob's public key, Alice gets the same answer as if Bob multiplied his private key by Alice's public key. This allows the two parties to use that answer as a shared secret that only they know. In the *Elliptic Curve Diffie-Hellman (ECDH)* protocol, we take the x coordinate of that shared secret and run it through a key derivation function (in Urbit's case, SHA-256) to get a symmetric key which allows them to both encrypt and decrypt messages. Using this key, they can communicate privately.

Similarly, *Elliptic Curve Digital Signature Algorithm*

(ECDSA) uses a more extensive algorithm to allow Alice to sign a message with her private key and for Bob to use his public key to verify that Alice did in fact sign it.

The elliptic curve that Urbit uses for key generation is the fast and un-patented *Curve25519*. This specifies the curve equation, the base point, and the finite field (the range of integers that we operate on, wrapping back to 0 if we go over the max value). There are two algebraically equivalent representations of Curve25519 that Urbit uses, with each defining its own set of standard operations: Montgomery Form (X25519), which is more efficient for ECDH, and Twisted Edwards Form (Ed25519), which is more efficient for EDDSA (an elliptic curve signature scheme similar to ECDSA).

Urbit implements the Ed25519 standard operations in its `+ed:crypto` zuse core for EDDSA, along with a few X25519-like operations for ECDH.

- `+scam`: scalar multiply. Takes a point and a scalar → returns scalar × point. This is the core elliptic curve operation. ward: point addition (Edwards curve formula). Needed for scalar multiplication by repeated doubling-and-adding.
- `etch/deco`: encode/decode a curve point into/from an atom. This is how keys are serialized.
- `luck`: derive a keypair (public + private) from a seed. This is how comets/planets get Curve25519 keys.
- `puck`: get the public key from a seed.
- `slar/shar`: perform a Diffie–Hellman secret derivation (ECDH). Given your private scalar and someone else's public point, return a shared secret.
- `sign/sign-raw/sign-octs`: produce signatures. This is the ECDSA/EDDSA-like functionality.
- `veri/veri-octs`: verify signatures against a public key.

With all of this context, you can imagine why we'd want to have two pairs of networking keys: ECDH encryption and EDDSA signatures are both complicated mathematical operations, so it's cleaner and safer to delineate between the two protocols completely in case of implementation bugs, future cryptographic discoveries, nonce reuse, and key rotations.

So how are these keypairs turned into an atom, where are they stored, and when are they used?

4 Asymmetric Cryptosuites

The usage of public and private keys we've described so far is called *asymmetric cryptography*. In current Arvo, the only existing implementation is `+crub:crypto` in zuse:

- `seal`: encrypt to a peer. Runs ECDH with our encode secret + their encode pubkey, hashes with SHA-256, signs with our auth secret, encrypts with AES-SIV.
- `sign`: sign a message with our auth secret (EDDSA).
- `sure`: verify a message was signed with the matching auth pubkey. `tear`: decrypt from a peer. ECDH with our enc secret + their enc pubkey, SHA-256, AES-SIV decrypt, then verify signature.
- `fig`: SHA-256 fingerprint of the public key(s).
- `pac`: SHA-256 fingerprint of the secret key(s).
- `pub`: concatenate encode pubkey + auth pubkey into one atom.
- `sec`: concatenate encode secret + auth secret into one atom.
- `pit`: generate a new keypair from a seed. Uses SHA-512 for privates, `+puck:ed:crypto` for publics.

- `nol`: reconstruct a `+crub` from a secret key atom.
- `com`: reconstruct a `+crub` from a public key atom.

Here we have the key derivation and signing functions as wrappers around `+ed:crypto` and `+aes:crypto`, along with arms for turning networking keys into atoms and atoms back into … `+crubs`? You'd think that we'd turn an atom back into a pair of networking keys. Right?

But what if you wanted to use a different cryptosuite with Ames? Then all of your old keys would be stranded as incompatible atoms. By using a `+crub` object instead, with keys in its payload and a battery for how to use them, we maintain consistency even if there were multiple implementations of `+acru`.

We do, then, need to track which cryptosuite to decode a key atom to—so we store this in the Jael secrets vane right next to the keys themselves:

```
+$  point
  $:  =rift
      =life
      keys=(map life [crypto-suite=@ud =pass])
      sponsor=(unit @p)
  ==
```

Now we can begin to home in on exactly what needs to change in the Urbit crypto suite in order for Groundwire to allow comets to rotate their keys.

In current Arvo, when a comet contacts another ship, that ship will ask the comet for an *unecrypted self-attestation packet* to prove that the sender does in fact control the private key whose public counterpart was hashed into the comet's `@p`:

```
::  +request-attestation: helper to request
::  attestation from comet. Also sets a timer
::  to resend the request every 30s.
::
++  request-attestation
  |=  =ship
  ^+  event-core
  =+  (ev-trace msg.veb ship |.("requesting attestion"))
  =.  event-core
    =/  =blob  (sendkeys-packet ship)
    (send-blob for=| ship blob (~(get by peers.ames-state) ship))
  =/  =wire  /alien/(scot %p ship)
  (emit duct %pass wire %b %wait (add now ~s30))
```

But Groundwire comets don't self-attest to their identity; that attestation comes into Jael via `%ord-watcher` in the same way that Azimuth attestations come into Jael via `%azimuth` and `%eth-watcher`. In the Groundwire codebase, the `+request- attestation` arm is replaced with:

```
++  fetch-comet-pki
  |=  =ship
  ^+  event-core
  =+  (ev-trace msg.veb ship |.("requesting attestion"))
  =/  lyf
    (rof [~ ~] /ames %j `beam`[[our %lyfe %da now] /(scot %p ship
  ?:  ?=([~ ~ [* * ^]] lyf)
    =.  event-core  (emil moves)
    (emit [[//keys]~ %pass /public-keys %j %public-keys ship ~ ~
  =.  event-core
    =/  =blob  (sendkeys-packet ship)
    (send-blob for=| ship blob (~(get by peers.ames-state) ship))
  =/  =wire  /alien/(scot %p ship)
  (emit duct %pass wire %b %wait (add now ~s30))
```

If we have this comet's networking keys in Jael (and thus `lyf` is not null), then instead of asking the comet to self-attest, we check our recorded networking keys. So far, so good: even though we're still expecting the comet's **`@p`** to match its public key in the non-Groundwire `?:` wutcol branch, we can skip that check and trust Jael for Groundwire comets. So why do we need to change our crypto suite?

5 Tweaked Keys

Groundwire's proposed change to comet attestation has greater philosophical implications for identity than you might expect.

Groundwire's strategy is to bet on Bitcoin being the one true long-term substrate that Urbit's PKI will live on, because Bitcoin is the only chain that offers an unambiguous protocol-level definition of canonical history ("the longest valid proof-of-work chain") that a Kelvin 0 Urbit could resolve without appeal to social coordination or trusted checkpoints.

But it *is* a bet, and a contentious one at that in an ecosystem that includes Ethereum and Nockchain. For this reason, we *don't* insist that Jael only ingest comet PKI data from Bitcoin.

All of Groundwire's changes to Arvo stem from modifying Ames's `+request-attestation` arm to `+fetch-comet-pki`, with the core proposition being that comets have the ability to publish their networking keys to any source of PKI truth that other Urbit ships choose to interpret. To enable this, comets need to somehow encode within Jael *which* PKI they've attested to. This is where tweaked keys come in.

The payload of `+crub` is:

```
[pub=[cry=@ sgn=@]\ sek=(unit [cry=@ sgn=@])]
```

which are an encryption public key, a signing public key, and optionally their secret counterparts.

The Groundwire codebase introduces a new crypto-suite, `+cric`, with the payload:

```
$%  $:  suite=%b
        pub=[cry=@ sgn=@ ~]
        sek=$@(~ [sed=@ cry=@ sgn=@])
    ==
    $:  suite=%c
```

```
        pub=[cry=@ sgn=@ tw=[ugn=@ dat=@]]
        sek=$@(~ [sed=@ cry=@ sgn=@])
==  ==
```

We refer to `+crub` as "Suite B" because it implements the NSA Suite B Cryptography Standard. `+cryc` includes everything in Suite B, along with a new "Suite C" (a coinage of our own).

The Suite C payload's main difference is that the signing keys are "tweaked". We store `ugn`, the untweaked public key, and `sed`, the random seed used to originally generate our untweaked keypair, so that we have everything we need to recreate them if we ever need to. Then, we can tweak them by passing them into a tweaking function along with a bytestring, `dat` (omitting from our tweak the additional new field `xtr` which can include extra data):

```
::  pub.s = public signing key
::  sek.s = private signing key
=/  mit
  (shax (can 3 [32 pub.s] [(met 3 dat) dat] ~))
(scad:ed pub.s sek.s mit)
```

First, the public key is hashed along with the data to produce `mit`, the "tweak scalar." All three of these arguments get passed into `+scad:ed`, a wrapper around two other `+ed` arms:

```
::  scalar addition on public and private keys
++  scad
  ~/  %scad
  |=  [pub=@uxpoint sek=@uxscalar sca=@uxscalar]
  ^-  [pub=@uxpoint sek=@uxscalar]
  [(scap pub sca) (scas sek sca)]
```

And these two arms, `+scap` and `+scas`, simply perform elliptic curve addition between the tweak scalar and the corresponding key!

Anyone with both your public untweaked key and your tweak data can recreate your tweaked key. And, sure enough, we recall the `$point:jael` type:

```
+$  point
  $:  =rift
      =life
      keys=(map life [crypto-suite=@ud =pass])
      sponsor=(unit @p)
  ==
```

wherein the `pass` field is the atom that, in Suite B, encodes the two public keys. In Suite C, it encodes the encryption key, the untweaked signing key, *and* the tweak data. Upon receiving a message from a comet, your ship can reconstruct the tweaked key and check if it matches the key that the comet provides.

This might seem like a ton of unnecessary work. If you already trust Jael's sources of truth (`%ord-watcher`, in this case), why bother with tweaked keys at all?

The reason is that tweaking keys allows for a bidirectional cryptographic commitment between on-chain and off-chain attestations to prevent replay, spoofing, and double-spawning attacks. If there wasn't an off-chain anchor pointing to on-chain data, then anyone could come in and attest to ownership of any comet and `%ord-watcher` would naively believe them.

In Groundwire's implementation, that anchor is the ordinal number of the sat used for the comet's inscription. When `%ord-watcher` sees an attempted key rotation, it checks the specified comet identity's `$pass` in Jael to verify that the transaction's ordinal number matches the current tweak data, and rejects it if it doesn't. Back in userspace, any agent can later recompute the tweak to confirm that the key they're communicating with really does originate from the expected on-chain inscription.

This tweaked key protocol enables a new conception of identity on Urbit: rift, life, keys, *and data.*

6 Routing

The final implication of Groundwire's change to Ames' interpretation of comet provenance is that, since Groundwire comets articulate a network independent of Azimuth, they shouldn't need to rely on the current sponsorship hierarchy for packet routing. To facilitate this, comets can include include another piece of info in their attestations, called a `$fief`:

```
+$  fief
  $%  [%turf p=(list turf) q=@udE]
      [%if p=@ifF q=@udE]
      [%is p=@isH q=@udE]
  ==
```

in which `$point:jael` is updated to contain a `(unit\ fief)`, potentially storing the comet's static IP address (`%if` for IPv4 or `%is` for IPv6) or domain (`%turf`).

On top of this, Ames has been updated to allow any ship to be a valid sponsor, necessitating a few changes to its routing logic to detect cycles and ensure that sponsorship chains are allowed to terminate in any sponsor with a `$fief`, rather than only in galaxies.