Hunting for Bugs in "Ethereum 2.0"

JP Aumasson
@veorq

CSO @ taurushq.com

Hunting for Bugs in *Ethereum*

JP Aumasson
@veorq

CSO @ taurushq.com

Hunting for Bugs in *Ethereum clients*

JP Aumasson
@veorq

CSO @ taurushq.com

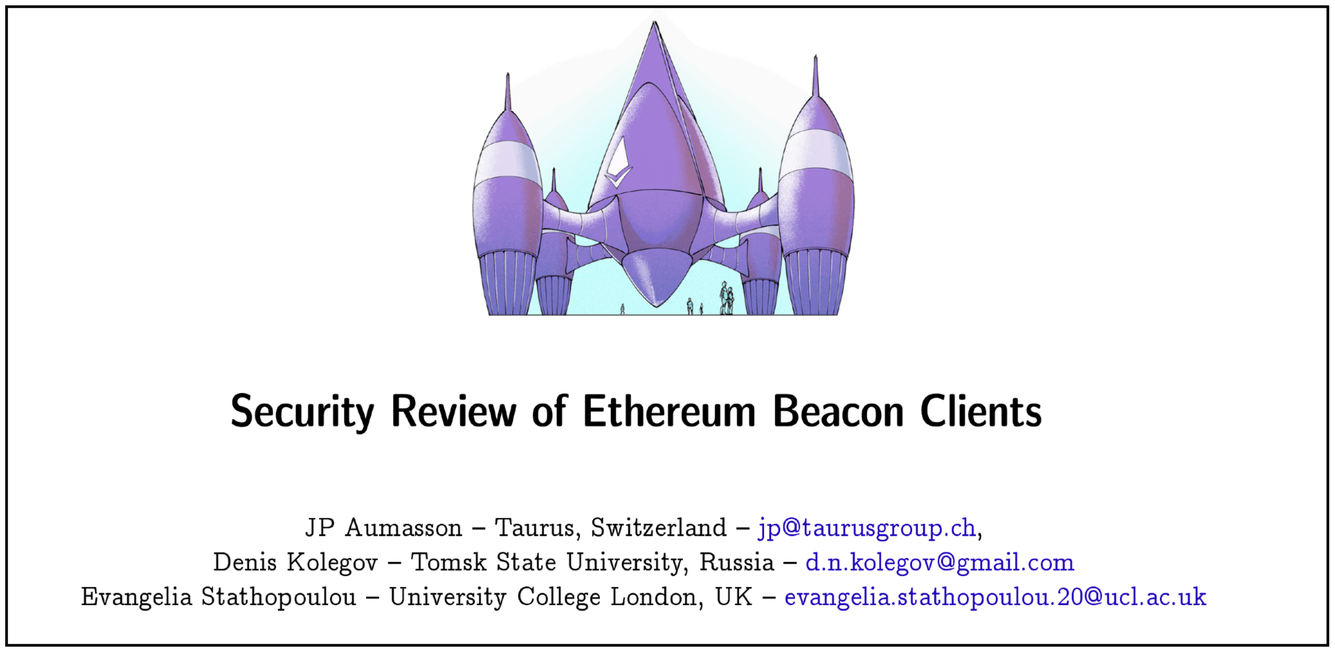
Hunting for Bugs in *Ethereum "clients"*

JP Aumasson
@veorq

CSO @ taurushq.com

Context

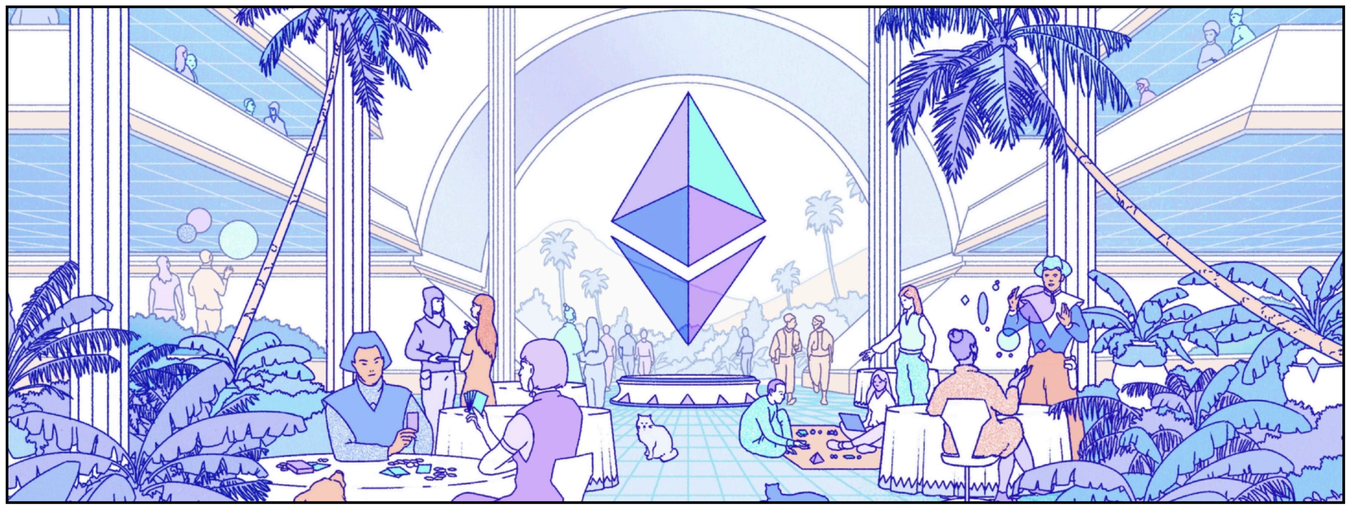
- Collab with **Denis Kolegov** (Protocol Labs) and **Evangelia Stathopoulou** (UCL)
- Research grant from the Ethereum Foundation
- 35 security issues reported, paper at https://arxiv.org/abs/2109.11677



Ethereum

The main public blockchain platform for

- Decentralised applications (“**dApps**”)
- User-defined **tokens**: ERC-20s, NFTs, tokenised securities, etc.
- **DeFi** applications (Uniswap, Compound, etc.)



Ethereum

The main public blockchain platform for

- Decentralised applications ("**dApps**")
- User-defined **tokens**: ERC-20s, NFTs, tokenised securities, etc.
- **DeFi** applications (Uniswap, Compound, etc.)

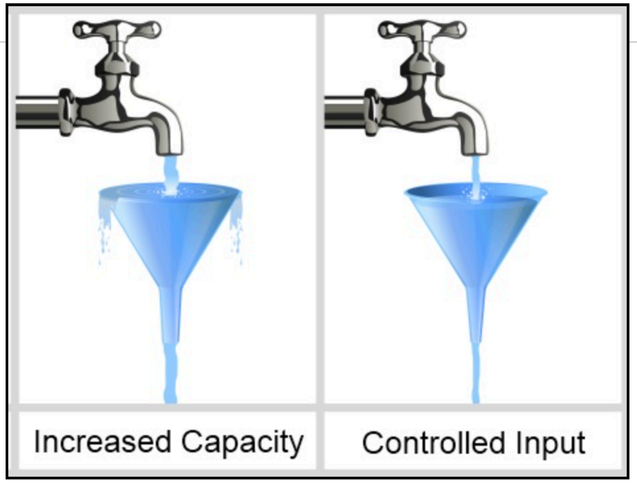
> 25 ⇩ Ethereum network is a mess... high gas fees, slow and congested, a lot of transactions fail and people lose money... this is not the future. Traditional finance is probably laughing at us... we need other blockchains to do better and avoid promising too much and then failing to delivery it.

Reddit user, 2021

Ethereum's scaling problem

To process new transactions, Ethereum nodes need to

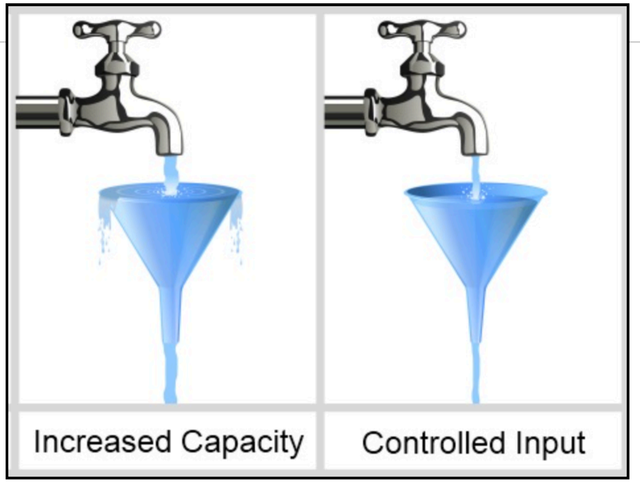
- Run **computation** ("recompute" smart contracts)
- Store **data** (function arguments, state variables)



Ethereum's scaling problem

To process new transactions, Ethereum nodes need to

- Run **computation** ("recompute" smart contracts)
- Store **data** (function arguments, state variables)



Bottleneck of **~15 transactions/second**, leading to

- Increased transaction cost (gas fees)
- Network congestion
- Unhappy users

- Ethereum Blockchain:
  - 600,000 additions per second
  - Cost to use? $250 a second!
- Raspberry Pi 4:
  - 3,000,000,000 additions per second
  - Cost to use? $45 to buy forever!

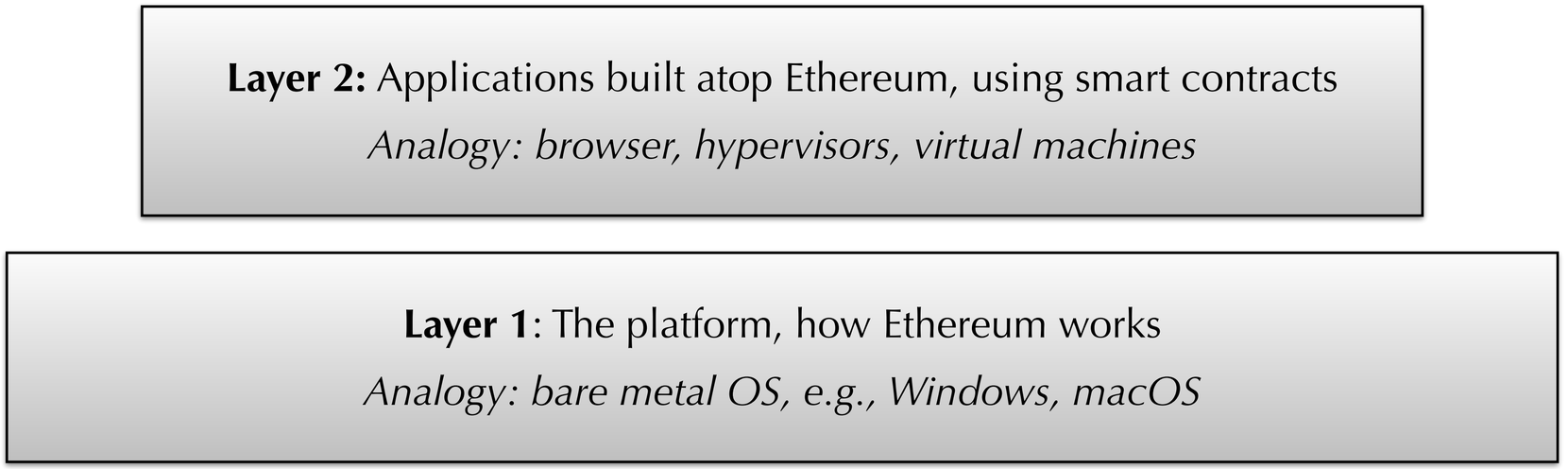
Berkeley EECS

https://cs161.org/assets/lectures/lec12.pdf

Scaling solutions

2 main classes of solutions to address the scalability problem:

- **L1: Change how Ethereum works** (change the “operating system”), by changing the consensus protocol, the way data is stored, how transactions are validated, etc.
- **L2: Create applications** that "define their own rules" to allow for faster transactions



Ethereum "2.0"

The layer-1 approach to scaling, via

- **Proof-of-stake**, instead of proof-of-work
- **Data sharding**, via shard chains
- A coordinator chain called the **Beacon Chain** (shipped on Dec 2020)

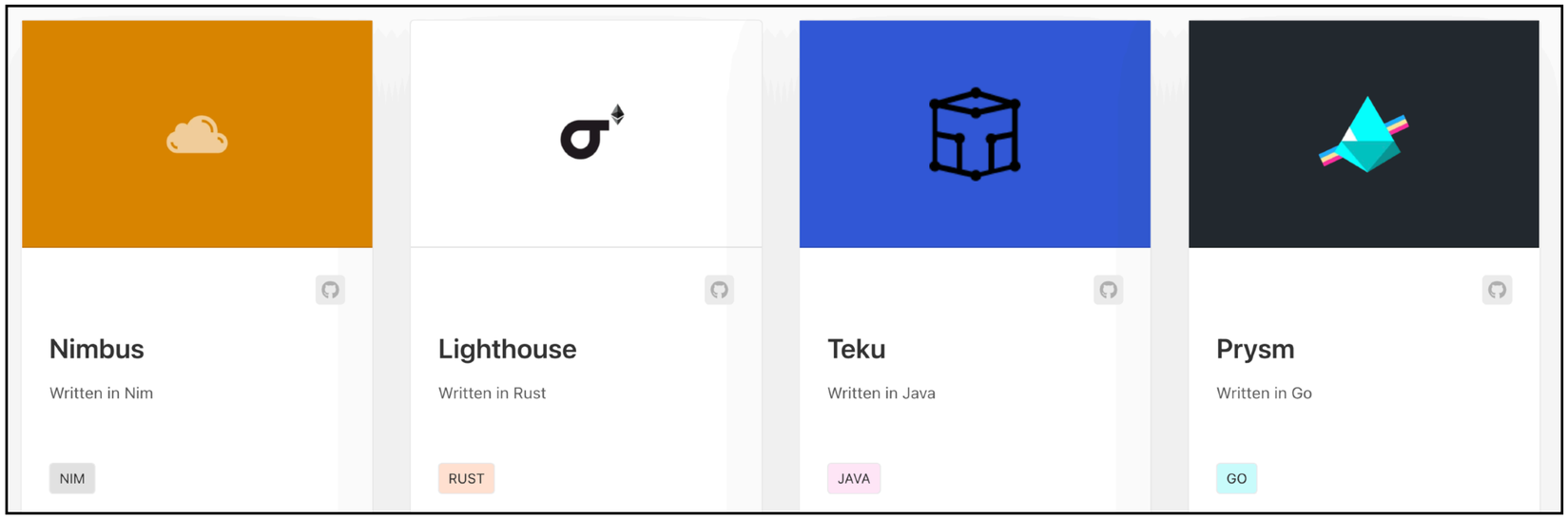
Tl;dr;

- The terms Eth1 and Eth2 (Ethereum 2.0) are being phased out
- Execution layer (Eth1) and consensus layer (Eth2) are the new terminologies
- The roadmap to scale Ethereum in a decentralized way remains the same
- You don't need to do anything

https://blog.ethereum.org/2022/01/24/the-great-eth2-renaming/

The Beacon Chain

- Network of nodes that interact to maintain a state as per a consensus protocol
- Nodes' **server software** are "**beacon clients**", or "consensus clients"



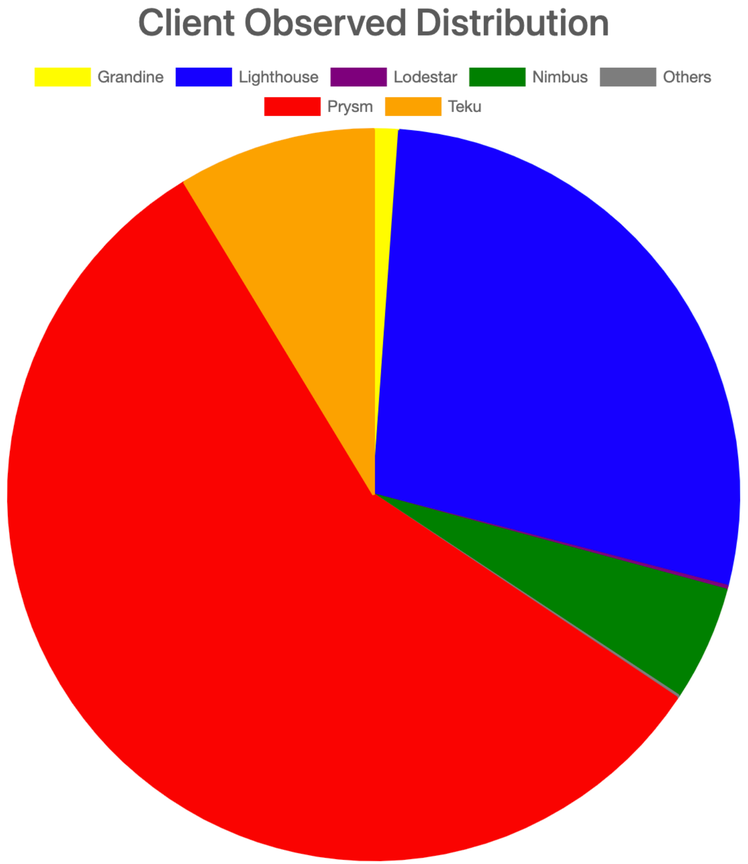
https://ethereum.org/en/upgrades/get-involved/#clients

The Beacon clients

Client & Repository	Language	Developers	Repo Stars	Open/Closed Issues
Lighthouse sigp/lighthouse	Rust	Sigma Prime sigmaprime.io	1.3k	100/846
Nimbus status-im/nimbus-eth2	Nim	Status status.im	222	152/526
Prysm prysmaticlabs/prysm/	Go	Prysmatic Labs prysmaticlabs.com	2.2k	114/2016
Teku ConsenSys/teku	Java	ConsenSys consensys.net	281	82/1251

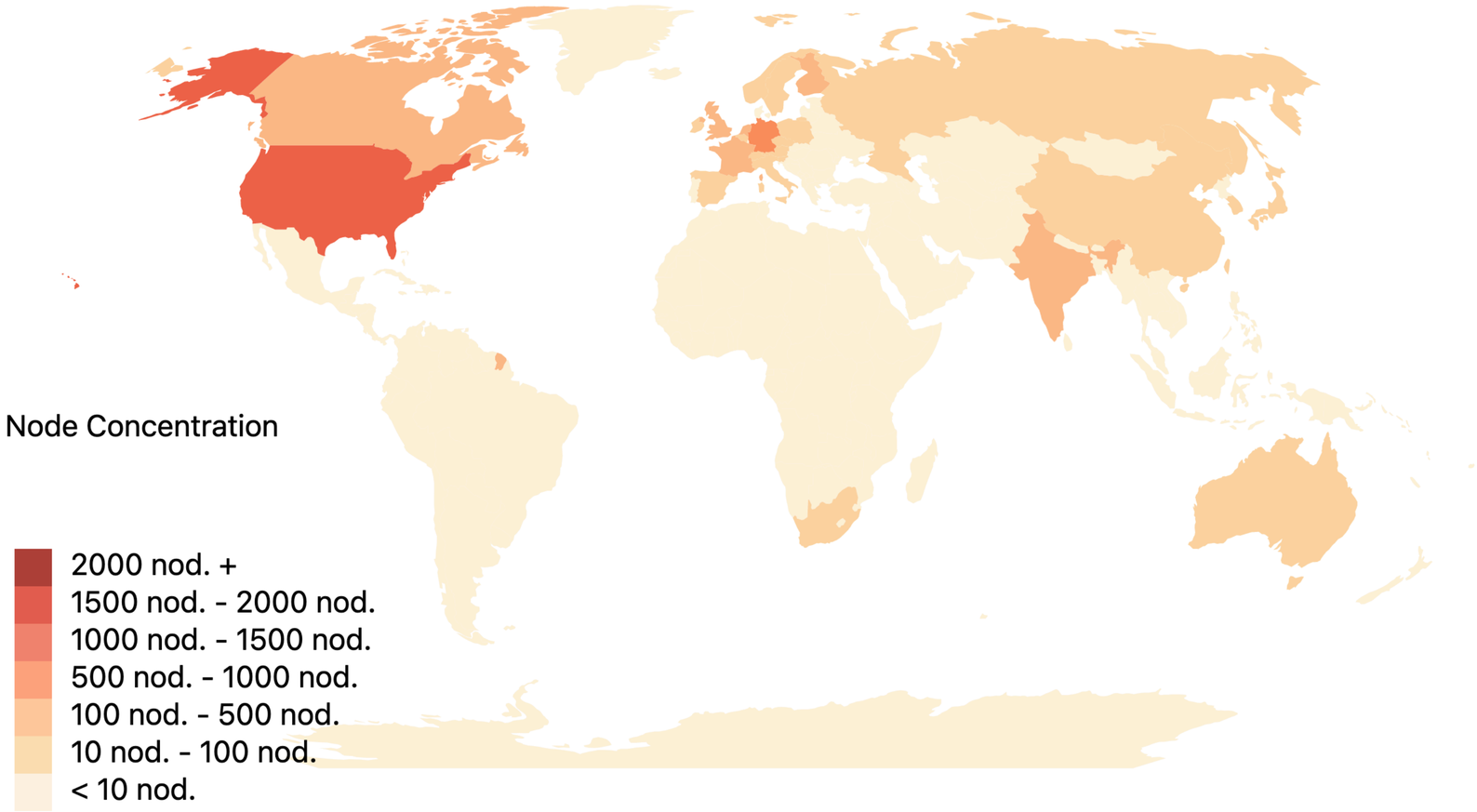
Table 2: Overview of the beacon clients reviewed, as of 20210913.

The Beacon clients



https://migalabs.es/crawler/dashboard

The Beacon clients



https://migalabs.es/crawler/dashboard

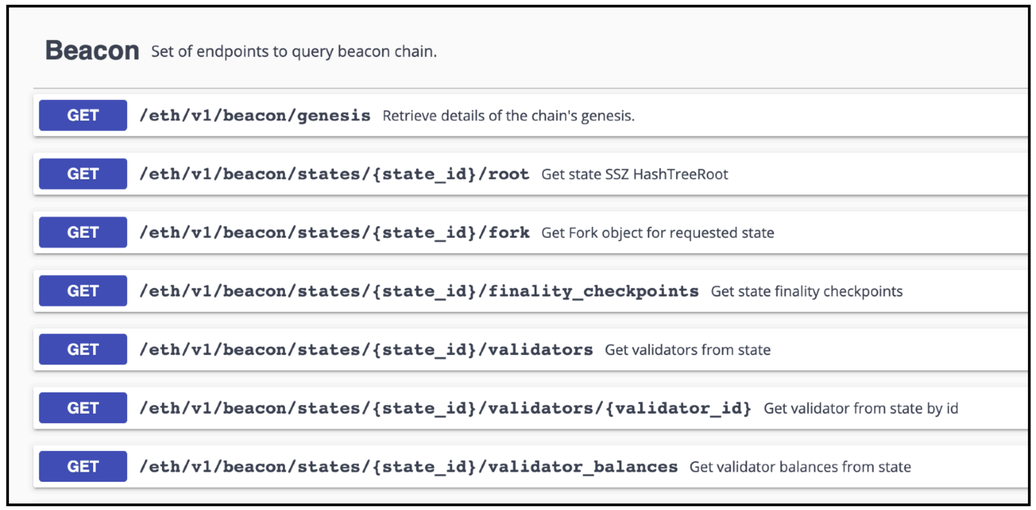
The Beacon clients

Run 2 services (+ an optional slasher service)

- A **beacon node**, a “passive" service that maintains a view of the chain
- A **validator**, the “active" service, proposing and signing state modifications, requires to stake 32 ETH to run a validator

Newest components compared to "Eth1":

- Cryptographic **signatures** (BLS)
- **Slashing**, the punishing mechanism
- The Beacon **API**



Methodology

- **Compared specifications** with the implementations (can find bugs in either)
- **Compared implementations** of a same functionality across 4 clients
  - A bug in one client may occur in others as well
  - “Why do they do this differently?” helps discover bugs
  - Review reuse of same core libs with different bindings
- Mostly **code review** + local code execution

BLS signatures

- Can **aggregate** signature/pubkeys, and allow efficient **batch** verification
- At the same time **much simpler** and **more complex** than ECDSA or Schnorr

```
Signature = SecretKey × Hash(Message)
```

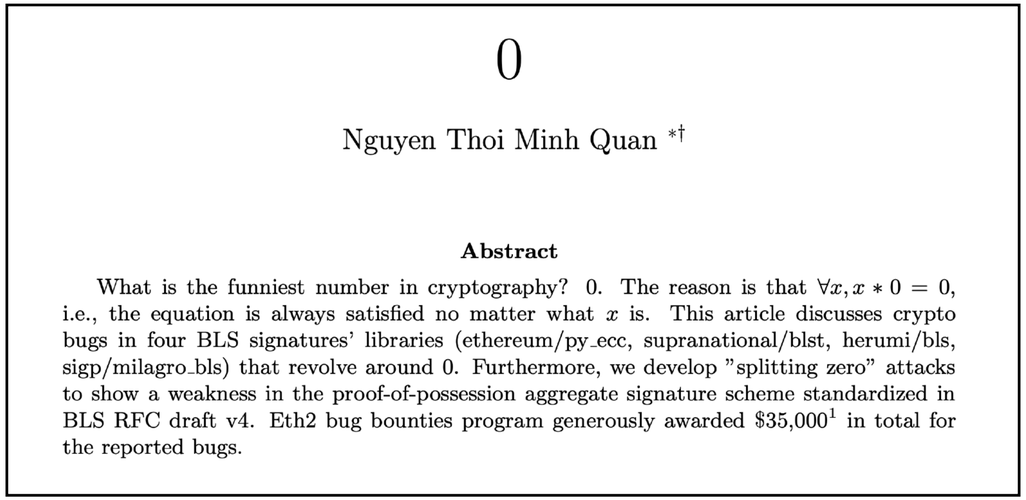
- What can go wrong?

BLS signatures

- Can **aggregate** signature/pubkeys, and allow efficient **batch** verification
- At the same time **much simpler** and **more complex** than ECDSA or Schnorr

```
Signature = SecretKey × Hash(Message)
```

- What can go wrong?



0

Nguyen Thoi Minh Quan *†

Abstract

What is the funniest number in cryptography? 0. The reason is that $\forall x, x * 0 = 0$, i.e., the equation is always satisfied no matter what x is. This article discusses crypto bugs in four BLS signatures' libraries (ethereum/py_ecc, supranational/blst, herumi/bls, sigp/milagro_bls) that revolve around 0. Furthermore, we develop "splitting zero" attacks to show a weakness in the proof-of-possession aggregate signature scheme standardized in BLS RFC draft v4. Eth2 bug bounties program generously awarded $35,000[1] in total for the reported bugs.

https://eprint.iacr.org/2021/323.pdf

BLS signatures

- Specified in an IETF draft, as multiple procedures for signing, verifying, etc.
- Example: `CoreVerify`

```
1. R = signature_to_point(signature)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
6. Q = hash_to_point(message)
7. C1 = pairing(Q, xP)
8. C2 = pairing(R, P)
9. If C1 == C2, return VALID, else return INVALID
```

BLS signatures

- Specified in an IETF draft, as multiple procedures for signing, verifying, etc.
- Example: `CoreVerify`

```
1. R = signature_to_point(signature)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
6. Q = hash_to_point(message)
7. C1 = pairing(Q, xP)
8. C2 = pairing(R, P)
9. If C1 == C2, return VALID, else return INVALID
```

BLS signatures

- Specified in an IETF draft, as multiple procedures for signing, verifying, etc.
- Example: `CoreVerify`

```
1. R = signature_to_point(signature)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
6. Q = hash_to_point(message)
7. C1 = pairing(Q, xP)
8. C2 = pairing(R, P)
9. If C1 == C2, return VALID, else return INVALID
```

BLS signatures bugs

- Reported **19 issues** related to BLS signatures, across all projects
- Only **low/mid** severity, no “get rich for free” exploitation scenarios :)

supranational/blst
Enforce limitation on IKM length

sigp/milagro_bls
Check that IKM is more than 32B in KeyGen

ChainSafe/bls
BLS secret key validation is missing

ChainSafe/blst-ts
Incomplete key validation Incorrect result for zero lengths arrays in aggregateVerify Detect unsafe coefficients in verifyMultipleAggregateSignatures

sigp/lighthouse
Missing check on seed and password length

Specifications
Bit security level < 128 BLS parameters section number fix

prysmaticlabs/prysm
Missing input validation in SecretKeyFromBigNum Detect unsafe coefficients in VerifyMultipleSignatures No length check in AggregatePublicKeys

ConsenSys/teku
Public key aggregation ambiguous infinite points handling Detect unsafe coefficients in fast BLS verification Incorrect BLS key validation Detect unsafe coefficients in fast BLS verification

Peer-to-peer (P2P) communication

ethereum / **consensus-specs** Public

Why are we using encryption at all?

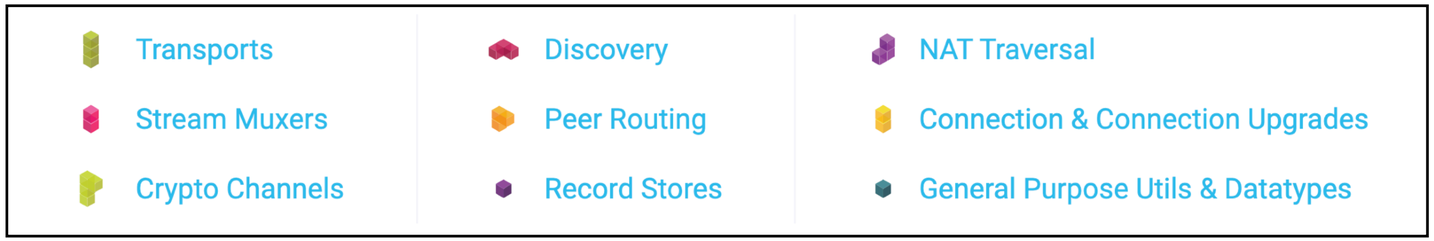
Transport level encryption secures message exchange and provides properties that are useful for privacy, safety, and censorship resistance. These properties are derived from the following security guarantees that apply to the entire communication between two peers:

- Peer authentication: the peer I'm talking to is really who they claim to be and who I expect them to be.
- Confidentiality: no observer can eavesdrop on the content of our messages.
- Integrity: the data has not been tampered with by a third-party while in transit.
- Non-repudiation: the originating peer cannot dispute that they sent the message.
- Depending on the chosen algorithms and mechanisms (e.g. continuous HMAC), we may obtain additional guarantees, such as non-replayability (this byte could've only been sent *now;* e.g. by using continuous HMACs), or perfect forward secrecy (in the case that a peer key is compromised, the content of a past conversation will not be compromised).

https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/p2p-interface.md

P2P

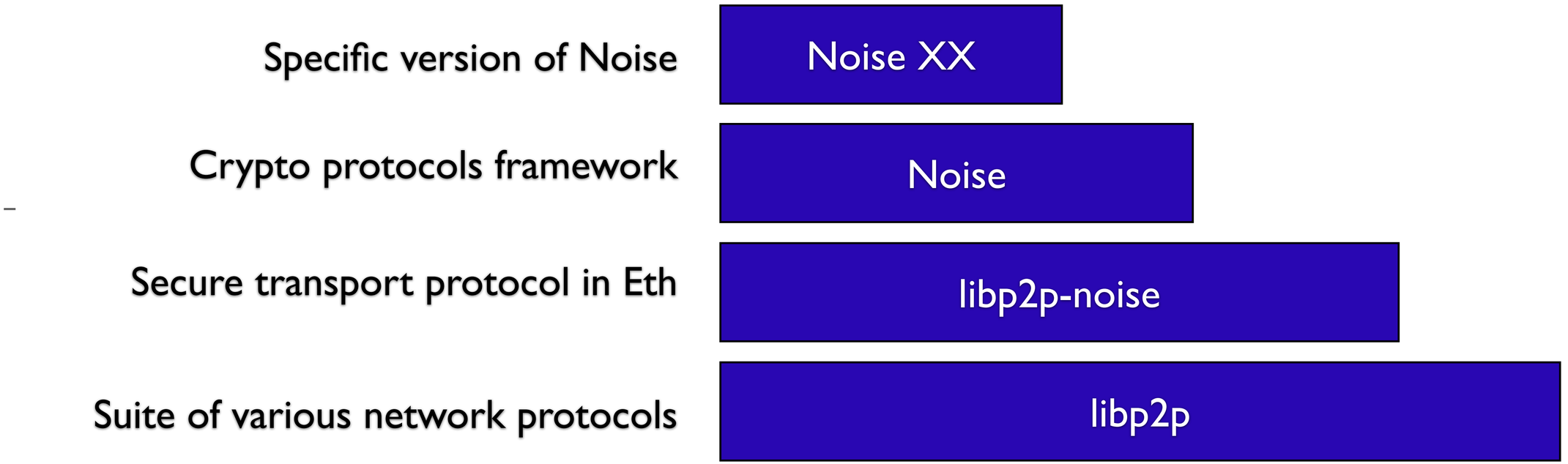
- Ethereum nodes' **secure transport** is based on the **libp2p-noise** protocol
- Libp2p-noise is part of the **libp2p** suite, "the de facto web3 networking layer"



https://libp2p.io/implementations/

P2P

- Ethereum nodes' **secure transport** is based on the **libp2p-noise** protocol
- Libp2p-noise is part of the **libp2p** suite, "the de facto web3 networking layer"



The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

Static key pair (SK-**S**, PK-**S**)

JESSICA

New **ephemeral** key pair
(SK-**E**, PK-**E**)

PK-**E**

PK-**E**, **Enc**(PK-**S**)

DH1=**DH**(PK-**E**, SK-**E**)
Decrypt PK-**S**
DH2=**DH**(SK-**E**, PK-**S**)

DH3=**DH**(SK-**S**, PK-**E**)

New ephemeral key pair
Pick (SK-**E**, PK-**E**)

DH1=**DH**(PK-**E**, SK-**E**)
DH2=**DH**(PK-**E**, SK-**S**)

Enc(PK-**S**)

Decrypt PK-**S**
DH3=**DH**(PK-**S**, SK-**E**)

Static key pair (SK-**S**, PK-**S**)

MORTY

Payloads encryption key combes all DH's, see https://noiseexplorer.com/patterns/XX/

The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

- What can go wrong?

The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

- What can go wrong? **Replay**!
- How to fix it?

The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

- What can go wrong? **Replay**!
- How to fix it? Sign `key||X`, where `X` is **unpredictable** (random, session hash, etc.)

The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

- What can go wrong? **Replay**!
- How to fix it? Sign `key||X`, where `X` is **unpredictable** (random, session hash, etc.)
- How could this be abused?

Pick (SK-**E**, PK-**E**)
DH1=**DH**(PK-**E**, SK-**E**)
DH2=**DH**(PK-**E**, SK-**S**)

Decrypt PK-**S**
DH3=**DH**(PK-**S**, SK-**E**)

The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

- What can go wrong? **Replay**!
- How to fix it? Sign `key||X`, where `X` is **unpredictable** (random, session hash, etc.)
- How could this be abused? **DoS**! (if UDP)
- How to fix it?

Pick (SK-**E**, PK-**E**)
DH1=**DH**(PK-**E**, SK-**E**)
DH2=**DH**(PK-**E**, SK-**S**)

Decrypt PK-**S**
DH3=**DH**(PK-**S**, SK-**E**)

The Noise XX key agreement

In Ethereum, each party also has an **identity key**, used to sign new static keys (PK-**S**)

- What can go wrong? **Replay**!
- How to fix it? Sign `key||X`, where `X` is **unpredictable** (random, session hash, etc.)
- How could this be abused? **DoS**! (if UDP)
- How to fix it? **Cookies**!

Pick (SK-**E**, PK-**E**)
DH1=**DH**(PK-**E**, SK-**E**)
DH2=**DH**(PK-**E**, SK-**S**)

Decrypt PK-**S**
DH3=**DH**(PK-**S**, SK-**E**)

> Furthermore, computing the `DH()` function is CPU intensive. In order to fend off a CPU-exhaustion attack, if the server is under load, it may choose to not process handshake messages, but instead respond with a cookie reply packet. In order for the server to remain silent unless it receives a valid packet, while under load, all messages are required to have a MAC that combines the receiver's public key and optionally the PSK as the MAC key. When the server is under load, it will only accept packets that additionally have a second MAC of the prior bytes of the message that utilize the cookie as the MAC key. We therefore compute `msg.mac1` and `msg.mac2` as seen in the handshake messages above.

https://www.wireguard.com/protocol/

Libp2p-noise int overflow

- In https://github.com/ChainSafe/js-libp2p-noise (used in the Lodestar client)
- Traced back to Noise Explorer's Go code generation
- Famous **nonce reuse** problem of stream ciphers

> A `CipherState` responds to the following functions. The ++ post-increment operator applied to `n` means "use the current `n` value, then increment it". The maximum `n` value (2^{64}-1) is reserved for other use. If incrementing `n` results in 2^{64}-1, then any further `EncryptWithAd()` or `DecryptWithAd()` calls will signal an error to the caller.

> - **Incrementing nonces**: Reusing a nonce value for `n` with the same key `k` for encryption would be catastrophic. Implementations must carefully follow the rules for nonces. Nonces are not allowed to wrap back to zero due to integer overflow, and the maximum nonce value is reserved. This means parties are not allowed to send more than 2^{64}-1 transport messages.

https://noiseprotocol.org/noise.html

Libp2p-noise int overflow

- In https://github.com/ChainSafe/js-libp2p-noise (used in the Lodestar client)
- Traced back to Noise Explorer's Go code generation
- Famous **nonce reuse** problem of stream ciphers

```
226       - func encryptWithAd(cs *cipherstate, ad []byte, plaintext []byte) (*cipherstate, []byte) {
      227 + func encryptWithAd(cs *cipherstate, ad []byte, plaintext []byte) (*cipherstate, []byte, error) {
      228 +         var err error
      229 +         if cs.n == math.MaxUint64-1 {
      230 +                 err = errors.New("encryptWithAd: maximum nonce size reached")
      231 +                 return cs, []byte{}, err
      232 +         }
227   233           e := encrypt(cs.k, cs.n, ad, plaintext)
228   234           cs = setNonce(cs, incrementNonce(cs.n))
229       -         return cs, e
      235 +         return cs, e, err
```

https://github.com/symbolicsoft/noiseexplorer

Libp2p-noise MitM

- A few days ago (not our bug)

CVE-2022-24759 Detail

Current Description

`@chainsafe/libp2p-noise` contains TypeScript implementation of noise protocol, an encryption protocol used in libp2p. `@chainsafe/libp2p-noise` before 4.1.2 and 5.0.3 does not correctly validate signatures during the handshake process. This may allow a man-in-the-middle to pose as other peers and get those peers banned. Users should upgrade to version 4.1.2 or 5.0.3 to receive a patch. There are currently no known workarounds.

Arian van Putten 🐦🇳🇱❤️🇺🇦
@ProgrammerDude

The fact that typescript doesn't complain about !validate() vs !await validate() lead to complete signature bypass.

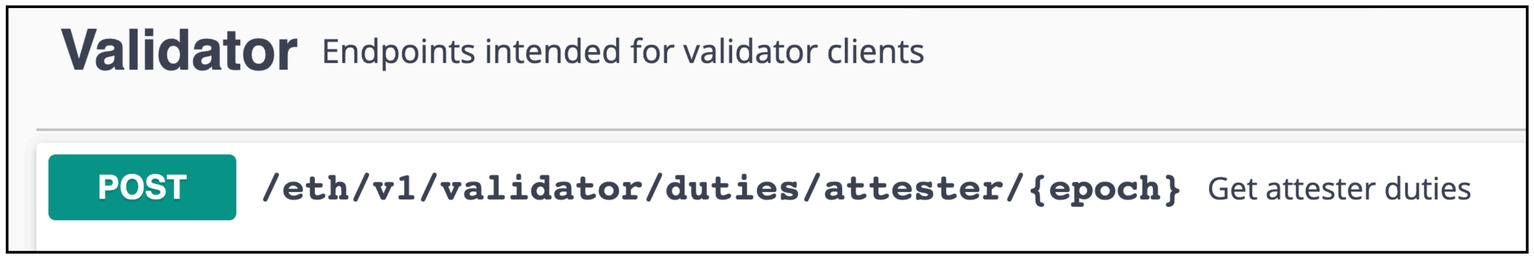
! On a Promise always returns true and is not a type error....

```
-   if (!payload.identitySig || !peerId.pubKey.verify(generatedPayload, payload.
+   if (!payload.identitySig || !(await peerId.pubKey.verify(generatedPayload, p
      throw new Error("Static key doesn't match to peer that signed payload!")
```

Beacon API

Found some of the **“usual" security issues** across clients, such as

- Incorrect handling of headers (e.g. `Content-Type`, `Accept`)
- Lack of JSON schema validation
- Public exposure of the API (without authentication)
- Authentication tokens written in logs
- POST and PATCH requests possible without API token
- DoS vectors



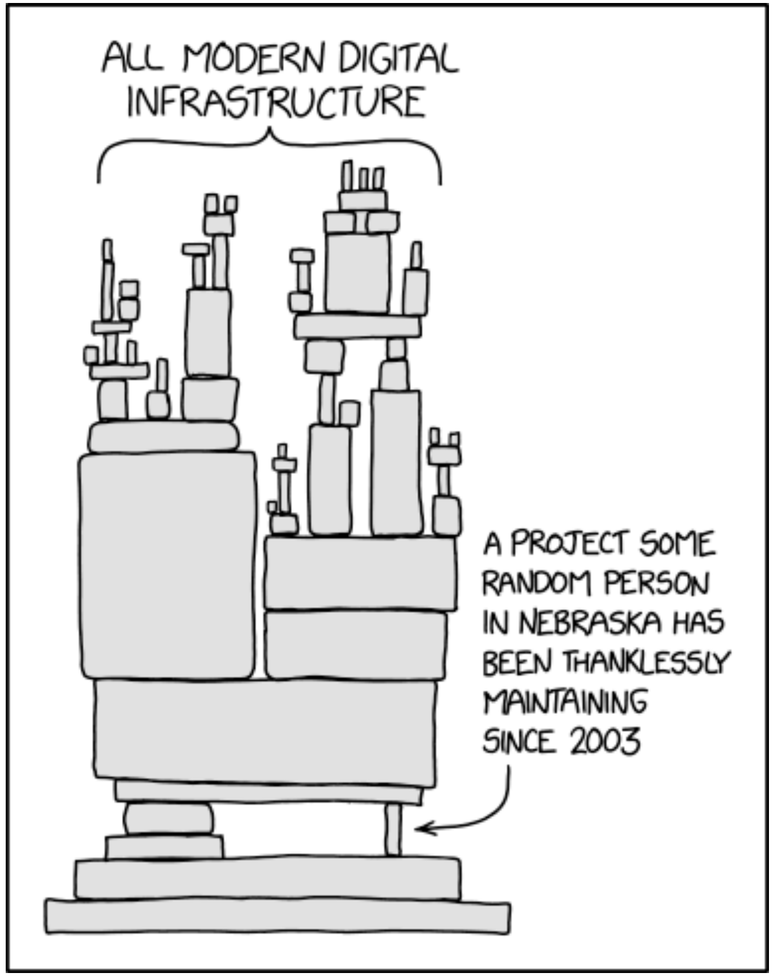
Supply-chain risks

Most modern software is **other people's code**

- Risk of sabotage (backdoors, bugdoors)
- Version management nightmare
- Copyright and licensing issues

Tooling is being developed for

- Inventorying dependencies (dependency graph)
- Finding outdate or vulnerable versions



Supply-chain risks

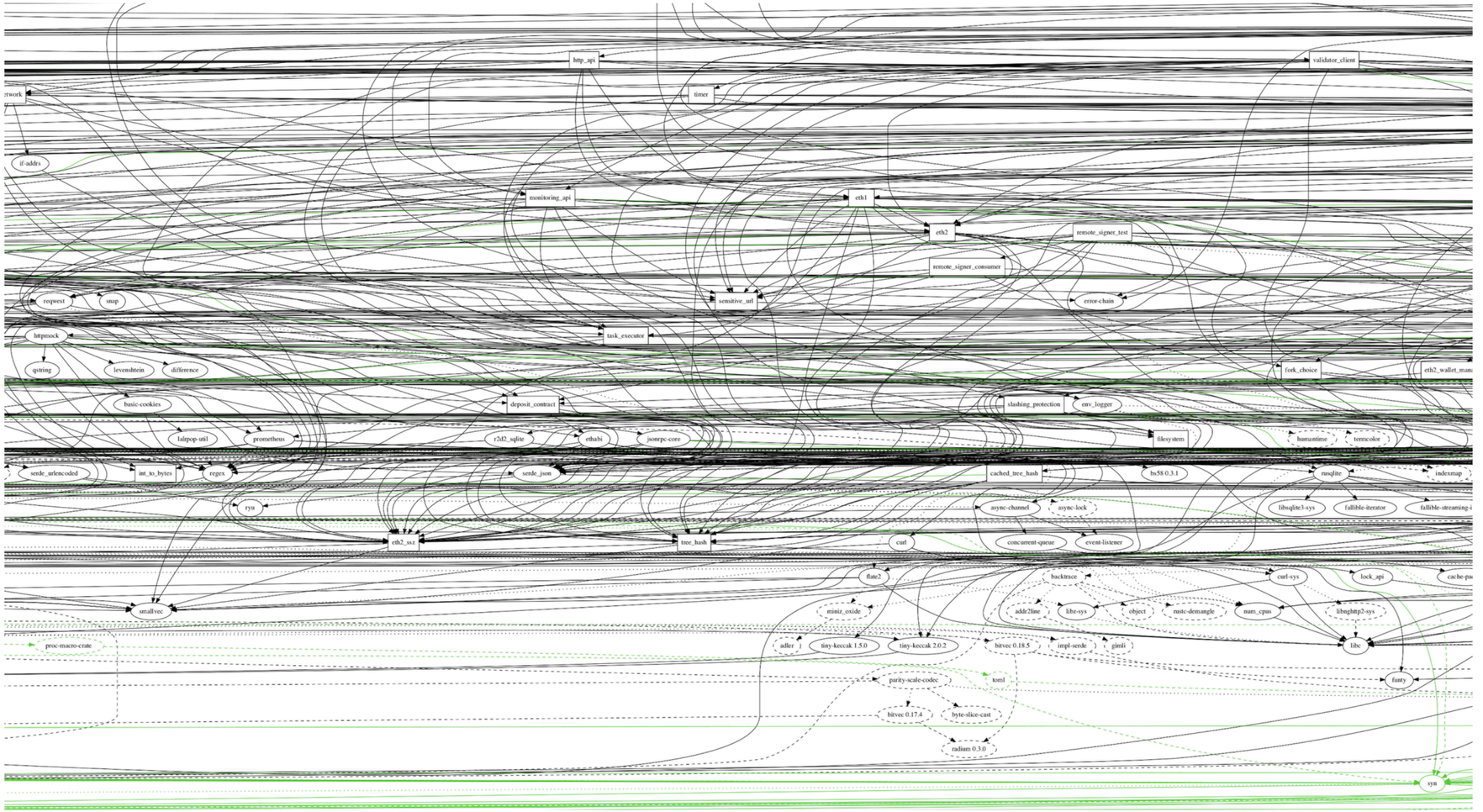


Figure 1: Lighthouse dependencies graph (excerpt).

Supply-chain risks

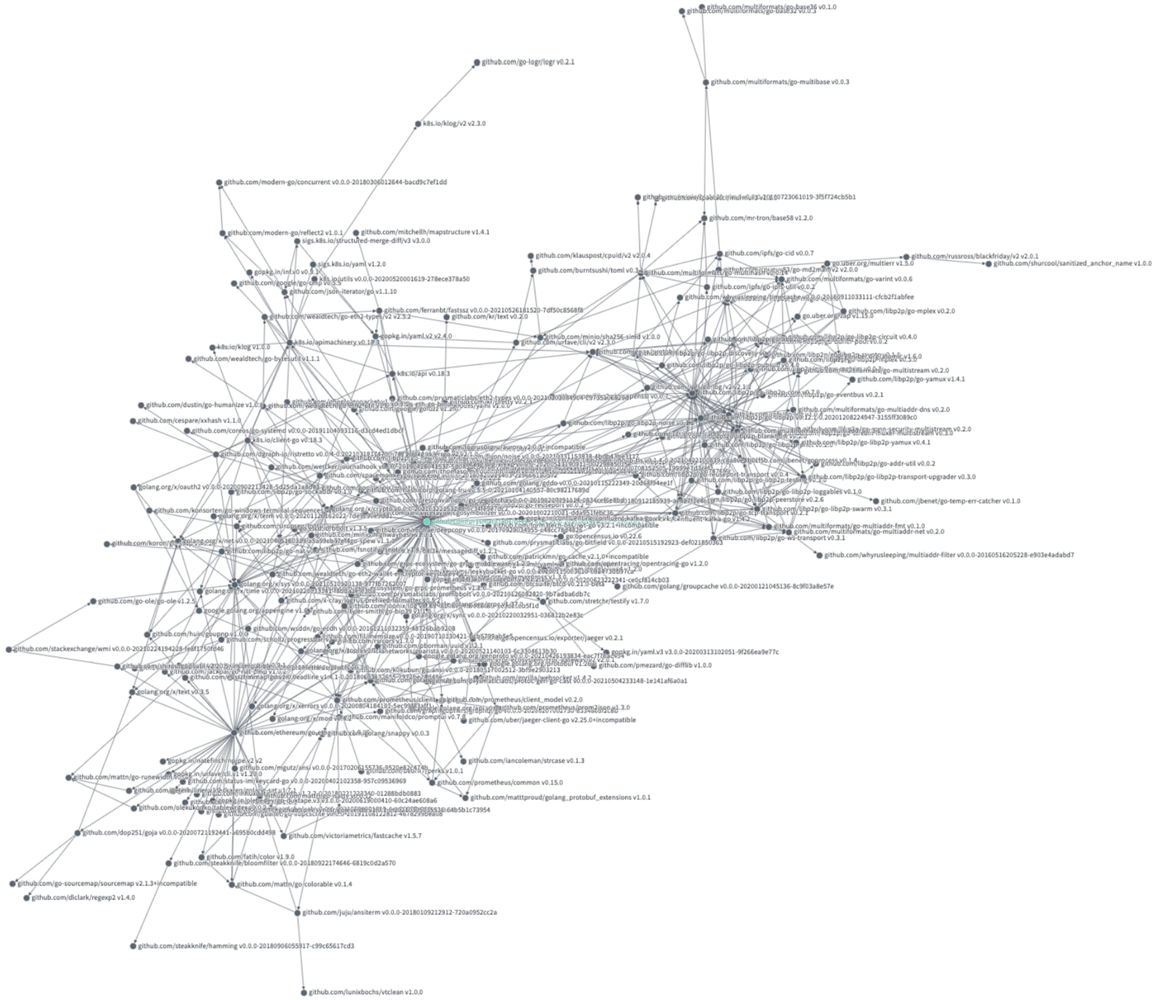


Figure 2: Prysm dependencies graph, generated using https://deps.dev/.

Supply-chain risks

Our goal: **find risk indicators** that are

- Easy to calculate
- Meaningful and fair
- Language-agnostic

Metrics about

- **Dependencies** (quantity, quality)
- SDLC & maintainance

Metric	Lighthouse	Nimbus	Prysm	Teku
Language	Rust	Nim	Go	Java
GitHub Stars	1.2k	212	2.2k	257
Direct dependencies	121	43	93	48
Total dependencies	440	56	665	230
Max degree of dependencies	15	3	13	13
Outdated versions	59	0	353	N/A
Vulnerable versions	5	0	5	18
CVEs	6	0	11	23
Last commit	10/06/21	05/08/21	10/08/21	06/08/21
Last release	10/06/21	05/08/21	03/08/21	28/07/21
Open issues	100	150	97	81
Closed issues	816	514	1999	1215

Table 3: Overview of the risk metrics, as of 20210810.

Conclusions

- **No high/critical sev** bug found
  - Already good level of testing, fuzzing, security audits
  - But complex systems + lot of code = hard to catch bugs
  - High incentives for attackers to invest in finding and stockpiling bugs

Conclusions

- **No high/critical sev** bug found
  - Already good level of testing, fuzzing, security audits
  - But complex systems + lot of code = hard to catch bugs
  - High incentives for attackers to invest in finding and stockpiling bugs
- *"What's the **best client**? Which one should I use?"* **It depends™**
  - Lighthouse is the most security-focused, Prysm is the most popular
  - Nimbus is lighter, Teku is enterprise-oriented
  - A reasonable level of client diversity seems preferable, security-wise

Thank you!

JP Aumasson
@veorq

CSO @ taurushq.com