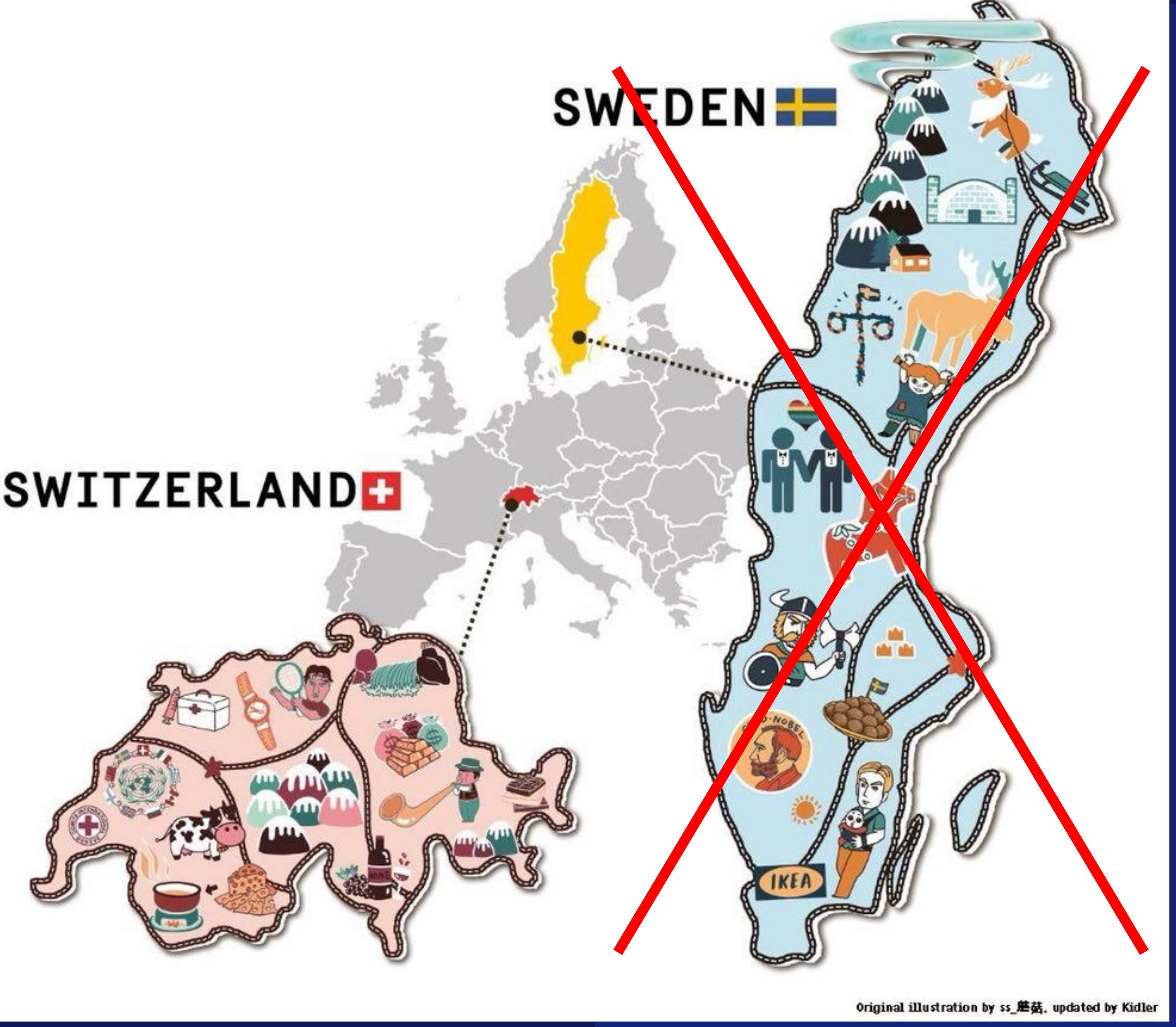
Randomness 101

or “how not to mess up your secret keys next time”

Yolan Romailler (@anomalroil)

Protocol Labs

Left blank on purpose for captioning system.



Who am I?

→ SWE @ Protocol Labs

→ CTF player (mostly crypto, forensic & misc)

→ Maths background, but don't worry!

→ Board games amateur

→ Love to go to confs: DEF CON, NSec, MCH, GopherConEU, etc.

Protocol Labs

Email
yolan@protocol.ai

Twitter
@anomalroil

Agenda

- What is randomness and its flavours?
- Why do we need it?
- Why are there problems with it?
- In practice, how to avoid problems

Protocol Labs

Intro:

What is randomness?

Chapter I

Chapter II

Chapter III

Chapter IV

Protocol Labs

What is randomness?

According to the Cambridge dictionary, randomness is:

Protocol Labs

What is randomness?

According to the Cambridge dictionary, randomness is:

- “the quality of being random”

Protocol Labs

What is randomness?

According to the Cambridge dictionary, randomness is:

- “the quality of being random”

Granted, they refine it a bit:

- “the quality of being random (= happening, done, or chosen by chance rather than according to a plan)”

Protocol Labs

What is randomness?

On my side, I prefer the Oxford Languages definition:

- “the quality or state of **lacking a pattern** or principle of organization; **unpredictability**”

Protocol Labs

Is that random?

```
0b1111111111111111111111111111111111111
0b0100111101011110110011010110000000101
0b1000000000000000000011111111111111111
0b1001000110100010101100111100010101011
0b1010101010101010101010101010101010101
```

Is that random?

~~0b1111111111111111111111111111111111111~~

0b0100111101011110110011010110000000101

0b1000000000000000000111111111111111111

0b1001000110100010101100111100010101011

0b1010101010101010101010101010101010101

Is that random?

~~0b1111111111111111111111111111111111111~~

0b0100111101011110110011010110000000101 → 0x09ebd9ac05

0b1000000000000000000111111111111111111

0b1001000110100010101100111100010101011

0b1010101010101010101010101010101010101

Is that random?

~~0b1111111111111111111111111111111111111~~

0b0100111101011110110011010110000000101 → 0x09ebd9ac05

~~0b1000000000000000000111111111111111111~~

0b1001000110100010101100111100010101011

0b1010101010101010101010101010101010101

Is that random?

~~0b1111111111111111111111111111111111111~~

0b0100111101011110110011010110000000101 → 0x09ebd9ac05

~~0b1000000000000000000001111111111111111~~

~~0b1001000110100010101100111100010101011~~ → 0x12345678ab

0b1010101010101010101010101010101010101

Is that random?

~~0b1111111111111111111111111111111111111~~

0b0100111101011110110011010110000000101 → 0x09ebd9ac05

~~0b1000000000000000000111111111111111111~~

~~0b1001000110100010101100111100010101011~~ → 0x12345678ab

~~0b1010101010101010101010101010101010101~~

Is that random?

And yet all of them have the same probability of occurring in a random draw!

~~0b1111111111111111111111111111111111111~~

0b0100111101011110110011010110000000101 → 0x09ebd9ac05

~~0b1000000000000000000111111111111111111~~

~~0b1001000110100010101100111100010101011~~ → 0x12345678ab

~~0b1010101010101010101010101010101010101~~

The notion of randomness

- So, we have some kind of intuition of “**what is random**”, but it still can be fooled.
- A more formal treatment of randomness can be done using “the Kolmogorov complexity” which can also help us understand our intuition.
- The **Kolmogorov complexity** of something is the length of a shortest program (in a given language) that produces that thing as output:

```
print(‘0b’+37*’1’)                                        → 19 chars
print(‘0b1111111111111111111111111111111111111’) → 48 chars
```

Protocol Labs

What is randomness?

Back to the start!

I like to say there are different flavours of randomness:

- Secret Randomness
- Public Randomness
- A few extra specific ones

What is *secret* randomness?

We often rely on secret randomness:

- to generate keys, both for public key and symmetric cryptography
- for ephemeral keys / IV / nonces

You might not realise, but you're using such randomness daily.

What is *public* randomness?

- Public randomness is simply a **random value that is meant to be *public***

- We want public randomness typically for:
  - reproducibility
  - auditability

Protocol Labs

Public != Secret

WARNING:

DO NOT USE PUBLIC RANDOMNESS TO GENERATE CRYPTOGRAPHIC MATERIAL

What is *verifiable* randomness?

- Public randomness is cool, but we usually use it when we need “public auditability”, we need to be “off the hook”.
- **Verifiable** randomness is randomness that can be verified to have been **properly issued and not manipulated**
- Its goal is usually to **increase the trust** we have in a random “draw” (think of lotteries, tombola, jury selection, etc.)

Protocol Labs

What is *distributed* randomness?

The notion of **distributed randomness** hides many problems:

- decentralisation of trust: no given trusted third party
- achieving consensus on a random value is hard
- high-availability: no single point of failure

Failing at producing proper randomness can be very dangerous for any distributed system, especially nowadays for blockchains.

Protocol Labs

Why:

Why do we need randomness?

Chapter I

Chapter II

Chapter III

Chapter IV

Protocol Labs

Why do we need randomness?

Protocols & Cryptography:

- **Protocols**: leader election in Proof of Stake blockchains, Tor (path selection), sharding
- **Gossiping**: randomly choosing peers in the network to disseminate information
- **Parameters**: Nonces & IV for symmetric encryptions, prime numbers generation, ECC parameters
- **Schemes**: Diffie Hellman exchange, Schnorr signatures, more generally for zero knowledge proofs, One-Time Pad

Why do we need randomness?

But also much more:

- **Lotteries, jury selection, sortitions, random audits…**
- **Statistics:** sampling, reducing bias in controlled trials in medicine, …
- **Software:** fuzzing, chaos monkey, …
- Even useful for cleromancy and divination!

Protocol Labs

But, why?

The New York Times

Comey and McCabe, Who Infuriated Trump, Both Faced Intensive I.R.S. Audits

The former F.B.I. director and his deputy, both of whom former President Donald J. Trump wanted prosecuted, were selected for a rare audit program that the tax agency says is random.

July 6, 2022

But, why?

Author | Topic: Bad signatures leading to 55.82152538 BTC theft (so far) (Read 64937 times)

BurtW (OP)
Legendary

Activity: 2646
Merit: 1104

All paid signature campaigns should be banned.

Bad signatures leading to 55.82152538 BTC theft (so far)
August 10, 2013, 10:53:13 PM
Merited by LoyceV (8), ETFbitcoin (6)
#1

I have only seen this discussed in the newbies section so I thought I would open a thread here for a more technical discussion of this issue.

Several people have reported their BTC stolen and sent to https://blockchain.info/address/1HKywxiL4JziqXrzLKhmB6a74ma6kxbSDj

As you can see the address currently contains 55.82152538 stolen coins.

It has been noticed that the coins are all transferred in a few hours after a client improperly signs a transaction by reusing the same random number. As discussed here:

http://en.wikipedia.org/wiki/Elliptic_Curve_DSA

the reuse of the same k value allows anyone to be able to recover the private key.

August 10, 2013

But, why?

December 30, 2010

ars technica

GAMING & CULTURE / GAMING & ENTERTAINMENT

PS3 hacked through poor cryptography implementation

A group of hackers named fail0verflow revealed in a presentation how they ...

by **Casey Johnston** - Dec 30, 2010 6:25pm CET

After beating several other security measures, the group was able to locate the PS3's ECDSA signature, a private cryptographic key needed to sign off on high-level operations. Normally, these kinds of keys are difficult to figure out, and require running many generations of keys to crack.

But when fail0verflow worked backwards from generated keys, they found out that a parameter that should have been randomized for each key generation wasn't being randomized at all. Instead, the PS3 was using the same number for that variable, every single time, making it easy to work out acceptable keys.

If this really works, it's a big slip on Sony's part. While PS3s are no stranger to software updates, this seems like it might affect operation on too many levels to be an easy fix.

Why:

The problems with randomness & how to avoid them

Chapter I

Chapter II

Chapter III

Chapter IV

Protocol Labs

“Randomness is hard”

This is something you’ll often hear when talking to applied cryptographers who have done some code assessments in their life.

In general it’s very important to have “proper” randomness, that is:

- **Unpredictable**: impossible to predict the next numbers
- **Bias-resistant**: the final output cannot be biased in any way

Protocol Labs

Why unpredictable?

- If you can predict the random value, you can “cheat” (gambling, games, …)
- If you can predict who’s going to be selected, fairness isn’t guaranteed anymore (think of leader election, sharding, jury selection, …)
- If you can predict “a secret key”, then the security of the system is compromised

Protocol Labs

Why unbiased?

For Schnorr-like signature schemes, such as ECDSA, DSA or EdDSA, a **nonce** (“number used only once”, also sometimes called “secret k”) with a bias on **less than one bit** will lead to **full key recovery attacks** from just seeing signatures! Attacks exploiting biased keys have been known and used in practice since 1999!

Protocol Labs

Remember that?

Author | Topic: Bad signatures leading to 55.82152538 BTC theft (so far) (Read 64937 times)

BurtW (OP)
Legendary

Activity: 2646
Merit: 1104

All paid signature campaigns should be banned.

Bad signatures leading to 55.82152538 BTC theft (so far)
August 10, 2013, 10:53:13 PM
Merited by LoyceV (8), ETFbitcoin (6)
#1

I have only seen this discussed in the newbies section so I thought I would open a thread here for a more technical discussion of this issue.

Several people have reported their BTC stolen and sent to https://blockchain.info/address/1HKywxiL4JziqXrzLKhmB6a74ma6kxbSDj

As you can see the address currently contains 55.82152538 stolen coins.

It has been noticed that the coins are all transferred in a few hours after a client improperly signs a transaction by reusing the same random number. As discussed here:

http://en.wikipedia.org/wiki/Elliptic_Curve_DSA

the reuse of the same k value allows anyone to be able to recover the private key.

August 10, 2013

Even worse

Paper 2019/023

Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies

Joachim Breitner and Nadia Heninger

Abstract

In this paper, we compute hundreds of Bitcoin private keys and dozens of Ethereum, Ripple, SSH, and HTTPS private keys by carrying out cryptanalytic attacks against digital signatures contained in public blockchains and Internet-wide scans. The ECDSA signature algorithm requires the generation of a per-message secret nonce. If this nonce is not generated uniformly at random, an attacker can potentially exploit this bias to compute the long-term signing key. We use a lattice-based algorithm for solving the hidden number problem to efficiently compute private ECDSA keys that were used with biased signature nonces due to multiple apparent implementation vulnerabilities.

2019: all vulnerable addresses were empty already…

Hands-on

In practice

Chapter I

Chapter II

Chapter III

Chapter IV

Protocol Labs

How to get a secret random byte?

https://go.dev/play/p/mED52o78h6P

```go
import (
  "crypto/rand"
  "fmt"
)

func main() {
  x := make([]byte, 1)
  if _, err := rand.Read(x); err != nil {
      fmt.Println("error:", err)
      return
  }
  fmt.Println(x)
}
```

Protocol Labs

How to get a secret random integer < 107?

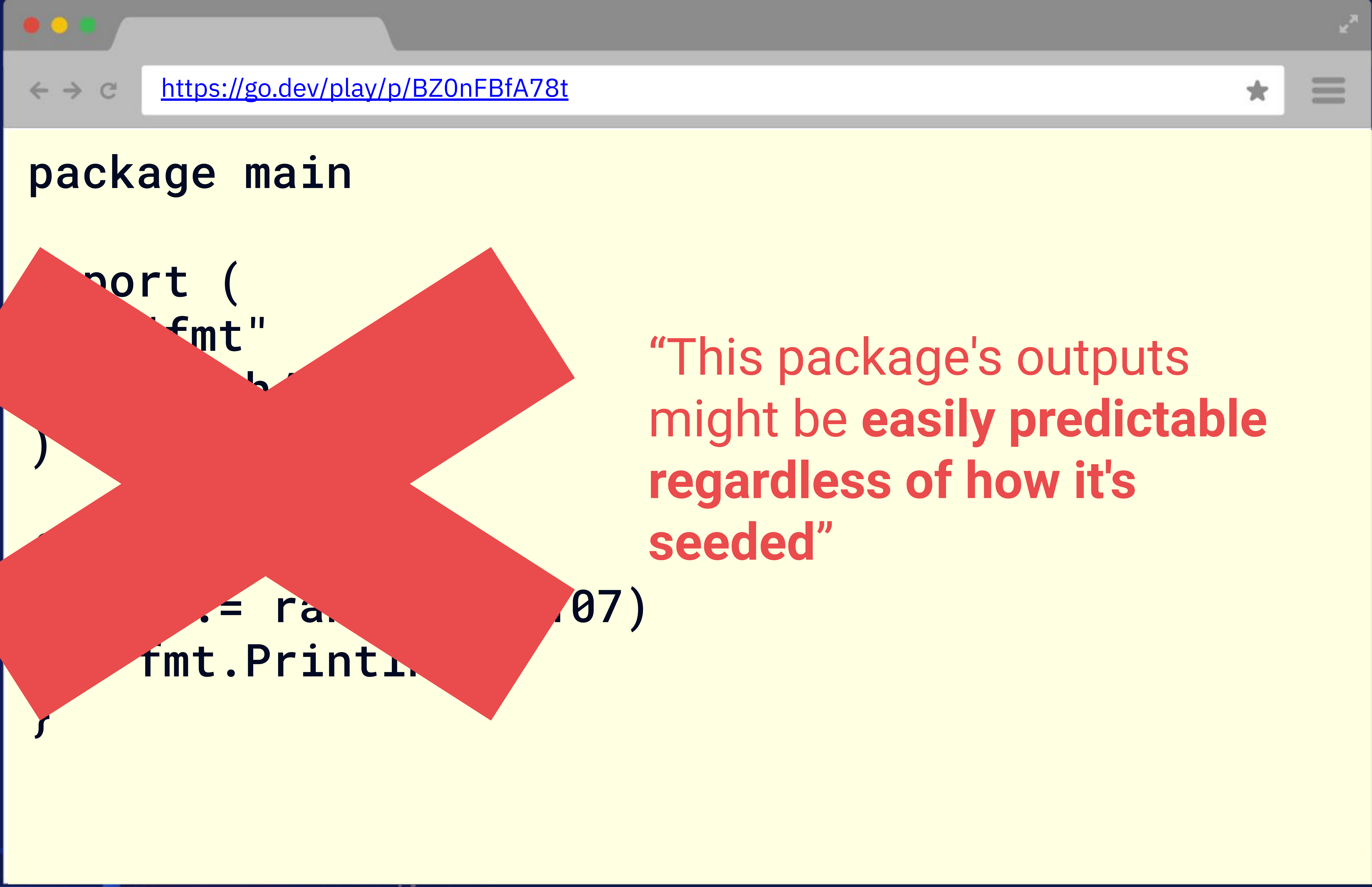
Maybe we can use “math/rand”?

https://go.dev/play/p/BZ0nFBfA78t

```
package main

import (
	"fmt"
	"math/rand"
)

func main() {
	x := rand.Intn(107)
	fmt.Println(x)
}
```

Protocol Labs

It's easy to have bad randomness



"This package's outputs might be **easily predictable regardless of how it's seeded**"

No, "math/rand" is not meant to be properly random!

Protocol Labs

How to get a secret random integer < 107?

Okay, let's take a byte and reduce it modulo 107 then!

https://go.dev/play/p/SRKRue9tU1H

```go
import "crypto/rand" [...]

func main() {
	b := make([]byte, 1)
	if _, err := rand.Read(b); err != nil {
		fmt.Println("error:", err)
		return
	}
	x := b[0] % 107
	fmt.Println(x)
}
```

Protocol Labs

How to get a secret random integer < 107?

No! Reducing it modulo 107 means we're now biased!

https://go.dev/play/p/eWtmS3vjWt9

```
import "crypto/rand" [...]

    main() {
         m            , 1)
                   rand.Read(b); err != nil {
                   ntln("error:", err)



    x := b[0]
    fmt.Println(x)
}
```

Reducing a random binary string to a value that is not a power of 2 introduces a **modulo bias**!

Protocol Labs

This is a Modulo Bias

Out of the **256 possible values** for a byte, from 0 to 255, if we reduce **modulo 107**, then the first 42 values are more likely to occur because 256 % 107 = 42

Protocol Labs

How to avoid bias?

Use the proper package “crypto/rand” whenever security relies on randomness (even for shuffling!!) and make sure to use either:

- probabilistically safe modulo (i.e. reduce a much larger value modulo)
- rejection sampling (i.e. keep picking a random value until it’s smaller than the biggest multiple of the max value of that bit length)
- Lemire’s divisionless method (not noticeably faster for CPRNGs sadly)

Read more about modulo bias in my post about it:

The definitive guide to “modulo bias and how to avoid it”

Protocol Labs

How to avoid bias?

```
package rand // import "crypto/rand"

Package rand implements a cryptographically secure random
number generator.

var Reader io.Reader
func Int(rand io.Reader, max *big.Int) (n *big.Int, err error)
func Prime(rand io.Reader, bits int) (p *big.Int, err error)
func Read(b []byte) (n int, err error)
```

Also, don’t use floats

Last time I explained this, someone asked me

> Am I suffering from modulo bias if I do:

```
>         rand.Read(b)
>         x :=  int(float32(b[0]) / 255.0 * 107)
```

Well, not the *modulo* bias per se, but this is still biased, yes.

Floating point arithmetic and precision is going to bias this in weird ways.

See playground demo: https://go.dev/play/p/iq16iCoeE8Q

See also: https://www.pcg-random.org/posts/bounded-rands.html

How to get a secret random integer < 107?

With rejection sampling now we're good!

https://go.dev/play/p/v6d3FwqS2Lv

```
import "crypto/rand" [...]

func main() {
   b := make([]byte, 1) //max value = 255
   x:= 255
   for x >= 255-255%107 { //the closest multiple
      rand.Read(b)
      x = int(b[0])
   }
   x %= 107
   fmt.Println(x)
}
```

Protocol Labs

How to get a secret random integer < 107?

Or using the proper method `rand.Int` from “crypto/rand”. It uses rejection sampling under the hood.

https://go.dev/play/p/bI_7_oiFCyr

```go
import (
  "crypto/rand"
  "fmt"
  "math/big"
)

func main() {
  x, _ := rand.Int(rand.Reader, big.NewInt(107))
  y := x.Int64()
  fmt.Printf("x=%d of type %T\n", x, x)
  fmt.Printf("y=%d of type %T\n", y, y)
}
```

Protocol Labs

This was for “secret” or “local” randomness

> Okay, but how about if I want to run a lottery and don’t want people saying I’ve rigged my PRNG or my “crypto/rand” package in case one of my friends wins?

This was for “secret” or “local” randomness

> Okay, but how about if I want to run a lottery and don’t want people saying I’ve rigged my PRNG or my “crypto/rand” package in case one of my friends wins?

i.e. “How do we do to get some public, verifiable randomness?”

History: Prior art

"RANDOM.ORG offers true random numbers to anyone on the Internet. The randomness comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs. People use RANDOM.ORG for holding drawings, lotteries and sweepstakes, to drive online games, for scientific applications and for art and music. The service has existed since 1998"

Protocol Labs

History: The NIST Beacons

- The idea of running a public, verifiable “trusted” randomness beacon was first proposed by NIST in 2011
- Their NIST Beacon v1 was launched on 2013-09-05
- Their NIST Beacon v2 was launched in 2019: https://doi.org/10.6028/NIST.IR.8213-draft

Previous attempts to generate public randomness

Can we do *simpler & faster* than before?

drand provides a public randomness service, just like we have:

- **DNS**: Highly available source of naming information
- **NTP**: Highly available source of timing information
- **PKIs**: Trusted network delivering certificates
- **Certificate transparency**: Certificate authenticity information

➡️ **Drand**: Highly available, decentralized, and publicly verifiable source of randomness introduced in 2019, launched for safe general availability in 2020.

Drand properties

- Drand is software run by a set of independent nodes that collectively produce randomness
- Drand is **open source**[1], coded in Go, supported by Protocol Labs
- **Decentralized randomness service** using
  - Distributed Key Generation based on [Verifiable Secret Sharing]
  - Threshold cryptography
  - Key is defined on G2 of the **BLS12-381 pairing curve**, achieves ~128 bits of security
- Binds together **independent entropy sources** into a publicly verifiable one
- Tested, **audited**, and deployed (more on that later)

[1] https://github.com/drand/drand

Drand properties

Decentralized: a threshold of nodes operated by different parties is needed to generate randomness; there is **no central point of failure.**

Unpredictable: no party learns anything about the output of the round **until a sufficient number of drand nodes** reveals their contributions thanks to **threshold cryptography**.

Bias Resistant: the output represents an **unbiased, uniformly random value.**

Verifiable: the random output is **third-party verifiable** by verifying the **aggregate BLS signatures** against the collective public key computed during setup.

The League of Entropy

16 organizations

2M+ rounds

23 nodes

2 years

4 public endpoints

0 disruptions since 2020-08-10

0 single point of failure

LEAGUE OF ENTROPY

CLOUDFLARE

c·Labs

The Initiative for CryptoCurrencies and Contracts

TIERION

Universidad de Chile

KUDELSKI SECURITY

ChainSafe

UCL

Protocol Labs

EPFL

ptisp

C4DT

Protocol Labs

Left blank on purpose for captionning system.

The “entropy”

The only moment where fresh entropy is required is during the Distributed Key Generation.

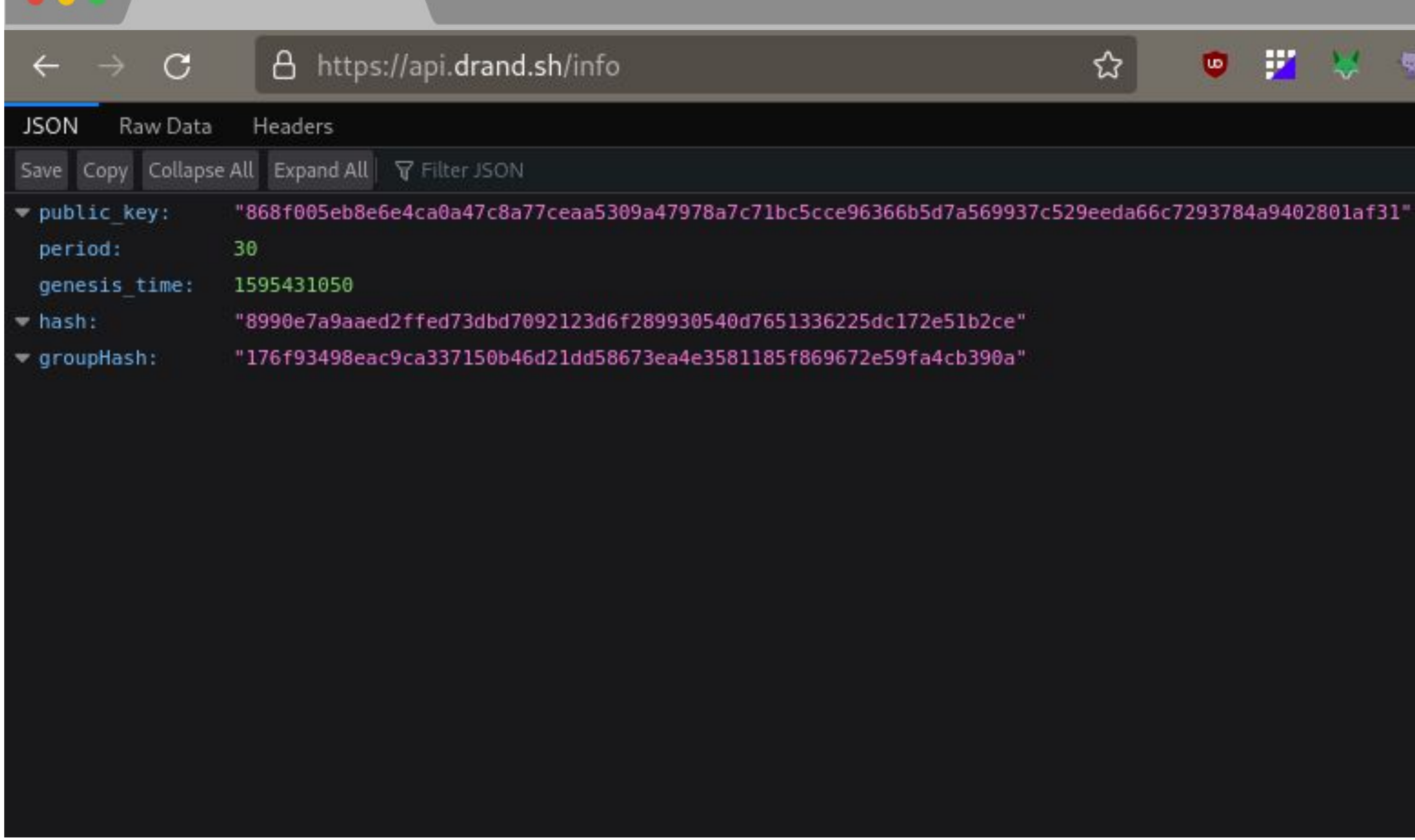
Some partners are getting their entropy from so-called “TRNG”, based on physical properties known to be unpredictable.

Lava lamps in the Cloudflare lobby. Courtesy of @mahtin

Public API: web endpoints

You can test it in your browser:

[*https://api.drand.sh/public/latest*](https://api.drand.sh/public/latest)



`curl https://api.drand.sh/public/latest`

Protocol Labs

Left blank on purpose for captionning system.

How to get public, verifiable randomness?

Just **use a verifying client directly**:

- Go: import “github.com/drand/drand/client”
- TS: https://github.com/drand/drand-client/
- Rust: https://github.com/CosmWasm/drand-verify

Protocol Labs

How to get public, verifiable randomness?

Using the verifying client directly, easy!

```
[]
  "github.com/drand/drand/client"
  "github.com/drand/drand/client/http"

func main() {
  cHash, _ :=
hex.DecodeString("8990e7a9aaed2ffed73dbd7092123d6f289930540d7651336225dc172e51b2ce")
  c, _ := http.New("https://api.drand.sh/", cHash, nil)
  v, _ := client.Wrap([]client.Client{c}, client.WithChainHash(cHash))

  r, _ := v.Get(context.Background(), 0)
  fmt.Println(r)
}
```

Protocol Labs

How to get public randomness?

We can use the public endpoints!

```go
import  "github.com/drand/drand/client/http" [...]

func main() {
  var chainHash, _ =
hex.DecodeString("8990e7a9aaed2ffed73dbd7092123d6f289930540d7651336225dc172e51b2ce")
  // create new client for url and chainhash
  c, _ := http.New("https://api.drand.sh/", chainHash, nil)
  // get the latest round of randomness
  r, _ := c.Get(context.Background(), 0)

  fmt.Printf("Round n°%d, random=\n%x\n", r.Round(), r.Randomness())
}
```

Round n°2324934, random=
5eb6f4d9fae65b4c6d94967dbb7c444860d01e60a3f34938ab495bb5c
a098167

Protocol Labs

How to get public, verifiable randomness?

First we can derive the randomness from the signature

```
import  "github.com/drand/drand/client/http" [...]

func main() {
  var chainHash, _ =
hex.DecodeString("8990e7a9aaed2ffed73dbd7092123d6f289930540d7651336225dc172e51b2ce")
  // create new client for url and chainhash
  c, _ := http.New("https://api.drand.sh/", chainHash, nil)
  // get the latest round of randomness
  r, _ := c.Get(context.Background(), 0)

  fmt.Printf("Round n°%d, random=\n%x\n\n", r.Round(), r.Randomness())

  h := sha256.New()
  h.Write(res.Signature())
  derRd := h.Sum(nil)
  fmt.Printf("Randomness from hash is indeed:\n%x\n", derRd)
}
```

Protocol Labs

How to get public, verifiable randomness?

And it checks out!

```
Round n°2324946, random=
1da0b7a31db72fe3526fb437ef241763143d2baeb1df434be70ef63c68e15ab0

Randomness from hash is indeed:
1da0b7a31db72fe3526fb437ef241763143d2baeb1df434be70ef63c68e15ab0
```

Protocol Labs

How to get public, verifiable randomness?

But when we're talking about signatures, we should first verify them, right?

```
import  ( bls12381 "github.com/drand/kyber-bls12381" [...]
  pKB, _ :=
hex.DecodeString("868f005eb8e6e4ca0a47c8a77ceaa5309a47978a7c71bc5cce96366b5d7a569937c529eeda66c7293784a9402801af31")

  suite := bls12381.NewBLS12381Suite()
  pk := suite.G1().Point()
  pk.UnmarshalBinary(pKB)

  buf := make([]byte, 8)
  binary.BigEndian.PutUint64(buf, r.Round())
  h := sha256.New()
  // we need the previous signature!
  p, _ := c.Get(context.Background(), r.Round()-1)
  h.Write(p.Signature())
  h.Write(buf)
  // we finally get the signed message
  msg := h.Sum(nil)
```

Protocol Labs

How to get public, verifiable randomness?

But when we're talking about signatures, we should first verify them, right?

```
import "github.com/drand/kyber/sign/bls" [...]

  // Finally we can verify the signature!
  err = bls.NewSchemeOnG2(suite).Verify(pk, msg, r.Signature())
  if err != nil {
    fmt.Println("Signature didn't verify.")
  } else {
    fmt.Println("Signature verified.")
  }

  // and derive the randomness out of the signature
  h.Reset()
  h.Write(res.Signature())
  derRd := h.Sum(nil)
  fmt.Printf("Randomness from hash is indeed:\n%x\n", derRd)
}
```

Protocol Labs

How to use verifiable randomness?

So, we got a random bytestring, but how do we use it?
We need to "derive" our values from it!

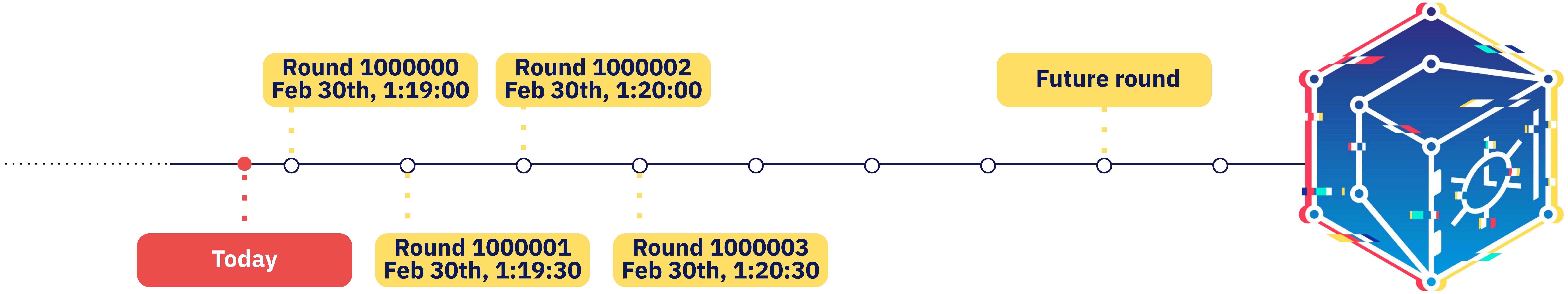
```
[...]

   derRd := h.Sum(nil)
   x:= int(derRd[0])
   for x >= 255-255%107 {
      h.Reset()
      derRd = h.Sum(derRd)
      x = int(derRd[0])
   }
   x %= 107
   fmt.Println(x)
}
```

Protocol Labs

Bonus: timelock encryption!

Relying on drand, we've released two open-source libraries and clients to do timelock encryption: **encrypt now towards the future**!

Using the fact that drand (and thus the League of Entropy) produces new **signed** rounds every 30 seconds in a reliable way.



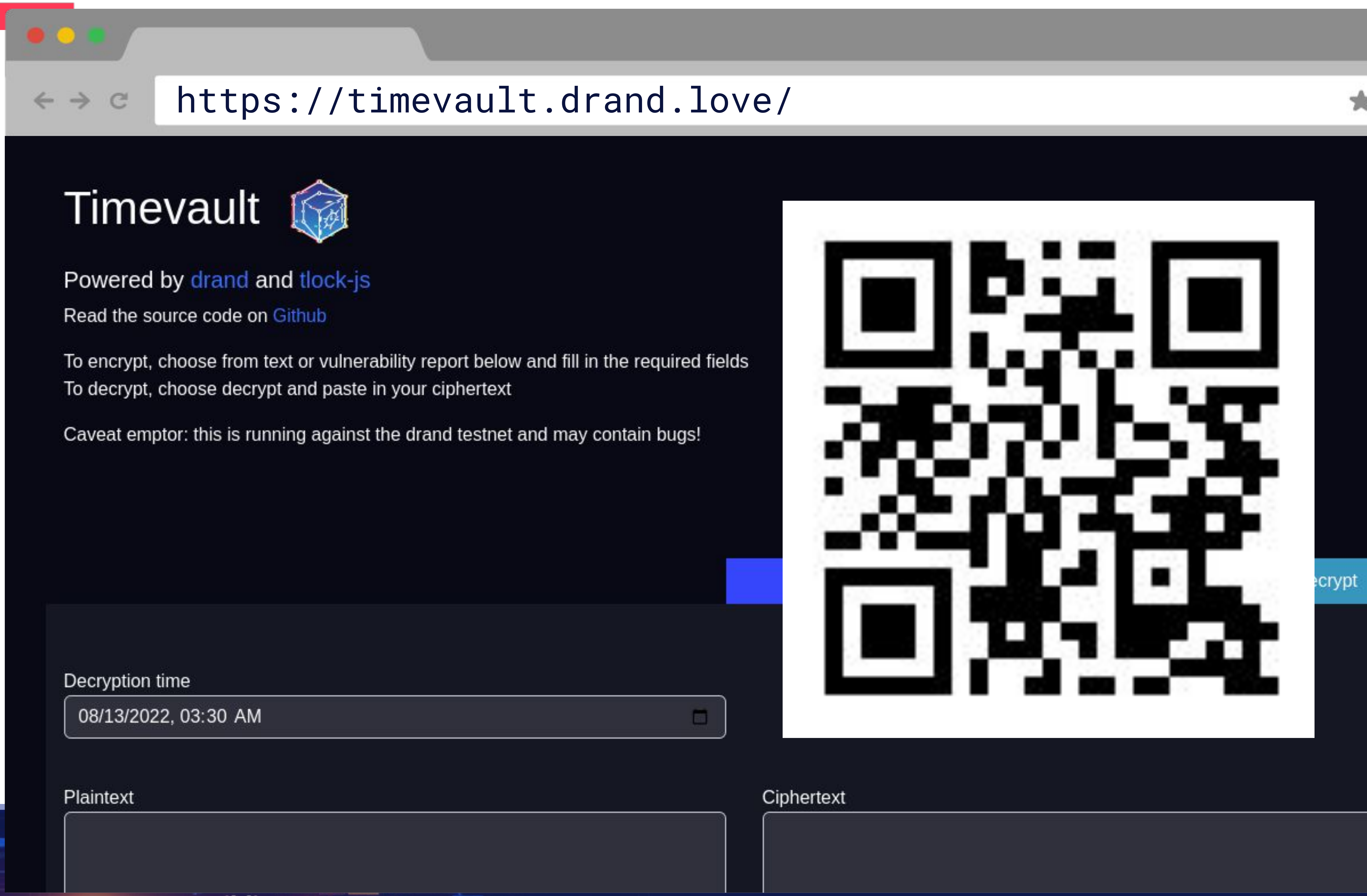
tlock

- Released in August: “A dead man’s full-yet-responsible disclosure system”

DEFCON

- Described in more gory math details my [public Research seminar](#).
- More details on [this public page](#).
- Incoming ePrint paper!

[timevault.drand.love](#)



Grow the League!

- Join the League of Entropy, help us provide free public randomness.
- We are looking for partners running nodes or relays.
- Infrastructure and operational requirements are minimal: Estimated commitment: 1-2 hours/month, 1vCPU, 512MB RAM

https://drand.love/partner-with-us/

Thank you !

For more information and/or if you want to reach out, go to:

https://drand.love

https://leagueofentropy.com

https://github.com/drand/drand

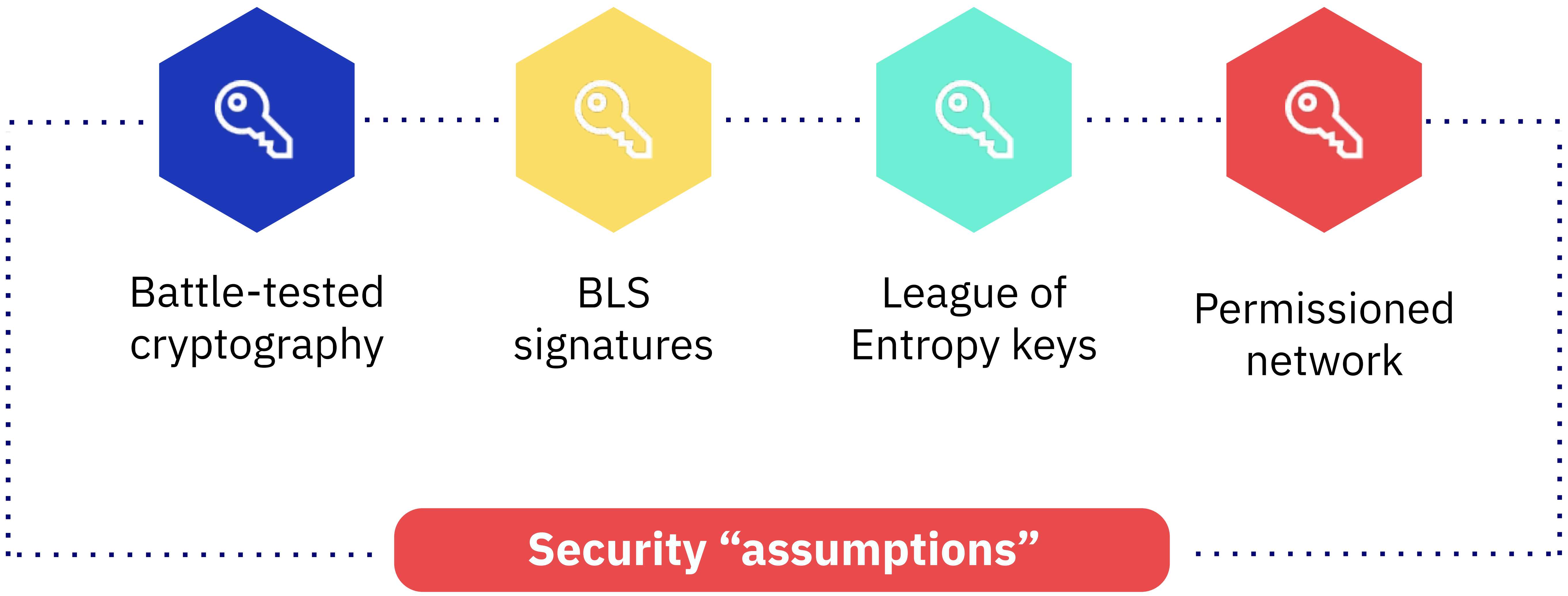
Email
yolan@protocol.ai

Twitter
https://twitter.com/anomalroil

Threats & security

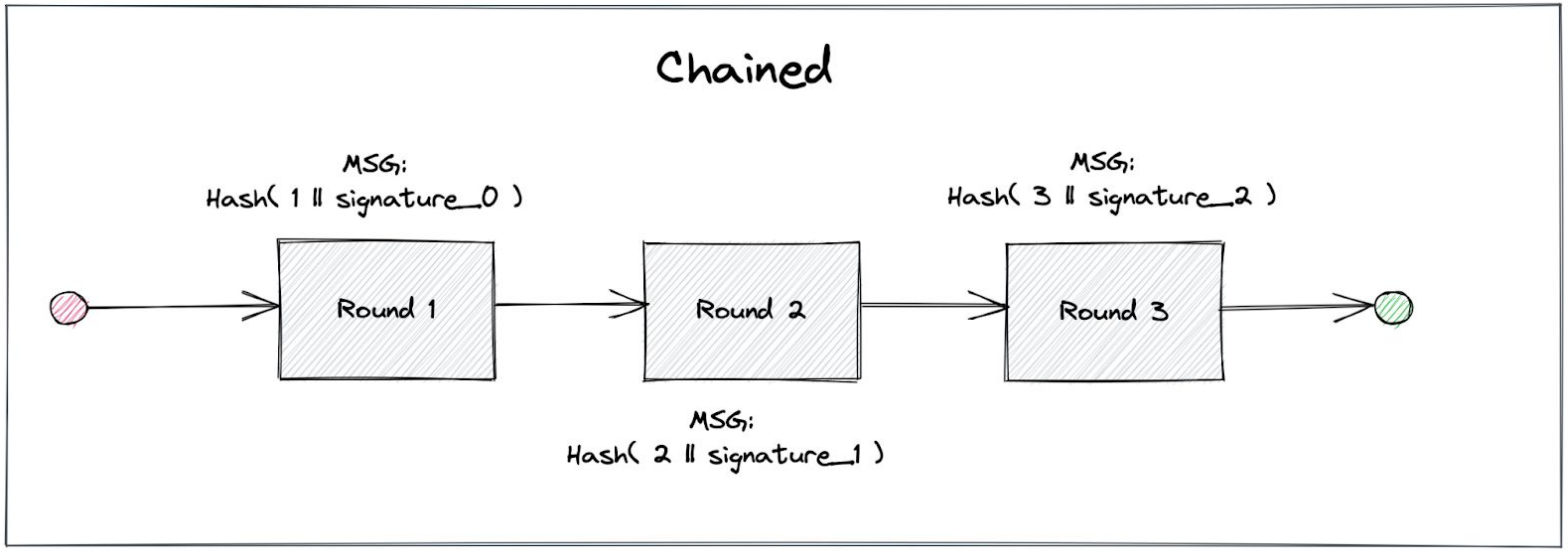
If you trust there are **never more than a threshold number of malicious nodes** on the (drand) network you're relying on, you're good to go!

UNLESS! Somebody builds a quantum computer breaking modern schemes, since the BLS signature scheme isn't quantum resistant.
(Which means we're probably safe for the next 5-10 years, maybe even 20.)



Chained Randomness

Current randomness is **chained**:

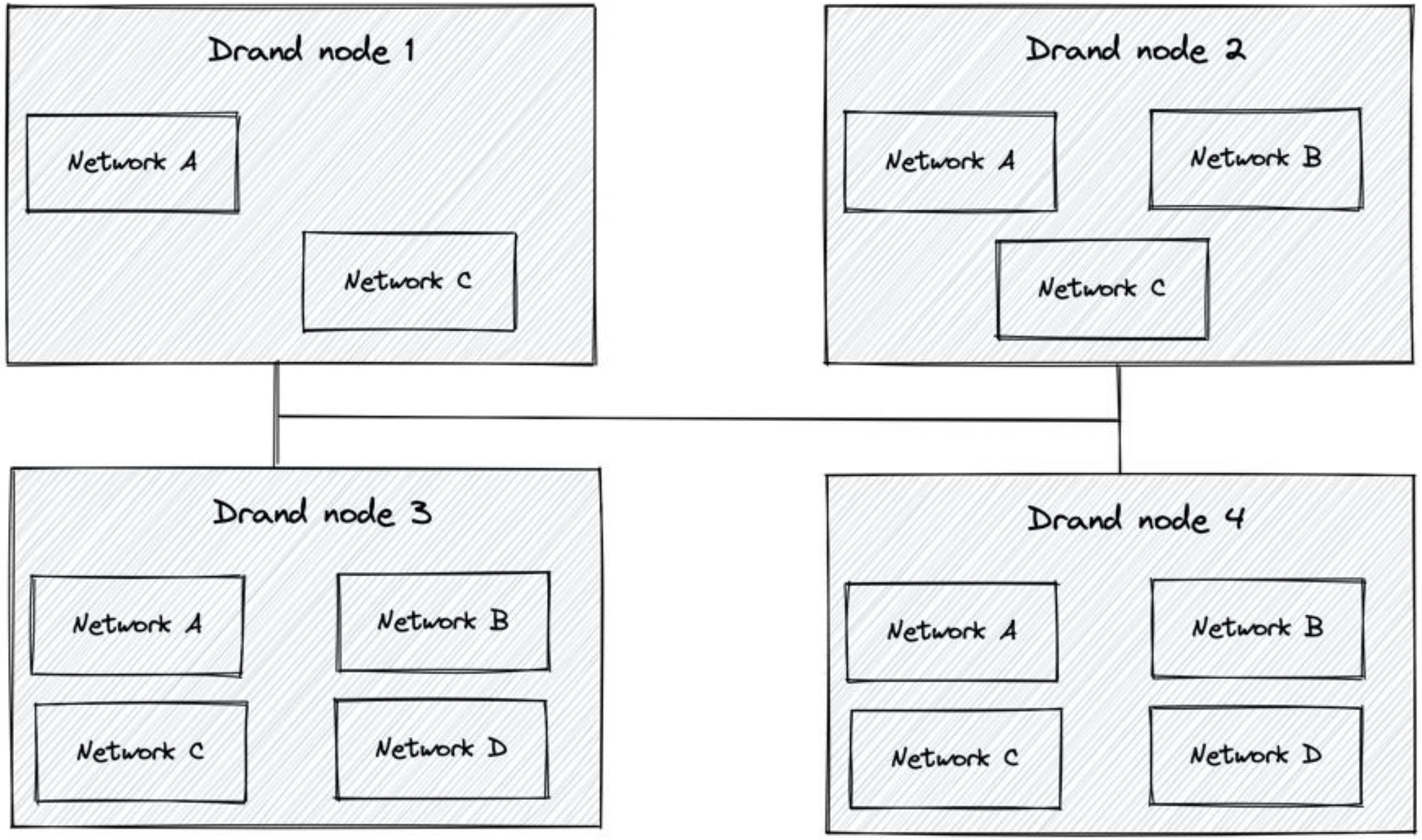


Multi Protocol

We can now have different protocols for different use cases in parallel!

Current target: have a higher frequency network

This was just launched on our testnet!



Unchained Randomness

New **unchained randomness**: