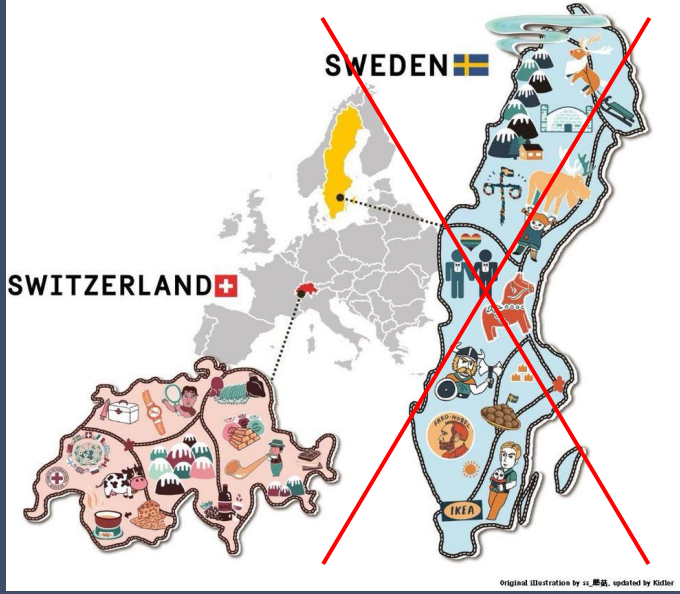
Have you distributed randomness?

Yolan Romailler
@AnomalRoil

August 2019 — BSidesLV

$ whoami



⟶ Cryptographer at Kudelski Security in Switzerland

⟶ CTF player (mostly crypto, forensic & misc)

⟶ Board games player

⟶ Speaker at Black Hat, DEF CON, North Sec, ... 2nd time at BSidesLV!

⟶ Go coder (“one day I’ll have time, yknw”)

Agenda

- What is distributed randomness
- What is drand
- ~~Questions~~

What is randomness?

If I take a random binary string, each bit has probability 0.5 to be set to 1 or to 0.

Alright, are these strings random then?

1000000000000000
0000000011111111
0000000100000000

Well, maybe, maybe not! That's the point of randomness, but while the probability of drawing these strings at random is very low, each is as probable as any other fixed binary string of the same length.

Why do we want randomness?

Many use-cases:

- Lotteries
- Election
- Computer applications (simulation of stochastic systems, numerical analysis, probabilistic algorithms, computer games, …)
- Statistics (random sampling has to be… random!)
- Cryptography & secure communications!

And yet computers and their programs are fundamentally deterministic beasts!

Now how about distributed, verifiable randomness?

Usage of “shared” randomness:

- Tor
- (Blockchain) sharding
- Smart contract lotteries

Usage of “verifiable” randomness:

- Elliptic curve parameters (remember Dual_EC_DRBG?)
- Sortitions (think of how the law court juries are constituted in the USA)

Both notions are there to help with the auditability and transparency of services that depend on random values.

BTW, Verifiable Random Function (VRF) is a thing: it’s the public-key version of keyed hash functions, and it’s got its very own IETF draft!

Wish for “randomness beacons”

As we have seen, it is desirable to have randomness beacons that would in particular be:

- **Unpredictable**: impossible to predict the next numbers
- **Bias-resistant**: the final output cannot be biased in any way
- **Publicly verifiable**: anybody can verify output is a “legit” random number
- **Decentralized**: output is produced by a set of (independent) parties.
- **Available**: the system must always be able to deliver random numbers

How about previous works?

Back then, existing providers of public randomness had to be trusted third-parties!

A new solution: drand

drand aims to address the need for distributed public randomness:

- drand is a software meant to be run by a set of independent nodes that will then collectively produce verifiable randomness
- drand allows to build a standalone randomness-as-a-service network
- ideally fetching randomness should be as easy as fetching time with NTP

drand nodes can provide both private randomness and public randomness that is:

- unpredictable and bias-resistant
- verifiable

The history behind drand

- ➔ < 2015: NIST prototype randomness beacon provides public randomness
- ➔ 2015: Ewa Syta with the DEDIS team at EPFL started working on Scalable Bias-Resistant Distributed Randomness, resulting in a published paper in 2017
- ➔ 2017: the DEDIS team started collaborating with DFINITY on various research topics, and integrated their pairing implementation into DEDIS' Kyber library.
- ➔ Septembre 2017: Nicolas Gailly, then a PhD student at DEDIS, started coding drand, using DEDIS' Kyber library
- ➔ 2018: more partners are interested by the idea of running randomness beacons
- ➔ June 2019: official launch of the League of Entropy, whose nodes are running drand and whose members are distributed over the globe.

The theory behind drand

Based on previous works presented in the “Scalable Bias-Resistant Distributed Randomness” paper.

Let us take a look at its different components!

Building blocks

Drand relies on the following cryptographic constructions:

- Feldman’s Verifiable Secret Sharing scheme
- Pedersen's distributed key generation protocol for the setup
- **Pairing-based cryptography** with Barreto-Naehrig curve BN256
- **Threshold BLS signatures** for the generation of public randomness
- Resharing scheme from a paper by Y. Desmedt and S. Jajodia
- **ECIES** for the encryption of private randomness

Shamir Secret Sharing (t, n) & VSS

Basic idea of SSS is that 2 points define a line, 3 points can define a parabola,...
I.e. it takes **t** points to uniquely define a polynomial of degree **t-1.**

Protocol:

- **A dealer** creates a polynomial **f(x)** of degree **t-1**, with its first coefficient being the secret value **s** we want to split into shares. That is, **f(0) = s**
- The dealer sends to each of the **n** shareholders their share **f(i)** for the node **i**
- The polynomial can be reconstructed out of **t** shares using Lagrange interpolation.

NB: SSS scheme assumes the dealer is honest. **SSS is centralized!**

Verifiable Secret Sharing (VSS) schemes protect against malicious dealers.

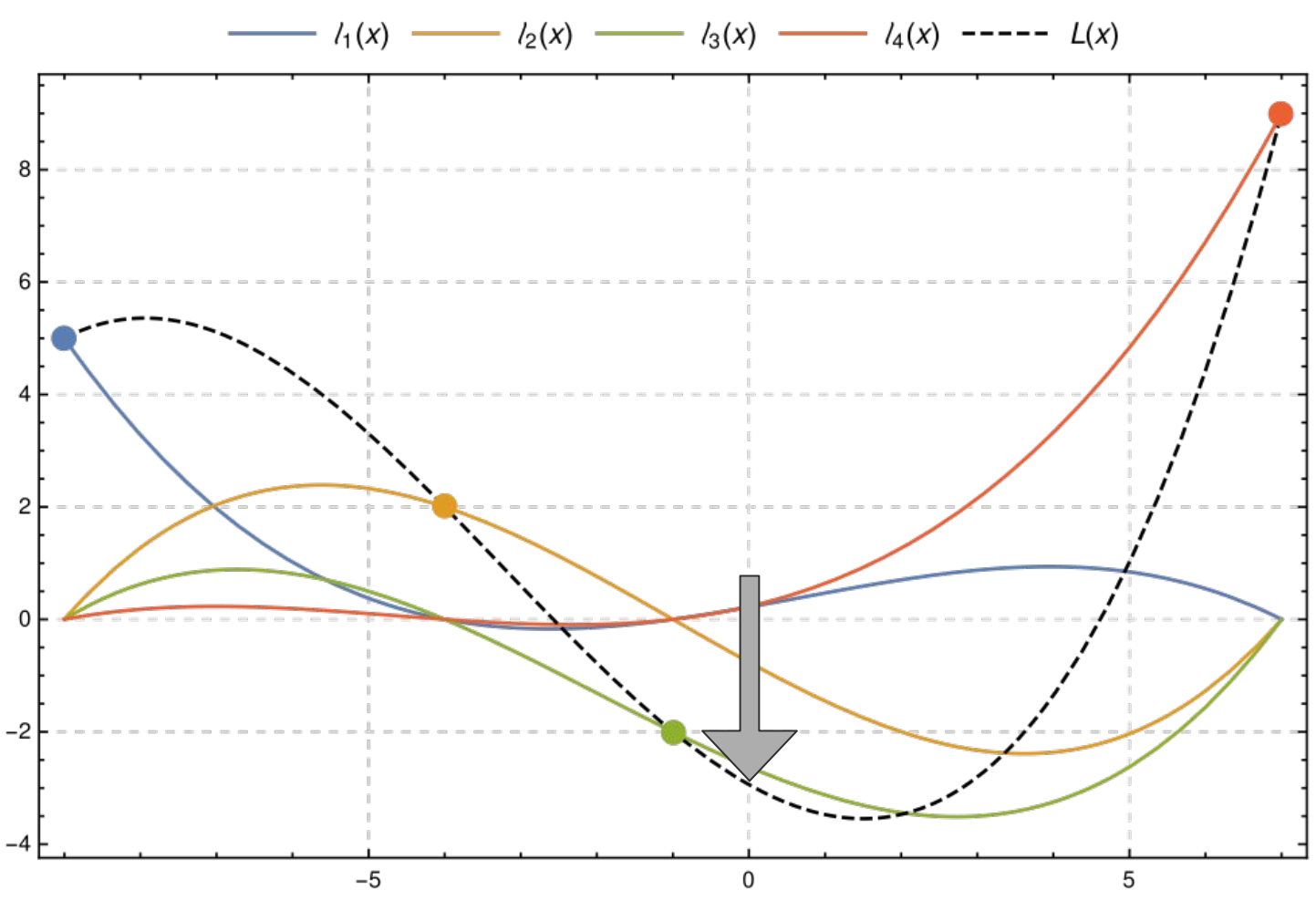


Image credit: modified version from wikipedia.org

Distributed Key Generation

DKG enables to “threshold-ize” many cryptographic primitives:

- digital signatures (Schnorr signatures, BLS …)
- asymmetric encryption (ElGamal encryption)

drand relies on Pedersen’s DKG scheme which essentially runs n instances of Feldman’s VSS in parallel on top of some additional verification steps.
This allows each node to verify the shares its received during setup, and detect and invalid dealer.

NB: the secret from Pedersen’s DKG can be biased, but it is safe to use for threshold signing as formally shown by Rabin et al.

Recall - (t-n) Distributed Key Generation

- **Goal**: Create shares of a private key that no party knows nor computed, with at least t shares needed to reconstruct the private key
- **Idea**: Run n secret sharing protocol in parallel and each node adds all its shares
- **Secret key s** = $\sum$ $\mathbf{s_i}$ is recoverable by using Lagrange Interpolation on t shares $\mathbf{s_i}$
- **Public key** is **P = s * G** with
  - **G** a generator of the group
  - **P** is publicly distributively computed by sharing commitments $F_i(x) = f_i(x) * G$

$f_1(x) = \mathbf{s_1} + a_{1,1} * x + \ldots + a_{1,t-1} * x^{t-1}$ →

$f_2(x) = \mathbf{s_2} + a_{2,1} * x + \ldots + a_{2,t-1} * x^{t-1}$ →

$f_n(x) = \mathbf{s_n} + a_{n,1} * x + \ldots + a_{n,t-1} * x^{t-1}$ →

	$s_{1,1} = f_1(1)$	$s_{1,2} = f_1(2)$	...	$s_{1,n} = f_1(n)$
+	$s_{2,1} = f_2(1)$	$s_{2,2} = f_2(2)$	...	$s_{2,n} = f_2(n)$
+	$s_{n,1} = f_n(1)$	$s_{n,2} = f_n(2)$	...	$s_{n,n} = f_n(n)$
=	$\mathbf{s_1}$	$\mathbf{s_2}$	...	$\mathbf{s_n}$

Slide credit: Nicolas Gailly

Pairing-based Cryptography

Pairing-based crypto needs 3 cyclic groups of prime order **q** :
$\mathbb{G}_1$, $\mathbb{G}_2$ that are additive groups and $\mathbb{G}_t$ a multiplicative group with generators g_1, g_2, and g_t, respectively.

A pairing is a map **e**: $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$ with these properties:

- Bilinearity: $\forall a,b \in \mathbb{Z}_q^*$, $\forall P \in \mathbb{G}_1$, $\forall Q \in \mathbb{G}_2$, we have **e**$(aP,bQ)$=**e**$(P,Q)^{ab}$
- Non-degeneracy: **e ≠ 1**
- For practical purposes **e** has to be computable in an efficient manner

Recall digital signatures

A digital signature scheme is a public-key scheme with a key pair **(pk, sk)**, where **pk** is a public key and **sk** is a private key.

Messages that are signed with the **sk** can then be verified with **pk**.

Being given message **m** and signature **s**, it is impossible to recover the secret key **sk**, even if one does know the public key **pk** and message **m**.

Furthermore without the **sk** it is not possible to compute a valid signature under public key **pk** for any message **m'**, even being given a signature for message **m**.

Boneh-Lynn-Shacham (BLS) signatures

drand relies on BLS signatures to produces its randomness.

- ➔ Signature scheme proposed in 2004
- ➔ Pairing-based signatures, known for their short size
- ➔ Deterministic (only depends on the signer's key and the message)

BLS has nice properties:

- ➔ Enables signature aggregation
- ➔ Enables threshold signing
- ➔ Obvious but important: signatures are unif. distributed in the **signature space**

BLS: Signing

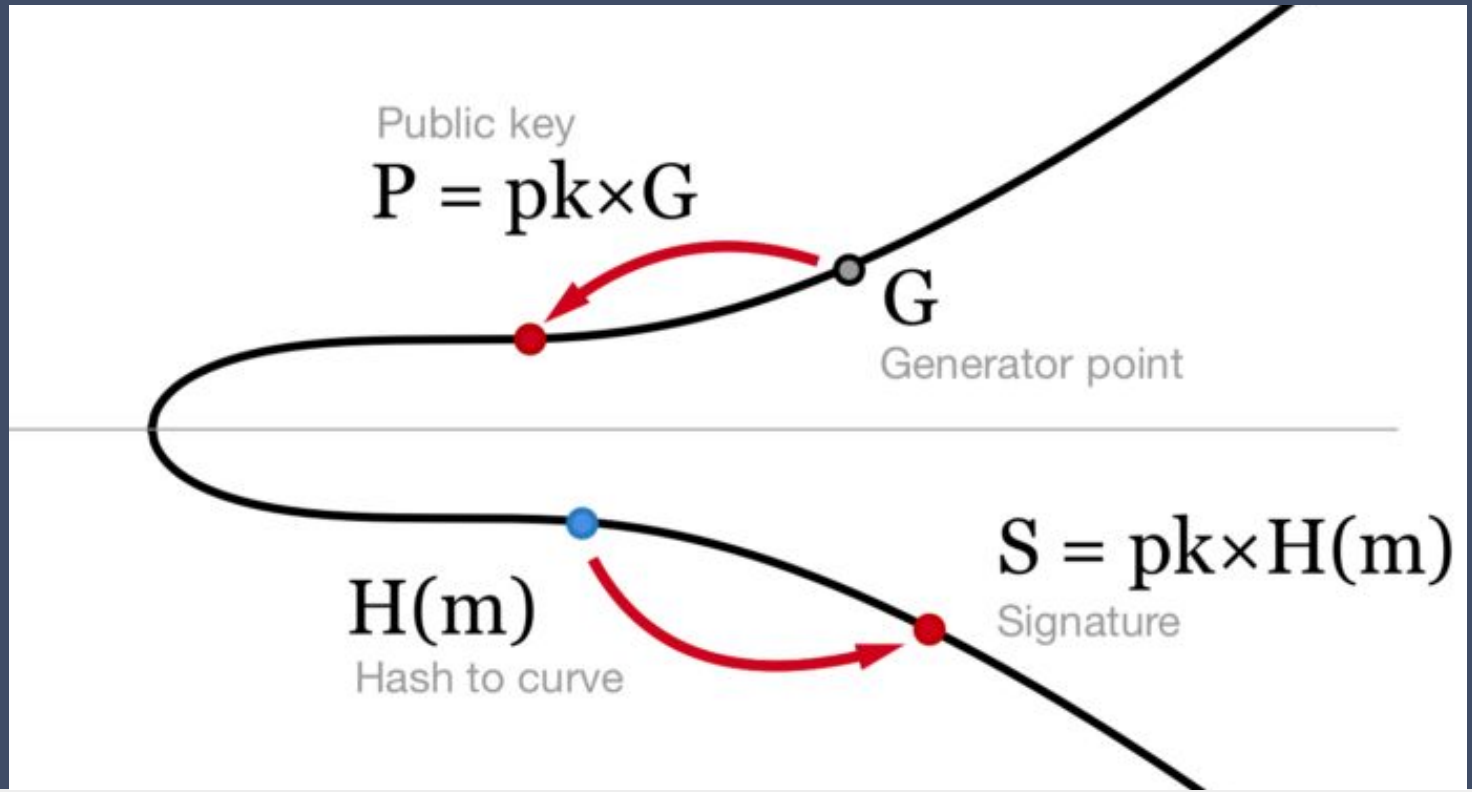


image credit: cryptoadvance on Medium

BLS: verifying

Thanks to the bilinearity of our pairing **e**, we can verify a signature by checking that:

e(P, H(m)) = e(G, S) holds.

If the signature has been signed using the private key pk corresponding to **P = pk·G**, then it holds since: **e(P, H(m)) = e(pk·G, H(m)) = e(G, pk·H(m)) = e(G, S)**

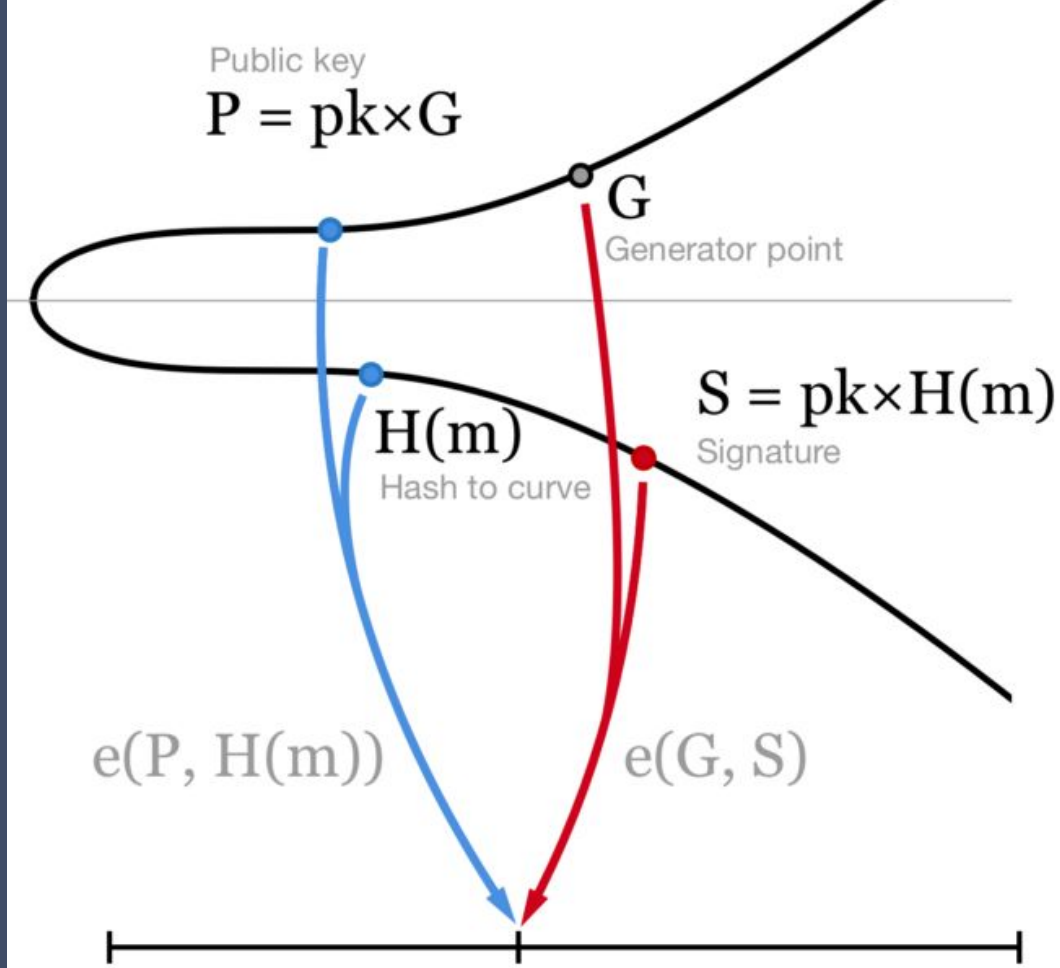


image credit: cryptoadvance on Medium

Threshold BLS Signatures

- ➔ When using secret sharing to split a BLS secret key, the shares themselves are valid BLS keys!
- ➔ When signing a message with them, one gets a valid signature for the public key that can be associated to that share.
- ➔ Taking **t** such share's signatures, which we call "**partial signature**", one can use Lagrange interpolation, just like with secret sharing, to reconstruct a signature that will verify under the main, shared public key.
  - ◆ NB: this means the shared secret key is never in memory during signing!

drand: setup of the protocol

This phase is expensive since it is in **$O(n^2)$**, but it must only be run once.

It is at that point that the Pedersen's Distributed Key Generation protocol is used.

At the end of the setup phase, each node **i** has:

- A share **s_i** of the distributed secret **s**
- The associated public key of the secret **P**

NB: the threat model assumes an attacker controls no more than **f** out of **n** nodes (for **f < n/2**)

drand: generation of randomness

Randomness generation itself is a **threshold BLS signature** protocol on *the previous round's signature* using the shares from the setup phase!

- The final, reconstructed signature is currently the beacon's output, since we want it to be verifiable BUT this is only well distributed over *the signature space*, which is not equivalent to random bytes, so one needs to hash the output to get bytes that are uniformly distributed out of that BLS signature.
- In the next version, the client fetching randomness, for now both drandjs and the CLI, will hash the signature after having verified it, in order to give actually random bytes to its users.

Chained Randomness

The drand beacons are synchronized in rounds of 1 minute. In every round drand produces a new random value using threshold BLS signatures linked together in a “chain of randomness” (not a blockchain, it got no blocks :P)

1. At round **r**, each node creates a partial BLS signature $\mathbf{u}_r$ using its private key and signs the previous full BLS signature $\mathbf{\Sigma}_{r-1}$ along with the current round index **r**
2. These partial signatures $\mathbf{u}_r$ are broadcast and every node can reconstruct the full signature $\mathbf{\Sigma}_r$ once they got **t** partial signatures

seed Σ_{-1}= "Truth is like the sun. You can shut it out for a time, but it ain't goin' away."

The features of drand

- ➔ Unpredictable, bias-resistant (thanks to BLS properties)
- ➔ Publicly verifiable (we can guarantee at least **t** nodes participated)
- ➔ Decentralized
- ➔ Available
- ➔ Fast: only needs RTT time + Lagrange interpolation

- ➔ Open source ! Check it out on Github: https://github.com/dedis/drand
- ➔ Provided with a JS example and a CLI tool to query nodes

The League of Entropy

A consortium of organizations and individuals that are running drand nodes together to form a distributed network.

It currently includes:

- Cloudflare
- EPFL
- Kudelski Security
- Protocol Labs
- UChile
- Individual nodes

Image credit: Cloudflare

Using cool source of entropy

Also, drand makes it easy to integrate your own source of entropy instead of using /dev/urandom !

For instance, Cloudflare is using its lava lamps to gather additional entropy, UChile is using seismic data, etc.

Have you got a cool one? Well...

Resharing allows for new members!

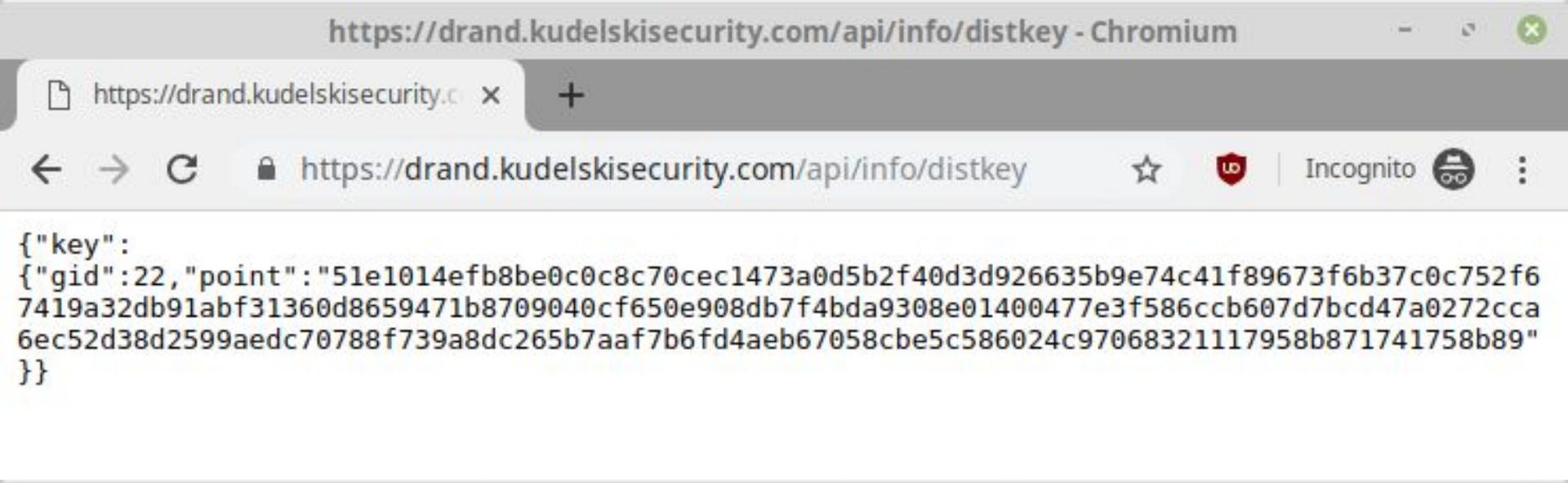
Another nice feature of drand is that we can perform resharing, which allows to on-board new members into an existing network such as the League of Entropy!
It allows to change the threshold **t** if so wished and to distribute new shares

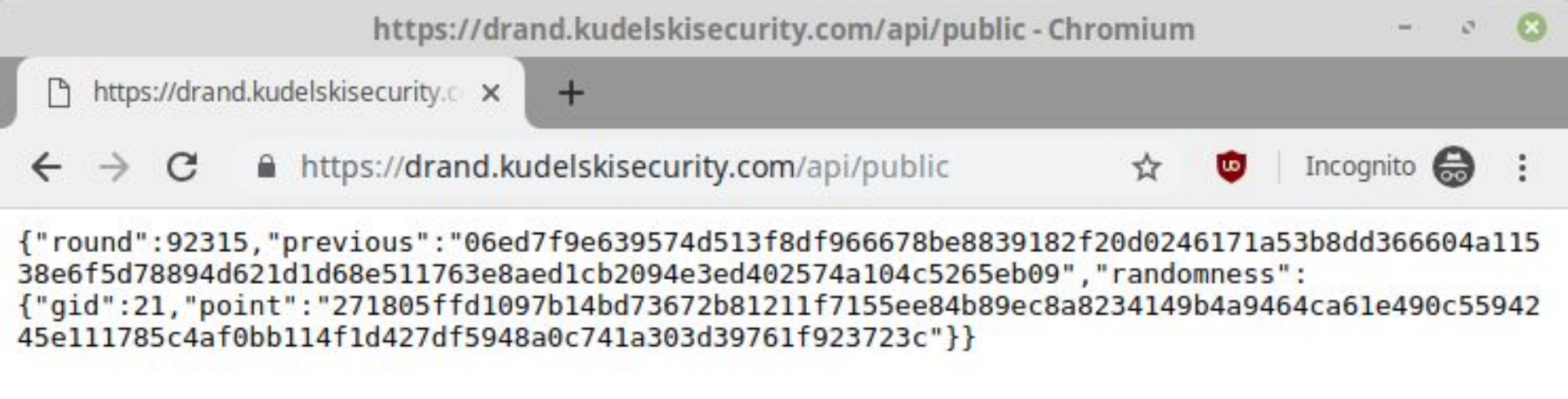
Wanna join us? Got a cool source of entropy?

Mail us at leagueofentropy@googlegroups.com and we'll get back to you for the second round of the DKG meant to on-board more members.

How to use drand?

LIVE DEMO TIME!


https://drand.kudelskisecurity.com/api/info/distkey - Chromium
https://drand.kudelskisecurity.c
https://drand.kudelskisecurity.com/api/info/distkey
Incognito
{"key":
{"gid":22,"point":"51e1014efb8be0c0c8c70cec1473a0d5b2f40d3d926635b9e74c41f89673f6b37c0c752f6
7419a32db91abf31360d8659471b8709040cf650e908db7f4bda9308e01400477e3f586ccb607d7bcd47a0272cca
6ec52d38d2599aedc70788f739a8dc265b7aaf7b6fd4aeb67058cbe5c586024c97068321117958b871741758b89"
}}


https://drand.kudelskisecurity.com/api/public - Chromium
https://drand.kudelskisecurity.c
https://drand.kudelskisecurity.com/api/public
Incognito
{"round":92315,"previous":"06ed7f9e639574d513f8df966678be8839182f20d0246171a53b8dd366604a115
38e6f5d78894d621d1d68e511763e8aed1cb2094e3ed402574a104c5265eb09","randomness":
{"gid":21,"point":"271805ffd1097b14bd73672b81211f7155ee84b89ec8a8234149b4a9464ca61e490c55942
45e111785c4af0bb114f1d427df5948a0c741a303d39761f923723c"}}

The risks of using (drand) || (the League of Entropy)

It is super important to notice that:

- drand is in beta right now, and has not been thoroughly reviewed
  - NB: the current output is only uniformly distributed in the signature space, you need to hash it!
- it provides PUBLIC randomness, i.e. do NOT generate secret keys with it

Furthermore, I need to stress that:

- drand is in BETA, it might not be stable, and the API might change.
- the League of Entropy has no SLAs!

Future Work

drand needs more love in order to get past the prototype level:

- Replace BN256 with the BLS12-381 curve, since BN256 security level has been impacted by cryptanalysis breakthroughs in the EFDLP.
- Further improve the robustness and scalability of the drand setup process.
- Build applications relying on drand (I'm counting on you!)
- Improve unit testing
- Add support for multiple drand networks within one node
- Post quantum version?

“Random” quotes

The generation of random numbers is too important to be left to chance.

Robert R. Coveyou

Random numbers should not be generated with a method chosen at random.

Donald Knuth

Questions?