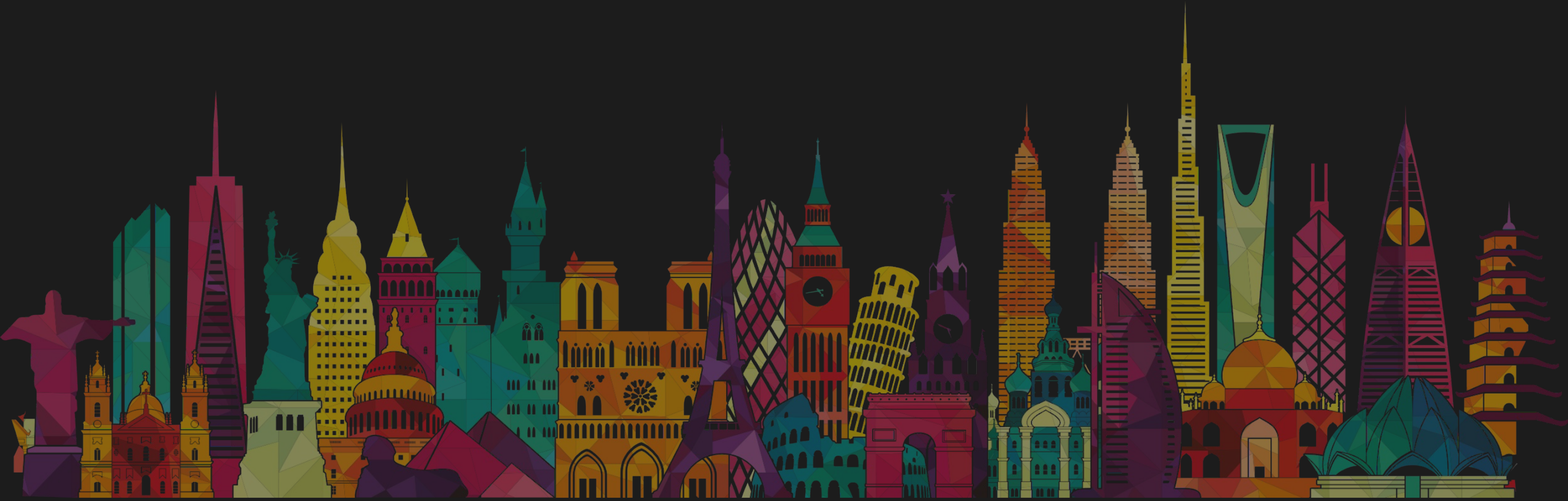
Zero knowledge proofs of shuffle with ShuffleProofs.jl



Dr. Janis Erdmanis (GitHub: JanisErdmanis, Twitter: graphitewriter, janiserdmanis.org)

Introduction

Usual cryptography => we can trust the other end to do the right thing

Remote electronic voting => How to combine privacy and accountability? ("evoting problem")

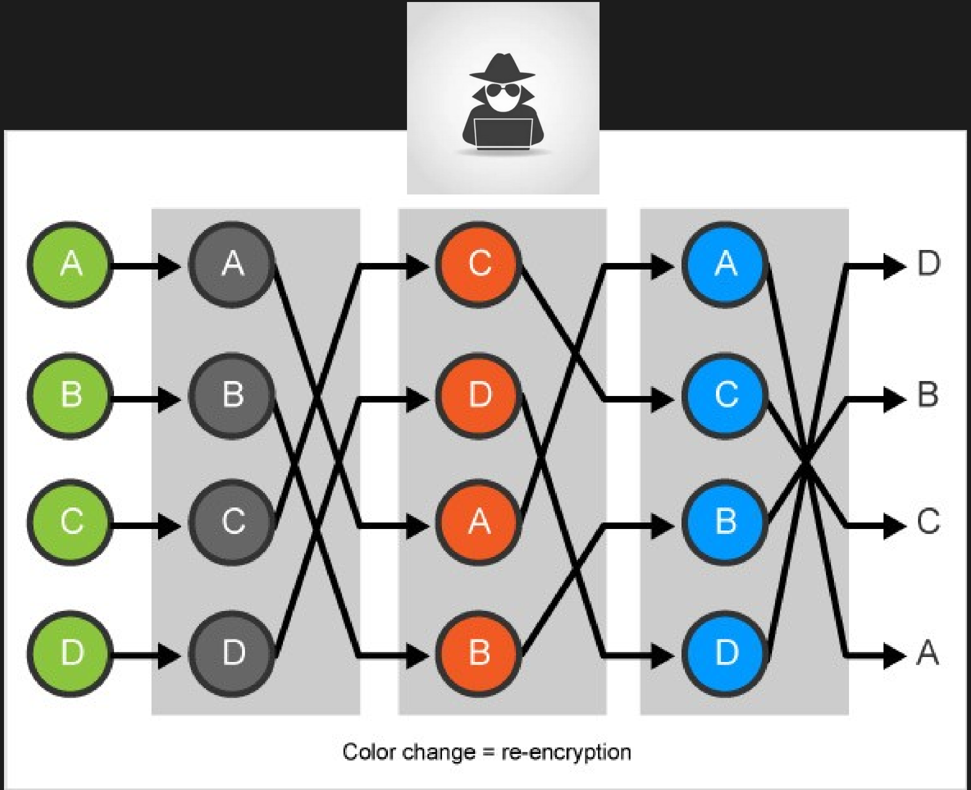
Introduction

E-voting: ironically it is hard because of efficiency

Privacy

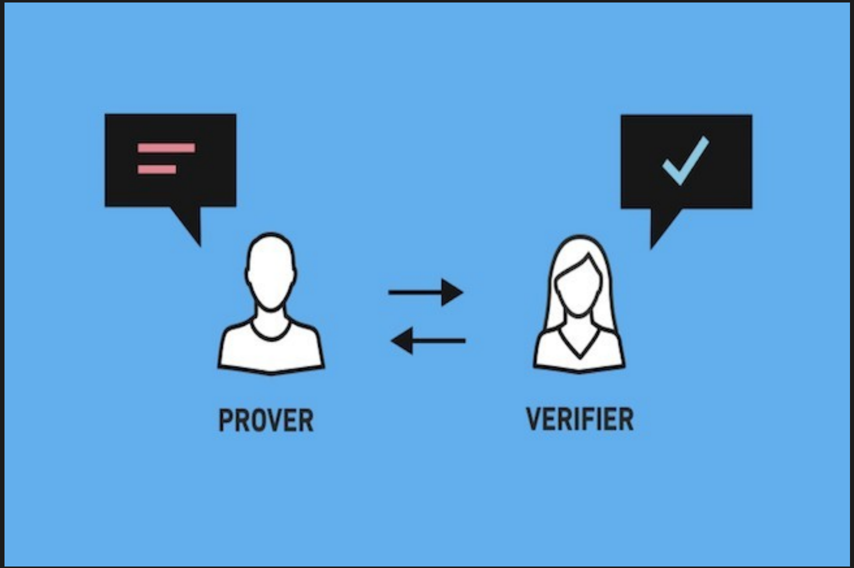
Accountability

Introduction



Introduction

Verificatum



* in noninteractive setting verifier is a “hash function”
to which prover feeds responses and receives challenges

Problem

Java is a verbose language and did not appeal for me.
Porting to Julia seemed like a good idea:

Leanness

Modularity

Worry free compatibility

However, a major downside of Julia is the lack of libraries in public key cryptography

CryptoGroups

Example

```julia
using CryptoGroups
G = PGroup(23, 11) # group type
g = G(3) # group element

for i in 1:11
    println("g^$i = $(g^i)")
end

# Outputs:
## g^1 = 3
## g^2 = 9
## g^3 = 4
## g^4 = 12
## g^5 = 13
## g^6 = 16
## g^7 = 2
## g^8 = 6
## g^9 = 18
## g^10 = 8
# And at g^11 throws an error for safety
```

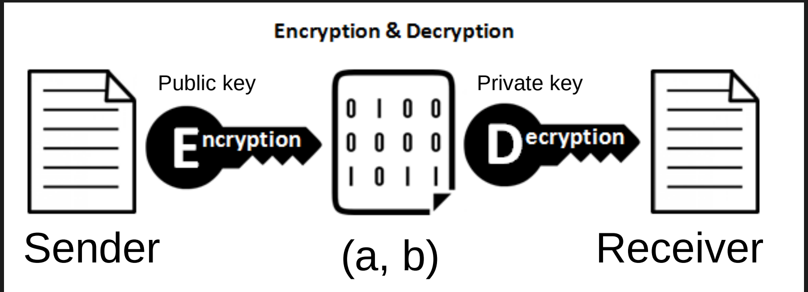
Ability for type parameters to hold values was essential in making a lean API.

Implements:

- *Elliptic curves over prime and binary fields*
- *Modular prime groups*
- *Relevant utility functions (point compression and basis selection)*
- *Common cryptographic constants*

Tailored for implementation of cryptographic schemes and protocols

ElGamal basics



Receiver generates a secret random number sk (his private key) and publishes a public key:

$$pk \leftarrow g^{sk}$$

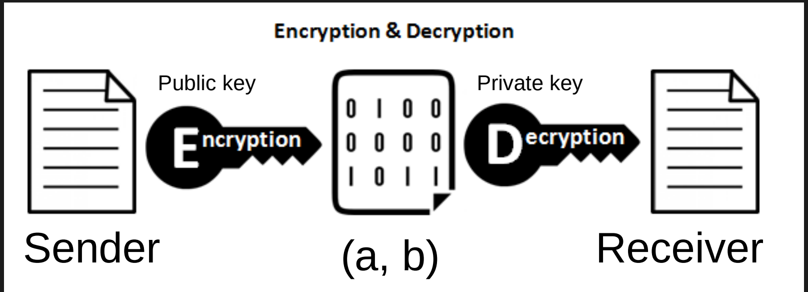
Sender: chooses a message m, and encrypts it with a randomization factor r:

$$(a, b) \leftarrow (g^r, m * pk^r)$$

Receiver: uses his private key sk to decrypt tuple (a, b):

$$b/a^{sk} = m$$

ElGamal reencryption



$$pk \leftarrow g^{sk}$$

$$(a, b) \leftarrow (g^r, m * pk^r)$$

Middleman: chooses a randomization factor r’ and reencrypts tuple (a, b):

$$(a', b') \leftarrow (a * g^{r'}, b * pk^{r'})$$

Receiver: uses his private key sk to decrypt tuple (a, b):

$$b' / a'^{sk} = m$$

Only receiver can tell that (a, b) and (a’, b’) holds the same message!

Typical reencryption mixnet voting system

Setup: m_A and m_B votes for candidates A and B. Public key pk.

Voters

Mix

Decryptor

Counting

$A: \ m = 7$
$B: \ m = 5$

$$V_1 : \ (g^{r_1}, m_A * pk^{r_1})$$

$$V_2 : \ (g^{r_2}, m_A * pk^{r_2})$$

$$V_3 : \ (g^{r_3}, m_B * pk^{r_3})$$

Input

	(a, b)
V_1	(3, 7)
V_2	(9, 3)
V_3	(8, 13)

Output

(a’, b’)
(13, 11)
(14, 13)
(7, 3)

m
7
5
7

Why should we trust election result when Mix and Decryptor can do whatever they want?

ShuffleProofs

```
using CryptoGroups
using ShuffleProofs

G = PGroup(23, 11)
g = G(3)

sk = 7
pk = g^sk

m_A = G(5)
m_B = G(7)

enc = Enc(pk, g)

ciphertexts = [enc(m_A, 2), enc(m_A, 3), enc(m_B,
5)]
proposition, secret = shuffle(ciphertexts, enc)

# The secret randomization factors can be used
# to test honesty of the mix with verify method:
verify(proposition, secret) == true
```

The proposition type

```
struct ElGamal{G <: Group} <: AbstractVector{G}
    a::Vector{G}
    b::Vector{G}
end

struct Shuffle{G <: Group} <: Proposition
   g::G
   pk::G
   e::ElGamal{G}
   e'::ElGamal{G}
end
```

Proposition holds all inputs and outputs of the mix

Honesty of the mix can be verified with secret randomization factors, but that violates privacy

ShuffleProofs: NIZK PoS

Proof of shuffle

```
verifier = ProtocolSpec(; g)
proof = prove(proposition, secret, verifier)

verify(proposition, proof, verifier) == true
```

See docs to choose verifier parameters with care

Proof can be published for on a bulletin board without any compromises on privacy

Appreciate and despair:

Type of proof

```
struct VShuffleProof{G<:Group} <: Proof
    μ::Vector{G}
    τ::Tuple{Vector{G}, G, Vector{G}, G, G, Tuple{G, G}}
    σ::Tuple{BigInt, Vector{BigInt}, BigInt, BigInt, Vector{BigInt}, BigInt}
end
```

For convenience a verifier can be passed directly to shuffle returning simulator object which contains proposition, proof and verifier

Shuffling with verifier

```
verifier = ProtocolSpec(; g)
simulator = shuffle(ciphertexts, enc, verifier)

verify(simulator) == true
```

Current state and the future

The current state:

- Much leaner API and more accessible for tinkering
- Easy get it running for any machine
- Poor performance and security in comparison with Verificatum, lacks features.

Verificatum generated PoS can be verified with ShuffleProofs:

Verification of Verificatum generated proof of shuffle

```
using ShuffleProofs
simulator = load_verificatum_simulator(DEMO_DIR)
verify(simulator)
```

In the future:

- Implement Verificatum compatible proof of decryption
- Add a command line interface
- Improve performance and add docs for CryptoGroups
- Interface existing cryptographic libraries with CryptoGroups API

Go to peacefounder.org and check out PeaceFounder Github organization to learn more

The end