

UniCrypt – A Mathematical Crypto-Library for Java

Rolf Haenni

Geneva, January 27th, 2016

Inhalt

- ▶ Introduction
- ▶ Design Principles and Architecture
- ▶ Layer 1: Mathematics
- ▶ Layer 2: Cryptography
- ▶ Summary and Outlook

Introduction

Introductory Example: JCA

```
1 KeyGenerator keyGenerator =
    KeyGenerator.getInstance("AES");
2 keyGenerator.init(128);
3 SecretKey key = keyGenerator.generateKey();

5 Cipher cipher =
    Cipher.getInstance("AES/ECB/PKCS5Padding");
6 cipher.init(Cipher.ENCRYPT_MODE, key);

8 byte [] message = new Random().getBytes(new byte [20]);
9 byte [] ciphertext = cipher.doFinal(message);

11 cipher.init(Cipher.DECRYPT_MODE, key);
12 byte [] cleartext = cipher.doFinal(ciphertext);
```

Introductory Example: UniCrypt

```
1  AESEncryptionScheme aes =  
    AESEncryptionScheme.getInstance();  
  
3  Element key = aes.getKeyGenerator().generateKey();  
  
5  Element message =  
    aes.getMessageSpace().getRandomElement(20);  
  
7  Element ciphertext = aes.encrypt(key, message);  
  
9  Element cleartext = aes.decrypt(key, ciphertext);
```

Motivation

- ▶ Multiple e-voting projects since 2008
- ▶ Various protocols implemented [CGS97, JCJ05, HS11, ...]
- ▶ Cryptographic primitives re-implemented
 - ▶ Secret-sharing
 - ▶ Pedersen commitments
 - ▶ ElGamal encryption and re-encryption
 - ▶ Zero-knowledge proofs
 - ▶ Cryptographic mixing
 - ▶ Elliptic curves
 - ...
- ▶ No suitable free library available off the shelf

Project Milestones

- ▶ February'12: UniVote project launch (student board elections)
- ▶ July'12: First unofficial UniCrypt release (student project)
- ▶ February'13: Second unofficial release (part of UniVote)
- ▶ March–June'13: Multiple elections in Berne, Zürich, Lucerne
- ▶ August'13: Independent project on GitHub
- ▶ September'13 – February'14: Complete re-design
- ▶ December'13: Proof of shuffle implemented (Wikström)
- ▶ February'14: Beta-version used in MobiVote
- ▶ August'15: First official release
- ▶ October'15: Elections in Lucerne and Basel

Project Team

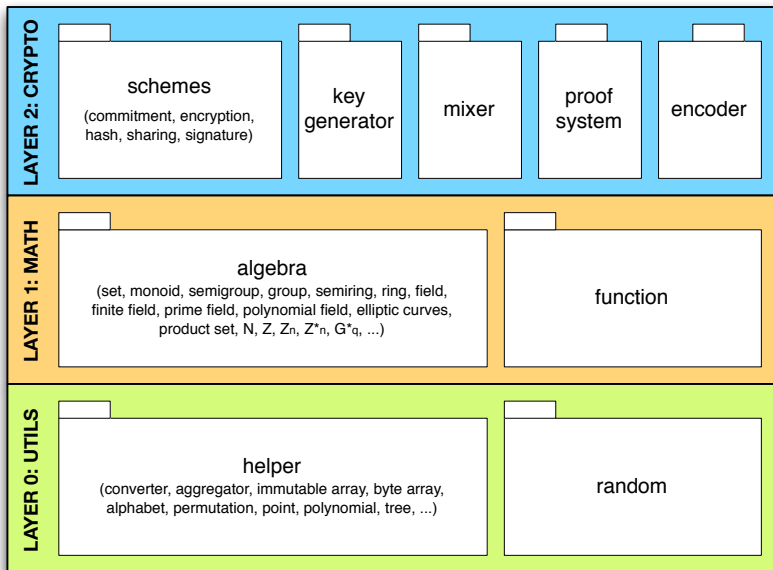
- ▶ Prof. Rolf Haenni: Project coordinator, design, development
- ▶ Prof. Reto E. Koenig: Design, random generators
- ▶ Prof. Philipp Locher: Cryptographic mixing, zero-knowledge proofs
- ▶ MSc Philémon von Bergen: Zero-knowledge proofs
- ▶ MSc Christian Lutz: Elliptic curves
- ▶ MSc Jürg Ritter: First unofficial release
- ▶ Gina-Maria Musaelyan: Documentation

Design Principles and Architecture

Design Principles

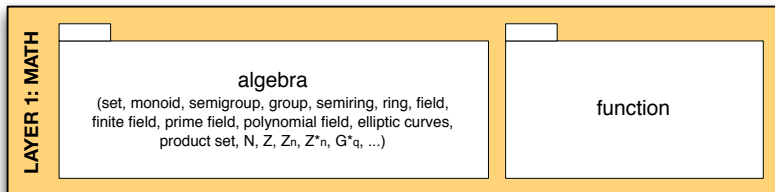
- ▶ Full coherence with mathematical and cryptographic concepts
- ▶ Consistent and self-explanatory nomenclature
- ▶ Clean and intuitive APIs
- ▶ Generic types (hidden from the developer if possible)
- ▶ Consistent coding style
- ▶ Immutable objects only
- ▶ Design patterns (only) if useful
- ▶ No cryptographic black-boxes (e.g. random generator)
- ▶ Java 7 compatibility

Architecture

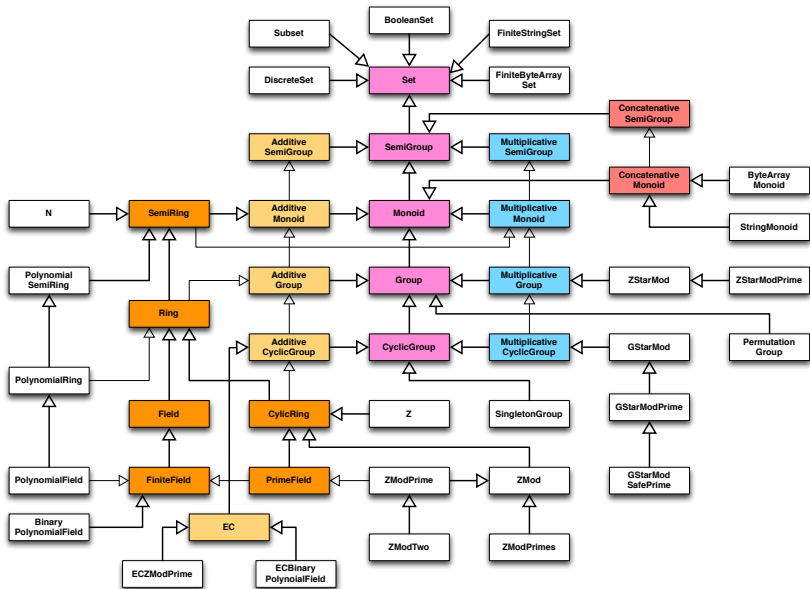


Layer 1: Mathematics

Architecture: Layer 1



Algebra Package



Basic Algebraic Structures

- ▶ No operation

- ▶ Set: Membership test $e \overset{?}{\in} S$, order $q = |S|$
- ▶ Element: Container for elements $e \in S$

- ▶ Single operation \circ

- ▶ SemiGroup: Associative binary operation $e_1 \circ e_2$
- ▶ Monoid: Identity element $i \circ e = e \circ i = e$
- ▶ Group: Inverse element $Inv(e) \circ e = e \circ Inv(e) = i$
- ▶ CyclicGroup: Generator $\langle g \rangle = \underbrace{\{g \circ \dots \circ g : 1 \leq i \leq q\}}_{i \text{ times}} = S$

- ▶ Examples

- ▶ DiscreteSet: $S = \{s_1, \dots, s_q\}$
- ▶ FiniteStringSet: $S_n = \mathcal{A}^0 \cup \mathcal{A}^1 \cup \dots \cup \mathcal{A}^n$ for alphabet \mathcal{A}
- ▶ PermutationGroup: $\Pi_n = \{\pi : \text{permutation of order } n\}$

Example: PermutationGroup

```
1 // Generate permutation group of size 5
2 PermutationGroup group =
    PermutationGroup.getInstance(5);

4 // Compute order |group| = 5! = 120
5 BigInteger order = group.getOrder();

7 // Pick random permutation and invert it
8 Element p1 = group.getRandomElement();
9 Element p2 = group.invert(p1);

11 // Combine p1 and p2 into p12 = (0,1,2,3,4)
12 Element p12 = group.apply(p1, p2);
```


Multiplicative Algebraic Structures

- ▶ Set with $*$ as operation

- ▶ MultiplicativeSemiGroup: $e_1 * e_2$
- ▶ MultiplicativeMonoid: $1 * e = e * 1 = e$
- ▶ MultiplicativeGroup: $e^{-1} * e = e * e^{-1} = 1$
- ▶ MultiplicativeCyclicGroup: $\langle g \rangle = \{g^i : 1 \leq i \leq q\} = S$

- ▶ Examples

- ▶ ZStarMod: $\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n : \gcd(x, n) = 1\}$
- ▶ ZStarModPrime: $\mathbb{Z}_p^* = \{1, \dots, p-1\}$
- ▶ GStarMod: $\mathbb{G}_q \subset \mathbb{Z}_n^*$ (cyclic subgroup of prime order q)
- ▶ GStarModPrime: $\mathbb{G}_q \subset \mathbb{Z}_p^*$
- ▶ GStarModSafePrime: $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for $p = 2q + 1$

Example: GStarModSafePrime

```
1 // Generate cyclic subgroup G_11 for p = 23
2 GStarMod g11 = GStarModSafePrime.getInstance(23);

4 // Compute order (23-1)/2 = 11
5 BigInteger order = g11.getOrder();

7 // Multiply two group elements: 3*9 mod 23 = 4
8 Element e1 = g11.getElement(3);
9 Element e2 = g11.getElement(9);
10 Element e12 = g11.multiply(e1, e2);

12 // Select default generator g and compute g^5
13 Element generator = g11.getDefaultGenerator();
14 Element result = g11.power(generator, 5);
```

Additive Algebraic Structures

- ▶ Set with $+$ as operation

- ▶ AdditiveSemiGroup: $e_1 + e_2$

- ▶ AdditiveMonoid: $0 + e = e + 0 = e$

- ▶ AdditiveGroup: $-e + e = e + (-e) = 0$

- ▶ AdditiveCyclicGroup: $\langle g \rangle = \{i \cdot g : 1 \leq i \leq q\} = S$

- ▶ Examples

- ▶ ECZModPrime: $E(\mathbb{Z}_p) = \{(x, y) : x^2 = y^3 + ax + b\} \cup \{\infty\}$

- ▶ ECBinaryPolynomialField: $E(\mathbb{Z}_{2^p})$

Concatenative Algebraic Structures

- ▶ Concatenation as operation

- ▶ ConcatenativeSemiGroup: $e_1 \parallel e_2$

- ▶ ConcatenativeMonoid: $\langle \rangle \parallel e = e \parallel \langle \rangle = e$

- ▶ Examples

- ▶ StringMonoid: $S = \mathcal{A}^0 \cup \mathcal{A}^1 \cup \dots = \mathcal{A}^*$

- ▶ ByteArrayMonoid: $B = (\{0, 1\}^8)^*$

Example: StringMonoid

```
1 // Define alphabet and generate string monoid
2 Alphabet alphabet = Alphabet.getInstance("ADEHLORW-");
3 StringMonoid monoid =
4     StringMonoid.getInstance(alphabet);
5
6 // Generate "HELLO-WORLD" from 3 strings
7 Element s1 = monoid.getElement("HELLO");
8 Element s2 = monoid.getElement("-");
9 Element s3 = monoid.getElement("WORLD");
10 Element s123 = monoid.concatenate(s1, s2, s3);
11
12 // Generate random string of length 10
13 Element s = monoid.getRandomElement(10);
```

Structures with Two Operations

- ▶ Set with two operations $+$ and $*$
 - ▶ SemiRing: $*$ distributes over $+$ with identity elements 1 and 0
 - ▶ Ring: Additive inverse $-e + e = 0$
 - ▶ Field: Multiplicative inverse $e^{-1} * e = 1$
 - ▶ FiniteField: $|S| < \infty$
 - ▶ PrimeField: $|S| = \textit{prime}$

- ▶ Examples
 - ▶ N: Semiring of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$
 - ▶ Z: Ring of integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \dots\}$
 - ▶ ZMod: Ring \mathbb{Z}_n of integers modulo n
 - ▶ ZModPrime: Prime field \mathbb{Z}_p
 - ▶ PolynomialRing: Polynomial ring $R[x]$ over a ring R

Example: Ring of Integers (mod n)

```
1 // Generate Z_25 and compute its order
2 ZMod z25 = ZMod.getInstance(25);
3 BigInteger order = z25.getOrder();

5 // Define 4 element 3, 7, 10, 12
6 ZModElement e1 = z25.getElement(3);
7 ZModElement e2 = z25.getElement(7);
8 ZModElement e3 = z25.getElement(10);
9 ZModElement e4 = z25.getElement(12);

11 // Compute (e1 + e2^2 - e3)/e4 mod 25
12 Element result =
    e1.add(e2.square()).subtract(e3).divide(e4);
```

Cartesian Products and Tuples

- ▶ Cartesian products $S_1 \times \cdots \times S_n$
 - ▶ ProductSet
 - ▶ ProductSemiGroup
 - ▶ ProductMonoid
 - ▶ ProductGroup
 - ▶ ProductCyclicGroup
- ▶ Corresponding combined elements
 - ▶ Tuple: general tuples $(e_1, \dots, e_n) \in S_1 \times \cdots \times S_n$
 - ▶ Pair: tuples $(e_1, e_2) \in S_1 \times S_2$ of arity 2
 - ▶ Triple: tuples $(e_1, e_2, e_3) \in S_1 \times S_2 \times S_3$ of arity 3
- ▶ Sets and elements can be composed recursively

Example: Products and Tuples

```
1 // Generate 3 atomic sets
2 Set s1 = ZMod.getInstance(25);
3 Set s2 = PermutationGroup.getInstance(5);
4 Set s3 = StringMonoid.getInstance(Alphabet.LOWER_CASE);

6 // Generate  $s1 \times s2$ ,  $(s1 \times s2)^2$ , and  $(s1 \times s2)^2 \times s3$ 
7 ProductSet s4 = ProductSet.getInstance(s1, s2);
8 ProductSet s5 = ProductSet.getInstance(s4, 2);
9 ProductSet s6 = s5.add(s3);

11 // Select random tuple  $t1 = ((e1, e2), (e3, e4)), e5$ 
12 Tuple t1 = s6.getRandomElement();

14 // Generate tuple  $t2 = (5, "hello")$ 
15 Tuple t2 = Tuple.getInstance(s1.getElement(5),
    s3.getElement("hello"));
```

Functions

- ▶ **Function:** mathematical concept of a function $f : X \rightarrow Y$
 - ▶ `public Set getDomain();`
 - ▶ `public Set getCoDomain();`
 - ▶ `public Element apply(Element e);`
- ▶ There is a large set of predefined functions
 - ▶ `AdapterFunction`, `AdditionFunction`, `ConstantFunction`, `ConvertFunction`, `EqualityFunction`, `HashFunction`, `IdentityFunction`, `InvertFunction`, `ModuloFunction`, `MultiplicationFunction`, `PermutationFunction`, `PowerFunction`, `SelectionFunction`, ...
- ▶ Functions can be combined in two ways
 - ▶ `ComposedFunction`: $f(x) = f_1 \circ \dots \circ f_n(x) = f_1(f_2(\dots f_n(x)))$
 - ▶ `ProductFunction`: $f(x_1, \dots, x_n) = (f_1(x_1), \dots, f_n(x_n))$

Example: HashFunction

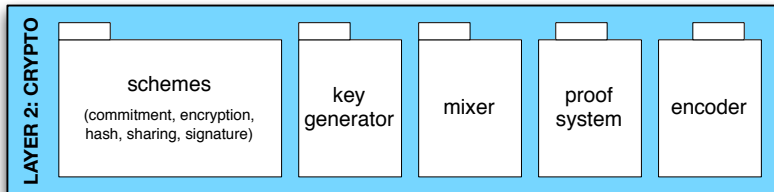
```
1 // Generate product group  $Z_{23}^{10}$  and random element
2 Group z23 = ZModPrime.getInstance(23);
3 ProductGroup pg = ProductGroup.getInstance(z23, 10);
4 Tuple tuple = pg.getRandomElement();

6 // Define hash function  $Z_{23}^{10} \rightarrow \{0,1\}^{256}$  (SHA256)
7 Function function = HashFunction.getInstance(pg);

9 // Apply function to tuple (returns ByteArrayElement)
10 Element hashValue = function.apply(tuple);
```

Layer 2: Cryptography

Architecture: Layer 2



Shamir Secret Sharing

- ▶ Prime field \mathbb{Z}_p
- ▶ Secret $s \in \mathbb{Z}_p$ to share among n people
- ▶ Threshold $t \leq n$
- ▶ Polynomial $f(x) = s + a_1x + a_2x^2 + \dots + a_{t-1}x^{t-1}$ for $a_i \in \mathbb{Z}_p$
- ▶ Share $s_i = (x_i, f(x_i))$, for $x_i \in \mathbb{Z}_p$ and $i = 1, \dots, n$
- ▶ Recovering of s using Lagrange interpolation

ShamirSecretSharingScheme

```
1 // Define prime field
2 ZModPrime z29 = ZModPrime.getInstance(29);

4 // Generate (5,3)-threshold secret sharing scheme
5 SecretSharingScheme sss =
    ShamirSecretSharingScheme.getInstance(z29, 5, 3);

7 // Create message m=25
8 Element message = sss.getMessageSpace().getElement(5);

10 // Compute shares
11 Tuple shares = sss.share(message);

13 // Select subset of shares and recover message
14 Tuple someShares = shares.removeAt(1).removeAt(3);
15 Element recoveredMessage = sss.recover(someShares);
```

ElGamal Encryption

- ▶ Cyclic prime order subgroup $G_q \subset \mathbb{Z}_p^*$ for $p = 2q + 1$
- ▶ Generator $g \in G_q$
- ▶ Private key $x \in \mathbb{Z}_q$
- ▶ Public key $y = g^x \in G_q$
- ▶ Randomization $r \in \mathbb{Z}_q$
- ▶ Message $m \in G_q$
- ▶ Encryption: $Enc_y(m, r) = (g^r, m \cdot y^r) \in G_q \times G_q$
- ▶ Decryption: $Dec_x(a, b) = b/a^x$

ElGamalEncryptionScheme I

```
1 // Create cyclic group and get default generator
2 CyclicGroup g_q = GStarModSafePrime.getInstance(23);
3 Element generator = g_q.getDefaultGenerator();

5 // Create ElGamal encryption scheme
6 ElGamalEncryptionScheme elGamal =
    ElGamalEncryptionScheme.getInstance(generator);

8 // Create keys
9 KeyPairGenerator kpg = elGamal.getKeyPairGenerator();
10 Element privateKey = kpg.generatePrivateKey();
11 Element publicKey = kpg.generatePublicKey(privateKey);

13 // Create random message
14 Element message =
    elGamal.getMessageSpace().getRandomElement();
```

ElGamalEncryptionScheme II

```
17 // Perform encryption
18 Element ciphertext = elGamal.encrypt(publicKey ,
    message);

20 // Perform decryption
21 Element cleartext = elGamal.decrypt(privateKey ,
    ciphertext);

23 // Get encryption and decryption function
24 Function encFunction elGamal.getEncryptionFunction();
25 Function decFunction elGamal.getDecryptionFunction();
```

Cryptographic Mixing

- ▶ Let E_1, \dots, E_n be a list of ElGamal encryptions of messages m_i with randomisations r_i and public key y
- ▶ Re-encryptions $E'_i = E_i * \text{Encrypt}_y(1, r'_i) = \text{Encrypt}(m, r_i + r'_i)$
- ▶ Permutation $\pi \in \Pi_n$
- ▶ Cryptographic mixing:

$$\text{shuffle}_\pi(E_1, \dots, E_n, r'_1, \dots, r'_n) = (E'_{\pi(1)}, \dots, E'_{\pi(n)})$$

ReEncryptionMixer

```
1 // Generate encryptions
2 Element e1 = elGamal.encrypt(publicKey, m1, r1);
3 Element e2 = elGamal.encrypt(publicKey, m2, r2);
4     :           :           :
5 Element e10 = elGamal.encrypt(publicKey, m10, r10);
6 Tuple encryptions = Tuple.getInstance(e1, e2, ... e10);

8 // Create mixer
9 Mixer mixer = ReEncryptionMixer.getInstance(elGamal,
      publicKey, 10);

11 // Shuffle the encryptions
12 Tuple shuffledEncryptions = mixer.shuffle(encryptions);
```

Non-Interactive Preimage Proofs

- ▶ Let $f : X \rightarrow Y$ be a homomorphic function (one-way)
- ▶ Private input $x \in X$
- ▶ Public input $y = f(x) \in Y$
- ▶ Prove knowledge of preimage x by computing

$$(t, c, s) = \text{NIZKP}\{x \in X : y = f(x)\}$$

non-interactively from x and y (without releasing any information about x)

- ▶ Example: proof knowledge of private ElGamal key $y = g^x$

PreimageProofGenerator

```
1 EncryptionScheme elGamal = ElGamalEncryptionScheme... ;  
  
3 // Generate keys  
4 KeyPairGenerator kpg = elGamal.getKeyPairGenerator();  
5 Element privateKey = kpg.generatePrivateKey();  
6 Element publicKey = kpg.generatePublicKey(privateKey);  
  
8 // Create proof generator  
9 Function function =  
10     kpg.getPublicKeyGenerationFunction();  
11 PreimageProofGenerator ppg =  
12     PreimageProofGenerator.getInstance(function);  
  
13 // Generate and verify proof  
14 Triple proof = ppg.generate(privateKey, publicKey);  
15 BooleanElement result = ppg.verify(proof, publicKey);
```

Summary and Outlook

Summary

- ▶ UniCrypt = Java library with advanced mathematical and cryptographic primitives
- ▶ Offers clean and intuitive APIs
- ▶ Growing in size
 - ▶ 68 interfaces
 - ▶ 217 classes
 - ▶ 34553 lines of codes (incl. comments, excl. tests)
- ▶ Open-source: available on GitHub
- ▶ Free for academic or non-commercial usage (dual license)
- ▶ Collaborations are welcome

Outlook

- ▶ Important missing components
 - ▶ Further encryption schemes (Paillier)
 - ▶ Further signature schemes (DSA, RSA)
 - ▶ Sponge functions (SHA3)
 - ▶ Further types of zero-knowledge proofs
 - ▶ Other cryptographic schemes (certificates, message authentication, authenticated encryption)
- ▶ Incomplete documentation
- ▶ Insufficient JUnit code coverage
- ▶ JavaScript counterpart



Source: <http://www.djibang.org/wp/wp-content/uploads/eisberg.jpg>

Questions?

<http://e-voting.bfh.ch>

<https://github.com/bfh-evg/unicrypt>

<https://github.com/bfh-evg/unicrypt-samples>