UniCrypt – A Mathematical Crypto-Library for Java

Rolf Haenni
TU Darmstadt, CDC, Oberseminar, 6.2.2014
Inhalt

- Introduction
- Design Principles and Architecture
- Layer 1: Mathematics
- Layer 2: Cryptography
- Summary and Outlook
Introduction
Introductory Example: JCA

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

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

byte[] message = new Random().getBytes(new byte[20]);
byte[] encrypted = cipher.doFinal(message);

cipher.init(Cipher.DECRYPT_MODE, key);
byte[] decrypted = cipher.doFinal(encrypted);
```
Introductory Example: UniCrypt

```java
AESEncryptionScheme aes = AESEncryptionScheme.getInstance();

Element key = aes.getKeyGenerator().generateKey();

Element message =
    aes.getMessageSpace().getRandomElement(20);

Element encryption = aes.encrypt(key, message);

Element decryption = aes.decrypt(key, encryption);
```
Motivation

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

► February’12: UniVote project launched (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
Project Team

- **Prof. Rolf Haenni**: Project coordinator, design, development
- **Prof. Reto E. Koenig**: design, random generators
- **MSc Philipp Locher**: Cryptographic mixing, zero-knowledge proofs
- **BSc Jürg Ritter**: First unofficial release
- **BSc Philémon von Bergen**: Zero-knowledge proofs
- **BSc Christian Lutz**: Elliptic curves
- **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 6 compatibility
Architecture

**Layer 1: Math**
- Algebra
- Function

**Layer 2: Crypto**
- Random
- Encoder
- Mixer
- Proof Generator
- Schemes (commitment, encryption, hash, sharing, signature)

**Helper**
- Utility
Layer 1: Mathematics
Architecture: Layer 1

Layer 1: MATH

- algebra
- function
- helper
- utility
Basic Algebraic Structures

- **No operation**
  - Set: Membership test \( e \in S \), 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 \( Inv(e) \circ e = i \)
  - CyclicGroup: Generator \( \langle g \rangle = \{g \circ \cdots \circ g : 1 \leq i \leq q \} = S \)

- **Examples**
  - DiscreteSet: \( S = \{s_1, \ldots, s_q\} \)
  - FiniteStringSet: \( S_n = A^0 \cup A^1 \cup \cdots \cup A^n \) for alphabet \( A \)
  - PermutationGroup: \( \Pi_n = \{\pi : \text{permutation of order } n\} \)
// Generate permutation group of size 5
PermutationGroup group =
    PermutationGroup.getInstance(5);

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

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

// Combine p1 and p2 into p12 = (0,1,2,3,4)
Element p12 = group.apply(p1, p2);
Multiplicative Algebraic Structures

Set with $\ast$ as operation

- MultiplicativeSemiGroup: $e_1 \ast e_2$
- MultiplicativeMonoid: $1 \ast e = e \ast 1 = e$
- MultiplicativeGroup: $e^{-1} \ast e = 1$
- 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: $\mathbb{Z}_p^* = \{ 1, \ldots, p - 1 \}$
- 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

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

// Compute order (23−1)/2 = 11
BigInteger order = g11.getOrder();

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

// Select default generator g and compute g^5
Element generator = g11.getDefaultGenerator();
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: \(-e + e = 0\)
  - 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: \(E(\mathbb{Z}_{2p})\)
Concatenative Algebraic Structures

- Concatenation as operation
  - ConcatenativeSemiGroup: $e_1 \| e_2$
  - ConcatenativeMonoid: $\langle \rangle \| e = e \| \langle \rangle = e$

- Examples
  - StringMonoid: $S = A^0 \cup A^1 \cup \cdots = A^*$
  - ByteArrayMonoid: $B = (\{0, 1\}^8)^*$
Example: StringMonoid

```java
// Define alphabet and generate string monoid
Alphabet alphabet = Alphabet.getInstance("ADEHLORW-"),
StringMonoid monoid =
    StringMonoid.getInstance(alphabet);

// Generate "HELLO-WORLD" from 3 strings
Element s1 = monoid.getElement("HELLO");
Element s2 = monoid.getElement("-");
Element s3 = monoid.getElement("WORLD");
Element s123 = monoid.concatenate(s1, s2, s3);

// Generate random string of length 10
Element s = monoid.getRandomElement(10);
```
Structures with Two Operations

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

- **Examples**
  - **$\mathbb{N}$**: Semiring of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$
  - **$\mathbb{Z}$**: Ring of integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$
  - **$\mathbb{Z}_{\text{Mod}}$**: Ring $\mathbb{Z}_n$ of integers modulo $n$
  - **$\mathbb{Z}_{\text{ModPrime}}$**: Prime field $\mathbb{Z}_p$
  - **$\text{PolynomialRing}$**: Polynomial ring $R[x]$ over a ring $R$
Example: Ring of Integers (mod n)

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

// Define 4 element 3, 7, 10, 12
DualisticElement e1 = z25.getElement(3);
DualisticElement e2 = z25.getElement(7);
DualisticElement e3 = z25.getElement(10);
DualisticElement e4 = z25.getElement(12);

// Compute (e1 + e2^2 - e3)/e4 mod 25
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 $(e_1, \ldots, e_n) \in S_1 \times \cdots \times S_n$
  - 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

```
// Generate 3 atomic sets
Set s1 = Z.getInstance();
Set s2 = N.getInstance();
Set s3 = StringMonoid.getInstance(Alphabet.LOWER_CASE);

// Generate s1 x s2, (s1 x s2)^3 and (s1 x s2)^3 x s3
ProductSet s4 = ProductSet.getInstance(s1, s2);
ProductSet s5 = ProductSet.getInstance(s4, 3);
ProductSet s6 = s5.add(s3);

// Select random tuple t1 = (e1,e2) from s4
Tuple t1 = s4.getRandomElement();

// Generate tuple t2 = (-5,"hello") from s1 x s3
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
- Functions can be combined in two ways
  - ComposedFunction: \( f(x) = f_1 \circ \cdots \circ f_n(x) = f_1(f_2(\cdots f_n(x))) \)
  - ProductFunction: \( f(x_1, \ldots, x_n) = (f_1(x_1), \ldots, f_n(x_n)) \)
Example: HashFunction

```java
// Generate product group $\mathbb{Z}_{23}^{\times 10}$ and random element
Group z23 = ZModPrime.getInstance(23);
ProductGroup pg = ProductGroup.getInstance(z23, 10);
Tuple tuple = pg.getRandomElement();

// Define hash function $\mathbb{Z}_{23}^{\times 10} \to \{0,1\}^{256}$ (SHA256)
Function function = HashFunction.getInstance(pg);

// Apply hash function to tuple (returns byte array)
Element hashValue = function.apply(tuple);
```
Layer 2: Cryptography
Architecture: Layer 2

LAYER 2: CRYPTO

- random
- encoder
- mixer
- proof generator
- schemes (commitment, encryption, hash, sharing, signature)
Shamir Secret Sharing

- Prime field $\mathbb{Z}_p$
- Secret $s \in \mathbb{Z}_p$ to share among $n$ people
- Threshold $t \leq n$
- Polynomial $f(x) = s + a_1 x + a_2 x^2 + \ldots + a_{t-1} x^{t-1}$ for $a_i \in \mathbb{Z}_p$
- Share $s_i = (x_i, f(x_i))$, for $x_i \in \mathbb{Z}_p$ and $i = 1, \ldots, n$
- Recovering of $s$ using Lagrange interpolation
// Define prime field
ZModPrime z29 = ZModPrime.getInstance(29);

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

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

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

// Select subset of shares and recover message
Tuple someShares = shares.removeAt(1).removeAt(3);
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: $\text{Enc}_y(m, r) = (g^r, m \cdot y^r) \in G_q \times G_q$
- Decryption: $\text{Dec}_x(a, b) = b/a^x$
ElGamalEncryptionScheme I

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

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

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

// Create random message
Element message =
    elGamal.getMessageSpace().getRandomElement();
```
// Perform encryption
Element encryption = elGamal.encrypt(publicKey, message);

// Perform decryption
Element decryption = elGamal.decrypt(privateKey, encryption);

// Get encryption and decryption function
Function encFunction = elGamal.getEncryptionFunction();
Function decFunction = elGamal.getDecryptionFunction();
Encoders

- An encoder represents an injective mapping between two sets
  - public Element encode(Element e)
  - public Element decode(Element e)

- Example: Encrypt string (base-64) with ElGamal
  - recall that \( m \in G_q \subset \mathbb{Z}_p^* \)
  - for example \( G_{11} = \{1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18\} \subset \mathbb{Z}_{23}^* \)
  - we need \( f : S_n \rightarrow G_q \) and \( f^{-1} : G_q \rightarrow S_n \), possibly constructed as a composed function \( f = f_1 \circ f_2 \) for
    - \( f_1 : S_n \rightarrow \mathbb{Z}_q \) (FiniteStringToZModEncoder)
    - \( f_2 : \mathbb{Z}_q \rightarrow G_q \) (ZModToGStarModSafePrimeEncoder)
```java
// Define underlying groups
Group g_q = GStarModSafePrime.getRandomInstance(256);
Group z_q = g_q.getZModOrder();

// Create alphabet and encoders
Alphabet base64 = Alphabet.BASE64;
Encoder encoder1 =
    FiniteStringToZModEncoder.getInstance(z_q, base64);
Encoder encoder2 =
    ZModToGStarModSafePrimeEncoder.getInstance(g_q);
Encoder encoder12 =
    CompositeEncoder.getInstance(encoder1, encoder2);

// Encode and decode message
Element message =
    encoder12DOMAIN().getElement("Hello World");
Element encoded = encoder12.encode(message);
Element decoded = encoder12.decode(encoded);
```
Cryptographic Mixing

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

$$\text{shuffle}_\pi(E_1, \ldots, E_n, r'_1, \ldots, r'_n) = (E'_{\pi(1)}, \ldots, E'_{\pi(n)})$$
// Generate encryptions
Element e1 = elGamal.encrypt(publicKey, m1, r1);
Element e2 = elGamal.encrypt(publicKey, m2, r2);
... : : :
Element e10 = elGamal.encrypt(publicKey, m10, r10);
Tuple encryptions = Tuple.getInstance(e1, e2, ... e10);

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

// Shuffle the encryptions
Tuple shuffledEncryptions = mixer.shuffle(encrypted);
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 $y = g^x$
EncryptionScheme elGamal = ElGamalEncryptionScheme...;

// Generate keys
KeyPairGenerator kpg = elGamal.getKeyPairGenerator();
Element privateKey = kpg.generatePrivateKey();
Element publicKey = kpg.generatePublicKey(privateKey);

// Create proof generator
Function function =
    kpg.getPublicKeyNameGenerationFunction();
PreimageProofGenerator ppg =
    PreimageProofGenerator.getInstance(function);

// Generate and verify proof
Triple proof = ppg.generate(privateKey, publicKey);
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

- Stage of development: alpha
- Important components under development
  - elliptic curves
  - true random generators
- Important components missing
  - signature schemes
  - certificates
  - further encryption schemes (RSA, Paillier, etc.)
  - further types of zero-knowledge proofs
  - other cryptographic schemes
- Documentation largely missing
- Insufficient code coverage by existing JUnit tests
Questions?

http://e-voting.bfh.ch

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