UniCrypt 2.0

Rolf Haenni

http://e-voting.bfh.ch

Seminar, E-Voting Group, BFH

April 24th, 2013

Outline

Hash Function with Multiple Arguments

Representing Elements by Natural Numbers

Simplified Element Interfaces

Proper RSA Implementation

Nomenclature and Factories

Extensions

Outline

Hash Function with Multiple Arguments

- Hash function $H : \{0,1\}^* \to \{0,1\}^n$
- Collision: $H(x) = H(y)$ for $x \neq y$
- Hash function with multiple arguments:

$$H : \{0,1\}^* \times \cdots \times \{0,1\}^* \to \{0,1\}^n$$

- Collision with multiple arguments:

$$H(x_1,\ldots,x_k) = H(y_1\ldots,y_k),\ \text{for } (x_1,\ldots,x_k) \neq (y_1,\ldots,y_k)$$

- Note that concatenation $|| : \{0,1\}^* \times \cdots \times \{0,1\}^* \to \{0,1\}^*$ leads to collisions, for example:

$$H(0,01) = H(0||01) = H(00||1) = H(00,1)$$

Current Solution: String Concatenation

- Alphabet $\mathcal{A}$ (for example $\mathcal{A} = \{0, 1, \ldots, 9\}$)
- Encoding: $\varepsilon : \{0,1\}^* \to \mathcal{A}^*$
- Separator: $s \notin \mathcal{A}$
- Decoding: $\delta : (\mathcal{A} \cup \{s\})^* \to \{0,1\}^*$
- Hash function with multiple arguments:

$$H(x_1, \ldots, x_k) = H(\delta(\varepsilon(x_1)||s|| \cdots ||s||\varepsilon(x_k)))$$

- If x_i itself is a tuple, it gets even more complicated

New Solution 1: Hash of Hash Values

- Hash function with multiple arguments:

$$H(x_1, \ldots, x_k) = H(H(x_1) || \cdots || H(x_k))$$

- Same collision probability as the hash function itself

New Solution 2: Pairing (Tuple) Function

- Encoding: $\varepsilon : \{0,1\}^* \to \mathbb{N}$
- Pairing (tuple) function: $\psi_k : \mathbb{N}^k \to \mathbb{N}$
- Decoding: $\delta : \mathbb{N} \to \{0,1\}^*$
- Hash function with multiple arguments:

$$H(x_1, \ldots, x_k) = H(\delta(\psi_k(\varepsilon(x_1), \ldots, \varepsilon(x_k))))$$

- If we choose ε and δ to be the standard binary representation, we can write

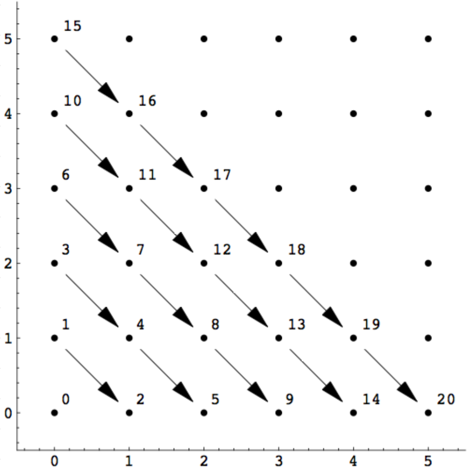
$$H(x_1, \ldots, x_k) = H(\psi_k(x_1, \ldots, x_k))$$

- Note that in UniCrypt, we already work with natural numbers most of the time

Cantor Pairing Function

- A *pairing function* maps two natural numbers bijectively into a single natural number
- The *Cantor pairing function* is defined as follows:
$$\psi(x_1, x_2) = \frac{1}{2}(x_1 + x_2)(x_1 + x_2 + 1) + x_2$$
- Note that $|\psi(x_1, x_2)| = 2 \cdot \max(|x_1|, |x_2|) + 1$
- The inverse function $\psi^{-1} : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ is called *unpairing function*
- For $y = \psi(x_1, x_2)$, let $s = \lfloor \frac{\sqrt{8y+1}-1}{2} \rfloor$ and $t = \frac{1}{2}(s^2 + s)$
- This implies:
$$(x_1, x_2) = \psi^{-1}(y) = (y - t, s + t - y)$$

Cantor Pairing Function



Elegant Pairing Function

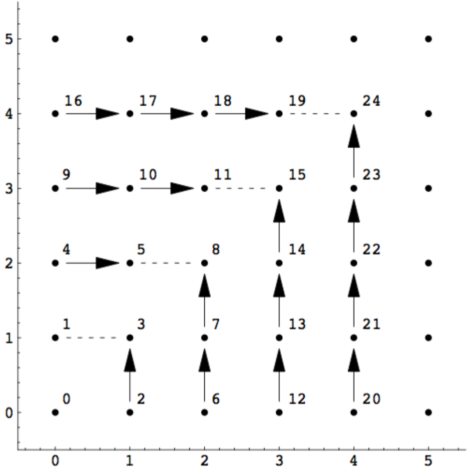
- The *elegant pairing function* is defined as follows:

$$\psi(x_1, x_2) = \begin{cases} x_1^2 + x_1 + x_2, & \text{if } x_1 \geq x_2 \\ x_1 + x_2^2, & \text{otherwise} \end{cases}$$

- Note that $|\psi(x_1, x_2)| = 2 \cdot \max(|x_1|, |x_2|)$
- For $y = \psi(x_1, x_2)$, let $s = \lfloor \sqrt{y} \rfloor$ and $t = y - s^2$
- This implies:

$$(x_1, x_2) = \psi^{-1}(y) = \begin{cases} (t, s), & \text{if } t < s \\ (s, t - s), & \text{otherwise} \end{cases}$$

Elegant Pairing Function



Tuple Function

- Any pairing function can be generalized recursively to a tuple function $\psi_k : \mathbb{N}^k \to \mathbb{N}$:

$$\psi_k(x_1, \ldots, x_k) = \begin{cases} \psi(\psi_{k-1}(x_1, \ldots, x_{k-1}), x_k), & \text{if } k > 2 \\ \psi(x_1, x_2), & \text{if } k = 2 \end{cases}$$

- The problem with this construction is the exponential size of the result (relative to k)
- Improved generalization with linear size (relative to k):

$$\psi_k(x_1, \ldots, x_k) = \begin{cases} \psi_{\frac{k}{2}}(\psi(x_1, x_2), \ldots,), & \text{if } k > 2 \text{ is even} \\ \psi_{\frac{k+1}{2}}(\psi(x_1, x_2), \ldots, x_k), & \text{if } k > 2 \text{ is odd} \\ \psi(x_1, x_2), & \text{if } k = 2 \end{cases}$$

Outline

Hash Function with Multiple Arguments

Representing Elements by Natural Numbers

Simplified Element Interfaces

Proper RSA Implementation

Nomenclature and Factories

Extensions

Groups and Elements

- UniCrypt works with *atomic elements* of *atomic groups*
  - → `ZPlus`: $\mathbb{Z}$
  - → `ZPlusMod`: $\mathbb{Z}_n$
  - → `ZStarMod`: $\mathbb{Z}_n^*$
  - → `GStarMod`, `GStarPrime`, `GStarSave`: $G_q \subseteq \mathbb{Z}_n^*$
  - → `BooleanGroup`: $\mathbb{B} = \{true, false\}$
  - → `SingletonGroup`: $\mathbb{S} = \{\diamond\}$
  - → `PermutationGroup`: Π_k
- and arbitrarily complex *tuple elements* of
  - → `ProductGroup`: $G_1 \times \cdots \times G_k$
  - → `PowerGroup`: G^k

Representing Atomic Elements

- For every atomic group G, there is an injective function
  - → $\phi_G : G \to \mathbb{Z}$, for `ZPlus`
  - → $\phi_G : G \to \mathbb{N}$, for all other groups

  with $\phi_G^{-1}(\phi_G(e)) = e$ for all $e \in G$
- In UniCrypt, $\phi_G(e)$ corresponds to `G.getBigInteger(e)` and $\phi_G^{-1}(x)$ corresponds to `G.createElement(x)`
- No such mapping exists for tuple elements, i.e., for elements of `ProductGroup` or `PowerGroup`
- **Goal**: Representation $\phi_G : G \to \mathbb{N}$ for all possible elements

Represening Elements of `ZPlus`

- Currently, `ZPlus` is needed as a group of infinite order
  - → $(\mathbb{Z}, +)$ is a cyclic group
  - → $(\mathbb{Z}, *)$, $(\mathbb{N}, +)$, $(\mathbb{N}, *)$, ... are not groups
- For an element of `ZPlus`, we can apply any invertible function $F : \mathbb{Z} \to \mathbb{N}$ to its integer representation
- For example, let

$$F(x) = \begin{cases} 2x, & \text{if } x \geq 0 \\ -(2x+1), & \text{otherwise} \end{cases}$$

- For $y = F(x)$, we get

$$x = F^{-1}(y) = \begin{cases} \frac{1}{2}z, & \text{if } Y \text{ is even} \\ -\frac{1}{2}(y+1), & \text{otherwise} \end{cases}$$

Representing Tuple Elements

- For elements of `ProductGroup` and `PowerGroup`, we can recursively apply the tuple function ψ_k
- Let $(e_1, \ldots, e_k) \in G$ be an element of $G = G_1 \times \cdots \times G_k$, then

  $$\phi_G(e_1, \ldots, e_k) = \psi_k(\phi_{G_1}(e_1), \ldots, \phi_{G_k}(e_k))$$

  is its unique integer representation
- Note that if all group elements can be represented uniquely by a natural number, then we can directly compute the element's hash value

Outline

Elements and Sub-Types

- In UniCrypt, there is a one-to-one correspondence between groups and elements
  - → `Group` ⇔ Element
  - → `AtomicGroup` ⇔ AtomicElement
  - → `AdditiveGroup` ⇔ AdditiveElement
  - → `MultiplicativeGroup` ⇔ MultiplicativeElement
  - → `BooleanGroup` ⇔ BooleanElement
  - → `ProductGroup`, `PowerGroup` ⇔ TupleElement
  - → `PermutationGroup` ⇔ PermutationElement
- Trivial type casts are often necessary to guarantee this correspondence
- Generics seems to be a natural solution, but it only works in one direction

Proposal for UniCrypt 2.0

- Reduce everything to `Element`:
  - → `AtomicElement` is no longer needed, since all elements map to natural numbers (see previous section)
  - → `AdditiveElement` only adds syntactic sugar for writing `e1.add(e2)` instead of `e1.apply(e2)`
  - → `MultiplicativeElement` only adds syntactic sugar for writing `e1.multiply(e2)` instead of `e1.apply(e2)`
  - → `BooleanElement` only adds syntactic sugar for writing `true/false` instead of `1/0`
  - → `TupleElement` allows accessing the i-the component of a tuple element of arity k, but this includes the 'atomic' case of $k = 1$
- `PermutationElement` seems to be the only true specialization of `Element`

Outline

Hash Function with Multiple Arguments

Representing Elements by Natural Numbers

Simplified Element Interfaces

Proper RSA Implementation

Nomenclature and Factories

Extensions

Algebraic View of RSA

- RSA does not work with groups
  - → $n = p{\cdot}q$
  - → $e \in \mathbb{Z}^*_{\phi(n)}$
  - → $d = e^{-1} \bmod \phi(n)$
  - → $m \in \mathbb{Z}_n$
  - → $c = Enc_e(m) = m^e \bmod n \in \mathbb{Z}_n$
- $Enc_e : \mathbb{Z}_n \to \mathbb{Z}_n$, but $(\mathbb{Z}_n, *)$ is not a group
- However, $(\mathbb{Z}_n, *)$ is a *monoid* (group without invertibility)
- Therefore, to implement RSA properly in UniCrypt, we need a super-type `Monoid` of `Group`
- Note that $(\mathbb{N}, +)$ is also a monoid: `NPlus` instead of `ZPlus`?

Outline

Hash Function with Multiple Arguments

Representing Elements by Natural Numbers

Simplified Element Interfaces

Proper RSA Implementation

Nomenclature and Factories

Extensions

UniCrypt Nomenclature

- The UniCrypt nomenclature for interface and class names is very self-consistent:
  - → `class UserClass implements User`
  - → `class UserAbstract implements User`
- Other naming styles:
  - → `class User implements IUser`
  - → `class CUser implements IUser`
  - → `class UserImpl implements User`
  - → `class DefaultUser implements User`
  - → `class AbstractUser implements User`

Suggestions and Solutions

- "Name your interface what it is."
  - → If your interface is a *truck*, call it `Truck` (not `ITruck` because it isn't an *itruck*)
  - → Then write implementations `DumpTruck`, `CementTruck`, etc.
  - → Don't call it `TruckClass` that is tautology just as bad as `TruckImpl` or `ITruck`
- "If you ever will have only one implementation, skip the interface. It creates this naming problem and adds nothing."
- "Hide the implementation behind a static factory method like `Trucks.create()`, for example as an anonymous inner class."

Outline

Hash Function with Multiple Arguments

Representing Elements by Natural Numbers

Simplified Element Interfaces

Proper RSA Implementation

Nomenclature and Factories

Extensions

Extensions

- Elliptic curves (student UniFR)
  - → ECGroup
  - → ECC (ECElGamalEncryption, ECPedersenCommitment, etc.)
- Zero-Knowledge Proofs
  - → OR-composition (Jürg, Philémon)
  - → Validity proof (Jürg, Philémon)
  - → Batch proofs (Philipp)
  - → Proof of shuffle (Philipp)
  - → ECC proofs
- Secret sharing (Jürg)
  - → Shamir's secret sharing
  - → Verifiable secret sharing
- Symmetric encryption: AES