Implementation of signature schemes with additional properties in C and Java

Cryptography Lab Project 2011
Oleg Arenz, Sarah Magin
1 Identity Based Identification and Signature
  1.1 How the Protocol works ................................................. 2
  1.2 Java Implementation .................................................. 3
    1.2.1 IdentityBasedProver .............................................. 4
    1.2.2 IdentityBasedVerifier ............................................ 4
    1.2.3 IBSTester .......................................................... 4
  1.3 C implementation .................................................. 4
    1.3.1 sizes.h ........................................................... 5
    1.3.2 identitybasedprotocol.c .......................................... 5

2 A Threshold Ring Authentication and Signature Scheme based on Coding Theory 6
  2.1 How the Protocol works ................................................. 6
  2.2 Our implementation .................................................. 7
    2.2.1 Key generation .................................................. 7
    2.2.2 Ring member simulation ......................................... 7
    2.2.3 Block permutation ............................................... 7
  2.3 Java Implementation .................................................. 8
    2.3.1 Commandline Parameters ....................................... 8
    2.3.2 Classes and Responsibilities ................................... 8
    2.3.3 Design Issues ................................................... 9
    2.3.4 Evaluation of the Implementation ............................ 9
  2.4 C Implementation .................................................. 9
    2.4.1 Commandline Parameters ...................................... 10
    2.4.2 Source files .................................................... 10
    2.4.3 Design Issues .................................................. 10
    2.4.4 Evaluation of the Implementation ............................ 10

3 Literature ................................................................. 11
1 Identity Based Identification and Signature

With identity based identification schemes the public key can be derived from unique identifiers of the sender, e.g. his e-mail address. This prevents attacks, which are based on taking over an identity by pretending that the own public key is in fact the one of a different person. Pierre-Louis Cayrel, Philippe Gaborit, David Galando and Marc Girault proposed the first code-based identity-based identification scheme in [1] and [2].

1.1 How the Protocol works

The basic flow of the identification scheme is as follows: First a master key is generated which is held by an authority. This master key is used to extract matching pairs of private keys and identifiers (the public keys). After obtaining his corresponding private key from the authority, the prover can use this key to prove his identity to the verifier. The CFS signature scheme is used for master key generation and key extraction. Stern’s zero-knowledge protocol allows the prover then to prove possession of his private key and thus identifying himself to the verifier. The three steps Master Key Generation, Key Extraction and Identification are explained in more detail below.

- **Master Key Generation:** Let $C$ be a q-ary linear code of length $n$ and of dimension $k$. Let $H$ be a parity check matrix of $C$. Let $V$ be an invertible matrix of size $(n-k) \times (n-k)$. Let $P$ be a permutation matrix of size $n \times n$. Then $\tilde{H} = VHP$ is used as the public key, while the decomposition of $\tilde{H}$ remains private. Additionally a hash function $h$ with range $\{0,1\}^{n-k}$ is provided.

- **Key Extraction:** In order to find the corresponding private key to an identifier $id$, an $s$ is searched with $h(id) = \tilde{H}^s$. However, as not every $h(id)$ is necessarily in the space of decodable elements, such an $s$ can not be found for every given id. This can be solved by using the CFS scheme. Hereby an index is concatenated to the id until it is decodable.

| 1. $i=0$ |
| 2. decode $h(id || i) = x'$ |
| 3. if no $x'$ is found, then $i=i+1$ and go back to step 2, else |
| 4. $s=D(h(id || i))$ |

This leads to the private key $(s, i)$.

- **Identification:** Identification is achieved by a slight modification of Stern’s protocols. We use the same public key $\tilde{H}$ as was used by the CFS scheme. In order to proof her identity to Bob, Alice picks a random n-bit word $y$, and a permutation $\sigma$ of $\{1, \ldots, n\}$. The identification consists of five steps:

1. Alice sends $c1 = h(\sigma || \tilde{H}^Ty^T)$, $c2 = h(\sigma(y))$ and $c3 = h(\sigma(y \oplus s))$ to Bob.
2. Bob picks a random number $b \in \{0,1,2\}$ and sends it to Alice
3. if $b = 0$, Alice reveals $y$ and $\sigma$
– if $b = 1$, Alice reveals $(y \oplus s)$ and $\sigma$
– if $b = 2$, Alice reveals $\sigma(y)$ and $\sigma(s)$

4. – if $b = 0$, Bob verifies that $c_1$ and $c_2$ were calculated correctly
– if $b = 1$, Bob verifies that $c_1$ and $c_3$ were calculated correctly
– if $b = 2$, Bob verifies that $c_2$ and $c_3$ were calculated correctly and that the weight of $\sigma(y)$ is $t = \frac{n-k}{\log_2(n)}$

5. Step 1-4 are repeated several times to reduce the possibility of cheating.

An identity-based signature scheme can be derived from this identification scheme by applying the Fiat-Shamir heuristic.

1.2 Java Implementation

The implementation in Java uses the existing implementations of the CFS and Stern scheme. They can both be downloaded on http://cayrel.net/spip.php?rubrique80. Our implementation combines those two. It currently consists of two classes and a test class as you can see in 1.
1.2.1 IdentityBasedProver

This class stands for the person who likes to prove his identity. It contains all the methods needed for this protocol.

First of all the prover needs a key pair in the classical way. To generate his keypair according to a Goppa code with the method `generateKeyPair()` two parameters are needed:

- **m**: influences the length and dimension of the code
- **t**: number of t correcting errors

The size of $\tilde{H}$ for a Goppa code $[2^m, 2^m - t m, t]$ is therefore

$$2^m t m$$

This method fills the global variables `publicKey` and `privateKey` as well as converting the public key to the representation needed by the Stern Protocol later: $mH\text{Stern}$. It also triggers the creation of a Pseudorandom Number Generator `prng`. With that keypair the prover can generate the secret $s$ using the identity. The `identity` is stored in a byte array. The resolving secret is stored as byte and as BitSequence representation in `secret` and `bitsecret`. Furthermore the publicIdentifier can now be calculated as the product of public key and secret. It should be the same as $h(\text{identity}||\text{index})$. The class offers two more methods to work with the Stern protocol.

- **makeSternCommit**: Condition for the usage of this method is that a secret was calculated. If so, then this method uses the class SternProver from the stern implementation to make a commitment.
- **giveSternAnswer**: After receiving a challenge from the verifier this method can be used to give an answer to this challenge and the commitment he made before.

1.2.2 IdentityBasedVerifier

This class stands for the person who likes to verify the identity of the prover. Therefore he needs the public Key of the prover and his public identifier. There are basically only two important methods in this class:

- **generateSternChallenge()**: int: This method computes a challenge for the prover. Simply said it just generates a random number between 0 and 2.
- **getSternVerify(Commitment, Answer, int)**: boolean: This method accesses a method in the stern package and returns true if the prover won the challenge and false if the signer lost it.

1.2.3 IBSTester

This class is only for testing. Its only method is the main-method. It is not considered for further usage.

1.3 C implementation

The C implementation consists of a lot of files, but to understand the identity-based identification only `sizes.h` and `identitybasedprotol.c` are important. However, all the other files are also needed.
1.3.1 sizes.h

In the sizes.h-File you can change your parameters $m$ and $t$. The corresponding variables are called EXTDEGREE and NBERRRORS. Unfortunately the keygeneration algorithm seems to cause a stackoverflow when choosing the extension degree to big.

1.3.2 identitybasedprotocol.c

In this file one can find all the method needed to execute the identity-based protocol. The work flow is as follows:

1. void setparameter(): Sets the parameters $n$, $r$, $w$, $rr$ and $lognw$ which are needed for the Stern Protocol according to the chosen parameters $m$ and $t$ needed for the CFS scheme.

2. void ibpgeneratekeypair(polyt*gen, gft*cfsLinv, unsigned char*pk): Generates a keypair matching the parameters in sizes.h and fills the global variable H(= public key for Stern Protocol)

3. unsigned char* ibpgeneratesecret(unsigned char*identity, polyt*keypair, gft*cfsLinv, unsigned char*pk): Generates a secret $s$ using the CFS scheme. The secret is formatted in a bit-for-bit representation

4. void ibpsterncommit(unsigned char*secret, BitSequence*c1, BitSequence*c2, BitSequence*c3, unsigned int*sig, BitSequence*sigsseed, BitSequence*y, BitSequence*yseed): computes the commitment and all needed variables for a Stern Commitment. The secret has to be in byte-for-bit representation

5. int ibpsternchallenge(): computes a challenge (returns a value between 0 and 2)

6. void ibpsternanswer(unsigned int b, const BitSequence const*secret, const unsigned int const*sig, const BitSequence const*sigsseed, const BitSequence const*y, const BitSequence const*yseed, BitSequence*a1, BitSequence*a2): Computes the answer to the given challenge. The needed parameters are calculated at the Stern Callenge

7. int ibpsternverify(const unsigned int b, const BitSequence const*iP, const BitSequence const*c1, const BitSequence const*c2, const BitSequence const*c3, const BitSequence const*a1,const BitSequence const*a2): Verifies the answer of the prover. The value 1 means the prover has won the challenge.
2 A Threshold Ring Authentication and Signature Scheme based on Coding Theory

This scheme - which was proposed by Carlos Aguilar Melchor, Pierre-Louis Cayrel and Philippe Gaborit in [3] - allows to prove, that t members out of a group of size N signed a message, without revealing the identities of the signers. It is based on Coding Theory and - more precisely - on the Minimum Distance Problem. A short explanation of the protocol is given below.

2.1 How the Protocol works

The authentication is carried out between a leader and a verifier. The leader wants to authenticate something to the verifier on behalf of t provers. To do so he forwards messages from the verifier to the provers and vice versa. In order to ensure that no information regarding the identity of the provers is leaked a modification of the Stern zero-knowledge authentication protocol is used for all communications between the verifier and the leader and between the leader and the provers. The following sketch shows how these different executions of the Stern protocol interact:

1. The leader collects the Stern-commitments of the t provers and creates commitments for the remaining n-t ring members by using their public key and zero as their secret key. He then uses a randomly chosen constant n-block permutation to create three master commitments which he sends to the verifier.

2. The verifier challenges the leader by sending him a random number 0 <= b <= 2.

3. In order to construct an answer for this challenge, the leader forwards the challenges to the provers and uses their answers and his block permutation to create his answer for the verifier. Again the n-t ring members who are not participating in the protocol are simulated with a private key zero.

4. The verifier confirms that the answer matches the commitment.

5. As it is possible to cheat if one anticipates the challenge the steps 1-4 are repeated several times.

This sketch shows why the Stern protocol needs to be modified for this scheme: The leader uses a block permutation to avoid damaging the commitments of the provers. The probably more significant difference to the original stern scheme however is that all public keys must be zero to allow the simulation of the n-t ring members with the private key zero and therefore it is now based on the Minimum Distance Problem instead of the Syndrome Decoding Problem. This additional restriction complicates the key generation, as we have to find a generator matrix for a given private key with $H \ast s^t = 0$. We will show a possible approach to generate this matrix in the next section, where we discuss our implementation.
2.2 Our implementation

In this section we describe our general approach to the implementation of the scheme. We will take a more detailed look at our implementations in the sections "Java Implementation" and "C Implementation".

2.2.1 Key generation

1. a vector of length n and weight w is chosen as secret key at random.
2. k-1 random vectors of length n and weight w are chosen at random.
3. the matrix formed by these vectors (every vector constitutes a row) is brought into reduced row echelon form by applying the Gauss-Jordan algorithm.
4. repeat steps 2-3 until the matrix is in systematic form.
5. the first k columns are removed (i.e. the part which forms the identity matrix)
6. the matrix is transposed
7. an identity matrix of size n-k is appended by adding n-k columns

The public key of the ring is an array which holds all matrices of the ring members.

2.2.2 Ring member simulation

Our implementation does not distinguish between those ring members who participate in the protocol and those who don’t. Upon initialization the ring members who don’t participate are created with their secret key set to 0.

2.2.3 Block permutation

Instead of sending a block permutation which includes the permutations of the ring members, we decided to send a N-bit permutation and the individual permutations of the ring members separately. The N-bit permutation is used as a constant n-block permutation. (N is the number of ring members, n is the key length). Sending the permutations as a whole would add additional complexity both on constructing and decomposing the block permutation. Note that sending the individual parts of the block permutation does not provide the verifier with additional information as a decomposition of the n-block permutation is possible anyway (as n is known to the verifier).
2.3 Java Implementation

Our Java implementation is based on the Stern implementation which can be downloaded at http://www.cayrel.net/research/code-based-cryptography/code-based-cryptosystems/article/implementation-of-code-based-zero. Most of the classes within the Stern package could be reused, most notably:

- stern.SternProver.java for modeling the ring members
- stern.communication.Commitment.java for modeling prover commitments as well as master commitments
- stern.communication.Answer.java for modeling prover answers
- stern.util.BitSequence.java for modeling vectors

The only class of the Stern package that has been modified is stern.control.StaticConfigurator which now points to our configuration files in the subdirectory ”ringConfig”. However we had to add several new classes, which all reside in the package ”thresholdRingAuthentication”.

2.3.1 Commandline Parameters

- -N <number of ring members>
- -t <number of provers>
- -rounds <number of rounds>
- -n <key length>
- -w <weight>

2.3.2 Classes and Responsibilities

- thresholdRingAuthentication.ThresholdRingAuthentication.java is the main class and controller of our implementation. It creates the participants of the protocol and determines the control flow.

- thresholdRingAuthentication.TRALeader.java is the leader in the protocol. It knows the ring members and communicates with them to create the master commitments and the answer.

- thresholdRingAuthentication.TRAVerifier.java is the verifier in the protocol. It knows the public keys and uses it to verify the answer. It also creates the challenge.

- thresholdRingAuthentication.comnication.TRAAnswer is a marker interface for the answers which are sent to the verifier. It is implemented by TRAAnswer1Impl TRAAnswer2Impl and TRAAnswer3Impl which represent the different answers to the different challenges. Leaving the interface empty and using different classes for the different answers allows assigning tailored getter-names and data types which makes the code more readable.

- thresholdRingAuthentication.KeyGeneration.KeyFactory is an interface which defines the methods randomPrivateKey() and createPublicKey(privateKey) used for key generation.
• `thresholdRingAuthentication.KeyGeneration.KeyFactoryImpl` implements `KeyGen`. It uses the algorithm for computing the public key that was shown earlier.

• `thresholdRingAuthentication.Matrix.TransformableMatrix` is an interface for a matrix that can be transformed. These transformable matrices are used by `KeyFactoryImpl`. The available transformations are: transformToReducedRowEchelonForm(), removeFirstColumns(int), transpose(), appendRow(BitSequence, int), appendMatrix(BitSequence[], int).


• `thresholdRingAuthentication.Matrix.MultipliableMatrix` is an interface for a matrix from the view of a prover. The only method it defines is multiplyWithVector(BitSequence). It can be created from a `TransformableMatrix`.


• `thresholdRingAuthentication.Matrix.MatrixAdapter` is an adapter which is given to the Stern-Prover as their public key. As we use `MultipliableMatrix` as the public key matrices within the `thresholdRingAuthentication` package while the `SternProver` use `stern.matrix.Matrix` we need this adapter.

### 2.3.3 Design Issues

The matrices are represented as arrays of BitSequences - one BitSequence per row. While this is faster than a two-dimensional array - especially for matrix multiplication - this has some side effects for matrix transformation: Appending rows is costly as a new array has to be initialized to override the existing one. Deletion of rows and columns is achieved by decrementing the special integer variables numRows and numColumns. While this is faster this leads to inconsistency between array dimensions and matrix dimensions.

### 2.3.4 Evaluation of the Implementation

The Java implementation is very clean. Easy to understand and modify. It is however noticeable slower than the C implementation. This is due to the fact that the Java implementation of Stern is slower than the C implementation of Stern and that direct memory manipulation is used within the C implementation.

### 2.4 C Implementation

Our C implementation is based on the C implementation of Stern which can be downloaded at http://www.cayrel.net/research/code-based-cryptography/code-based-cryptosystems/article/implementation-of-code-based-zero. Several modifications of the Stern code had to be done to integrate our implementation as the methods of the C implementation work with hard coded parameters.
2.4.1 Commandline Parameters

- `-N <number of ring members>`
- `-t <number of provers>`
- `-r <number of rounds>`
- `-p <key length> <row length> <weight> <r>'`
- `-k <row length>` (override row length given with -p)

2.4.2 Source files

- `ringAuthMain.c` homes the main-function. It is responsible for reading the commandline parameters, and initiates key generation and the start of the protocol.

- `ringAuth.c` defines methods for the protocol between the leader and the verify: commit(), answer(), and verify(). The challenge can be created with stern_challenge() in stern.c.

- `stern.c` provides methods for the protocol between the leader and the provers (commit, answer and verify). In the original package this file also defined the main method; but all of that code has been removed and copied to ringAuthMain.c.

- `keygen.c` is responsible for the generation of key pairs. The important methods are randBitSeqFixedWeight(BitSequence *placeToCreate) to create the private key, and void generateKeyPair(BitSequence * publicKey, BitSequence * privateKey), which creates a fitting matrix for the privateKey given as parameter.

- `permutation.c` is used to create a permutation of a specified length. It is used to create a permutation which can be used as a constant block permutation. It uses the code of sigma.c from the original stern package, which was modified to take the length of the permutation as a parameter. (sigma.c always creates permutations of length n)

2.4.3 Design Issues

The design is very fragile and harder to understand than the Java design. Bitsequences are mostly stored in byte-for-bit-representation for modification and in bit-for-bit-representation for communication. This makes it easy to run into segmentation faults after modifications of the code. The individual steps of the protocol are strictly speaking not in the correct order, as the challenge for the provers is sent and their answers are received before the leader creates the master commitment. Though the challenge is already calculated at that point in time the method which creates the master commitment does not have access to it. This unorthodox sequence was used in order to reuse the commit- and answer-implementation for the communication between the leader and the provers. As the C implementation does not have a special class for the leader, he is simulated by ringAuthMain.c.

2.4.4 Evaluation of the Implementation

The C implementation is harder to understand and modify than the Java implementation but faster.
3 Literature