Secure Computation API

Primitives for Implementation

Yehuda Lindell

Department of Computer Science

Bar-Ilan University, Israel

# Layer 1 – Low-Level Primitives

**Libraries to use:** Bouncy Castle, CACE, Crypto++, Certicom (if possible)

**General**

1. GetRandom: use cryptographic call, if possible from OS
2. Randomness extraction HKDF: <http://eprint.iacr.org/2010/264>

**Pseudorandom generator**

1. RC4 (throw out first 1024 bytes)
2. SHA-based PRG

**Pseudorandom permutations**

1. AES\_128
2. AES\_256
3. 3DES
4. PRP for arbitrary input/output length [Katz-Lindell, Section 6.6; start with PRF with length half of that needed for the PRP]

**Pseudorandom functions**

1. All of the pseudorandom permutations
2. HMAC with any hash function
3. PRFs with arbitrary input and output length
   1. Arbitrary input length: using any of the HMAC-SHA constructions
   2. Arbitrary output length:
      1. If shorter than basic, truncate
      2. If longer than basic: apply multiple times with CTR index and the output length

**Collision-resistant hash functions**

1. SHA-1
2. SHA-256
3. SHA-224 (this is truncated SHA-256)
4. SHA-512
5. SHA-384 (this is truncated SHA-512)
6. Universal one-way/target collision-resistant hashing: <http://webee.technion.ac.il/~hugo/rhash/rhash.pdf>

**Universal hash functions**

1. Perfect universal hash functions (use package giving GF[2k] computations)
2. GF[2k] operations: <http://www.shoup.net/ntl/doc/tour-modules.html>
3. <http://www.cosic.esat.kuleuven.be/publications/article-73.ps>

**Trapdoor permutations**

1. RSA
2. Rabin [Katz-Lindell, Section 11.2]

**Discrete log**

1. **Zp\*** for p of length 1024, with p=2q+1 and q prime (group of order q); fine also with general q
2. **Zp\*** for p of length 2048, with p=2q+1 and q prime (group of order q)
3. Elliptic curve groups: as given by NIST and implemented in bouncy castle, CACE or whatever you find

Minimal operations:

1. Get generator
2. Get random element
3. Multiply group elements
4. Add, multiply and find inverse mod q (the order)
5. Compute inverse of a group element
6. Exponentiate
7. Multiple exponentiations with same base
8. Verify membership in the group
9. grhs computations
10. ***Bilinear operations*** [references, CACE]

**Quadratic residuosity (QR)** (version 2.0 of the SDK; not at this stage)

1. Modulus of length 1024
2. Modulus of length 2048

**Lattice constructions** (version 2.0)

**Note on lengths:** For all of the above, if the underlying library allows more flexibility with sizes (modulus size and dlog with arbitrary sizes for p and q) then this is even better. The above is the “minimum”.

# Layer 2 – Non-Interactive Primitives

**Message authentication codes**

1. CBC-MAC with length in first block, using all pseudorandom permutations
2. HMAC for any hash function

**Symmetric encryption**

* CBC mode (random IV) using all pseudorandom permutations

1. CTR mode (random IV) using all pseudorandom permutations
2. Encrypt-then-MAC (any encrypt mode and any MAC)

**Asymmetric encryption**

1. El Gamal over given discrete log groups
2. RSA-OAEP, PKCS v2.1
3. Cramer-Shoup – DDH (<http://en.wikipedia.org/wiki/Cramer%E2%80%93Shoup_cryptosystem>)
4. Cramer-Shoup – N-residuosity (version 2.0)
5. Paillier\_1024 (Damgard-Jurik version)
6. Paillier\_2048 (Damgard-Jurik version)
7. QR\_Blum-Goldwasser (version 2.0)

**Homomorphic encryption**

1. Paillier (Damgard-Jurik version)
2. El Gamal (in the exponent)
3. QR\_Blum-Goldwasser (version 2.0)

**Special encryption** (version 2.0)

1. Attribute-based encryption (from SHARPS)
2. Format-preserving encryption [FFX standard]
3. Order-preserving encryption
4. Identity-based encryption

**Digital signatures**

1. RSA-PSS, PKCS v2.1
2. DSA, ECDSA
3. Hash-based one-time signatures [Katz-Lindell, Section 12.5]
4. Cramer-Shoup (version 2.0)

# Layer 3 – Interactive Protocols

**Standard commitment schemes**

1. Pedersen commitments: <http://cs.nyu.edu/courses/fall01/G22.3033-003/lect/lecture14.ps> , Section 2.5
2. Heuristic hash-based commitments: define Commit(x) = HASH(x||r) where r is 128-bits random
3. Statistically-hiding hash-based commitments: <http://cs.nyu.edu/courses/fall01/G22.3033-003/lect/lecture14.ps> , Section 2.3
4. Public-key encryption based commitments: commit to x by choosing new (pk,sk) and sending (pk,E(pk,x)); specifically El Gamal

**Trapdoor (equivocal) commitment schemes**

1. Any commitment with a ZK of committed value; instantiated at first with Pedersen and ElGamal commitments

**Extractable commitment schemes**

1. Any commitment with a ZKPOK for the commitment; instantiated at first with Pedersen and ElGamal commitments.

**Full-trapdoor commitments (equivocal and extractable)**

1. Use ZK and ZKPOK as in equivocal and extractable
2. Using a hash function as a random oracle
3. Using ElGamal encryption and a CRS

**Homomorphic commitment schemes**

1. Additive homomorphic operations on Pedersen
2. Multiplicative homomorphic on ElGamal

**UC-Secure commitments**

1. New protocol [Lindell2010]
2. Based on random oracle

**Non-malleable commitments**

1. Based on hash function (heuristic)
2. Based on non-malleable encryption in the CRS (common reference string) model

**Sigma protocol**

1. Sigma protocol of DLOG
2. Sigma protocol of Diffie-Hellman tuple
3. Sigma protocols for Damgard-Jurik (Paillier)
4. Sigma protocol that know committed value for Pedersen commitments
5. Sigma protocol that know committed value for ElGamal commitments
6. Sigma protocol that Pedersen committed value is *x*
7. Sigma protocol that ElGamal committed value is *x*
8. Sigma protocol of ElGamal secret key
9. AND of multiple sigma protocols
10. OR of two sigma protocols
11. OR of multiple sigma protocols
12. General compound statements (AND/OR) of sigma protocols
13. Template for Sigma protocol – programmer fills in procedures as below and Sigma protocol is built automatically:
    1. Prover compute 1st message
    2. Prover compute 2nd message
    3. Verifier check
    4. Verifier query length
    5. Optional: simulator instructions for generating prover messages (for OR and general compound statements)

**Zero knowledge**

1. Zero-knowledge for every Sigma-protocol using any perfectly-hiding commitment
2. ZKPOK for every Sigma-protocol using any perfectly-hiding equivocal commitment
3. Fiat-Shamir transform for any Sigma protocol: just get verifier message by HASH(x,first-message)
4. UC-secure ZKPOK from any sigma protocol, using any UC-secure commitment

**Coin tossing**

1. Basic Blum single-coin tossing using any commitment scheme
   1. P1 commits to a single random bit using any commitment scheme
   2. P2 sends a random bit to P1
   3. P1 decommits
   4. Both parties output XOR of bits
2. [Lindell01] coin tossing
   1. P1 commits to a random r
   2. P1 proves in ZKPOK that it knows the committed value
   3. P2 sends a random s
   4. P1 sends r (without decommitting)
   5. P1 proves in ZKPOK that r is the committed value (item 6 in sigma)
   6. Both parties output XOR of r and s
3. [Lindell01] coin tossing, version 2
   1. P1 commits to a random r using a fully-trapdoor commitment scheme
   2. P2 sends a random s
   3. P1 decommits to r
   4. Both parties output XOR of r and s
4. Semi-simulatable coin-tossing
   1. P1 sends a perfectly-hiding commitment to r
   2. P2 sends a perfectly-binding commitment to s
   3. P1 opens r
   4. P2 opens s
   5. Both parties output XOR of r and s

**Oblivious transfer**

1. Naor-Pinkas, privacy only (using any DH group) [HL, Section 7.2.1]
2. AIR (using any homomorphic encryption); version 2 [HL, Section 7.2.2]
3. PVW\_plain (using any DH group or N-residuosity) [HL, Section 7.5] and [PVW]
4. PVW\_plain (using a Fiat-Shamir proof instead) [HL, Section 7.5 at the end]
5. PVW\_UC in the CRS model (using any DH group or N-residuosity) [PVW], <http://www.cc.gatech.edu/~cpeikert/pubs/OTpaper.pdf>

**Batch oblivious transfer**

1. PVW\_plain in batch mode [HL, Section 7.5]
2. PVW\_plain-Fiat-Shamir in batch mode [HL, Section 7.5]
3. PVW\_UC in batch mode [HL, Section 7.5]

**Oblivious polynomial evaluation**

1. Based on OT
2. Based on homomorphic encryption
3. Fully-secure oblivious polynomial evaluation (?)

**Oblivious pseudorandom function evaluation**

1. Based on OT [HL, Section 7.6]

**Key exchange protocols**

1. Naïve (use given symmetric keys directly)
2. Passive Diffie-Hellman
3. UC-secure key exchange
4. SSL-based key exchange (maybe)

**Authenticated broadcast channel**

1. Exponential broadcast for a small number of parties
2. Authenticated broadcast
3. Simultaneous broadcast

**Information theoretic techniques**

1. Secret sharing (including general polynomial interpolation)
   1. Class that can work over any field
   2. Use Zp\* as basic field (here p can be small)
2. VSS
3. Arithmetic circuit protocols (addition, multiplication…)

# Important Note

A crucial part of the design phase will be a pseudocode write-up of all the required primitives. This is to ensure that the translation from a protocol described in a paper to one that is to be implemented is correct.

## Bibliography

Katz-Lindell: Introduction to Modern Cryptography

HL: Hazay Lindell, Efficient Secure Two-Party Computation