

Java Implementation of Group and Blind Signatures

Orange Labs

Nicolas Desmoulins, Research & Development
21/06/2010, Trust2010, Anonymous Signature Workshop



unrestricted



Concept of group signature

- Group member can sign messages « on behalf of the group »
- Group signatures are anonymous and unlinkable for every verifier
 - Except, when needed, for a given authority
- Several variants exist
 - Example: List signature
 - also known as Direct Anonymous Attestation (DAA) (Breickell, Camenisch, Chen, ACM CSS 2004)
 - Enables in some cases to link the signatures of the same user
 - During specified sequence (a certain period of time)
- Implementation of ACJT based group signatures (Ateiese, Camenisch, Joye and Tsudik, Crypto2000)
 - GSignature
 - LSignature

Concept of Blind Signatures (1)

- A signer can sign messages for users
- The signer does not know the message he is signing
- The signer should not be able to recognize the message nor the signature he has produced
 - The user is anonymous w.r.t all other users
- BSignature implemented based on Schnorr Signature (Crypto1989)
 - With several other variants of blind signatures

Concept of Blind Signatures (2)

- Variants:

- Partially blind signature (PBSignature)
(Abe, Fujisaki, Asiacrypt 1996)
 - In the case the signer needs to know some parts of the message
 - Message divided into two parts
 - M , known by both signer and user
 - m , only known by the user
- Invariable partially blind signature (IPBSignature)
(Canard, Malville, Traoré, DIM 2008)
 - If part of the blind part need to be used again in a blind signature process
- Fair blind signature (FBSignature)
(Stadler, Piveteau, Camenisch, Eurocrypt 1995)
 - In the case where it is necessary to revoke anonymity

Java Implementation

- Crypto Library with different Privacy Enhancing Cryptography (PEC)
 - Group Signature -> ACJT based schemes
 - Blind Signature -> Schnorr based schemes
- Schemes with different privacy properties
 - Choice will depend of the context of use
- Blind Signature and Group Signature more complex than classical Signature schemes
 - Lots of computation
 - In particular great number of modular exponentiations
 - What about performances?
 - Are these schemes limited to PC?
 - Strategies used to boost performances

Example of ACJT group signature

- A group signature is composed of the following elements
 - El Gamal encryption of A: $T_1 = Ay^w$ and $T_2 = g^w$
 - Commitment $T_3 = g^e h^w$ on e
 - Signature of knowledge U (implying the message)

$$\text{POK}(e, x, w, ew : T_1^e = a_0 a^x y^{ew} \wedge T_2 = g^w \wedge T_2^e = g^{ew} \wedge T_3 = g^e h^w)(m)$$

- Group signatures implemented
 - Between 26 and 31 modular exponentiations for signature generation

Optimizations – use of external native lib

- Modular exponentiation is the most costly operation
- Java offers "BigInteger" class to do computation on big numbers
 - modPow performance similar to pure C code
 - But native libs like OpenSSL or libGMP a lot faster
 - Use architecture-specific assembly code
 - **accelerate modular exponentiation by a factor 3!**
- Use of optimized libs from Java is interesting
 - "BigIntegerExt" class which
 - Offers the same API (plus additional methods) than BigInteger
 - Uses native lib if possible (only for modular exponentiation)
 - Uses Java BigInteger if not
 - modPow operation 3 times faster
 - But requires compilation of C glue code for each architecture

Optimizations – use of precomputations

- Lots of modular exponentiations are done with a modulus and bases known in advance $g_i^r \bmod p$
 - System parameters or part of public key
 - Some algorithms can accelerate computation with the help of some precomputation
 - "fixed-based comb" method in "HandBook of applied cryptography" (14.116, 14.117)
 - Can be configured for best performances/storage compromise
 - Typically single exponentiation 3.5 times faster
 - with 64 kBytes of precomputation per (base,modulus) couple
 - Java and C version of the algorithm developed

Optimizations – parallel computing

- Some computation can be done in parallel
 - Only useful in case of multi-core PC
 - Not always efficient as thread use has a cost
 - In our case, only really efficient with group signature

Benchmark implementation

- PC: Dell D620, Processor Intel Centrino Dual-core, T2300, 1.66GHz (2006 model)
- OS Linux 2.6.x, **32 bits**, JDK 1.6.0 (-server mode)
- Average performances of Java implementation
- Element of comparison:
 - Single 160/1024-bits exponentiation: java ~4 ms, native ~1 ms
 - Single 1024/1024-bits exponentiation: java ~25 ms, native ~9 ms
 - DSA Signature (in ms)

DSA, 1024 bits modulus, 160 bits exponent		
Signature	pur Java	4.5
	with native lib	1.5
Verification	pur Java	8.5
	with native lib	3

unrestricted

Group Signature schemes

- Based on ACJT group signature scheme
 - 2048 bits RSA modulus
- Computation time in ms
- Using precomputations (in parenthesis)
- Use parallel computation (1.7 gain here)

		GSignature	LSignature
signature	pure Java	2550 (2050)	2900 (2200)
	with native lib	820 (500)	950 (560)
verification	pure Java	2000 (1700)	2200 (1800)
	with native lib	680 (470)	750 (530)

- Take a lot of time even if optimizations help a lot
 - Fast enough to be practical on PC

Blind Signature schemes

- Based on Schnorr signature
 - 1024 prime modulus
- Computation time in ms
- Using precomputations (in parenthesis)

		BSignature	PBSignature	IPBSignature	FBSignature
signature	pure Java	22 (6)	34 (26)	85 (65)	140 (72)
	with native lib	7.5 (1.8)	15 (9)	34 (23)	46 (24)
verification	pure Java	8.5 (2.2)	17 (12)	27 (16)	45 (26)
	with native lib	3 (0.6)	7.5 (4)	11 (7)	15 (11)

- A lot faster than group signatures
 - Not surprising as group signature schemes use greater numbers (modulus 2 times greater, big exponents)

On JME mobile phones

- JME (Java Mobile Edition) phones
 - Java subset for mobile phone → limited API
 - No BigInteger
 - BigInteger from JavaSE was ported
 - Performances remain poor
 - More than 200 ms for 160/1024 modular exponentiation
 - But use of precomputations as efficient as on PC
 - Not possible to call native code from Java
- JME platform is not very good for cryptography
 - Fair blind signature should require less than 3 seconds
 - No good but still acceptable
 - GSignature too costly for tested mobiles

On Android mobile phones

- Tested on Android 1.5 phone, HTC Hero
 - No JIT compiler (java code is slow!) at least until Android 2.2
- Includes subset of Java5 API
 - Includes BigInteger class
 - Use OpenSSL with ARM-specific ASM code (5 times faster)
 - Modular exponentiation is fast
 - but for precomputation had to compile native lib (ndk)
 - Easy to re-use existing Java code
- Blind Signature perfs (in ms)

	BSignature	PBSignature	IPBSignature	FBSignature
signature	70 (20)	95 (80)	304 (215)	380 (220)
verification	22 (15)	43 (30)	79 (50)	120 (70)

- Group Signature: around 12s (7s) for signature, 7s (6s) for verification

unrestricted

Conclusion

- PEC can be integrated in a lot of devices
 - Useful tools to design privacy by design services
- In particular in today's mobile phones
 - powerful enough to do Blind Signatures
 - Group signatures performances remain an issue
 - But already accessible to some powerful phones
- Smartcard/SIM integration remains to be investigated
 - Preliminary results show that
 - Performance wise, SIM with cryptoprocessor fast enough for blind signatures
 - API (Javacard) not adapted to implement new schemes
 - requires ugly hacks (in the best case)

thank you



Orange, the Orange mark and any other Orange product or service names referred to in this material are trade marks of Orange Personal Communications Services Limited.
© Orange Personal Communications Services Limited.

Unauthorized reproduction is strictly prohibited.

