Digital Signature Schemes and Their Realization
CS290G Project

Esra Küçükoğuz

UCSB
Submitted to: Çetin Kaya Koç

12.8.2008

Content

- RSA
  - Algebraic Attack
- DSA
- ECDSA
- Parallelization of ECDSA Signature Generation
  - Parallelization
  - Securing the Point Multiplication
  - Securing the Modular Inversion
- Accelerating (EC)DSA Signature Verification
- EC Optimization
  - Point Halving Algorithm
- Hash Functions
- Key Collisions in (EC)DSA

RSA Digital Signature

- public keys: $k_{pub} = (n, b)$
- private keys: $k_{pr} = (p, q, a)$
- where $n = p \cdot q$

Finding private key of RSA is as hard as the factoring problem

Signature:

- $y = Sig_{k_{pr}}(x) = x^a (\mathrm{mod} n)$

Verification:

- Check if $x = Ver_{k_{pub}}(y) = y^b (\mathrm{mod} n)$?
  If they are not equal, signature is invalid.

Algebraic Attack on RSA

- If (x, y) and (x', y') are known:
- $x'' = (x \cdot x')$ and $y'' = y \cdot y' = (x \cdot x')^a (\mathrm{mod} n)$
- y'' is a valid signature for x'' which is actually never signed by the private key owner.

To prevent Algebraic Attack:

append z to x so y'' is not a valid signature for x'' anymore

- $x_{new} = x|z$ and $x'_{new} = x'|z$
- Therefore, $y_{new} \cdot y'_{new} \neq (x_{new} \cdot x'_{new})^a (\mathrm{mod} n)$

DSA

- Digital Signature Algorithm (DSA) is based on El-Gamal Signature scheme.
- Finding private key of DSA is as hard as the discrete logarithm problem.
- public keys: $k_{pub} = (\beta, \alpha, p)$
- private keys: $k_{pr} = (a)$
- where α is a primitive element in Z_p^*
- and $\beta = \alpha^a (\mathrm{mod} p)$ with a is a random $\in \{2, 3, \ldots, p-2\}$

DSA

Signature:

- Choose a random $k \in \{0, 1, 2, \ldots, p-2\}$ such that $(\gcd(k, p-1) = 1)$
- Compute signature $Sig_{k_{pr}}(x, k) = (\gamma, \delta)$
- $\gamma = \alpha^k (\mathrm{mod} p)$
- $\delta = (x - a \cdot \gamma) \cdot k^{-1} (\mathrm{mod} p - 1)$

Verification:

- Compute $Ver_{k_{pub}}(x, (\gamma, \alpha)) = \beta^{\gamma} \cdot \gamma^{\delta}$
- If result equals $\alpha^x (\mathrm{mod} p)$, then signature is valid.

ECDSA

- ECDSA is also based on El-Gamal Signature scheme.
- There is no known subexponential-time algorithm for the elliptic curve discrete logorithm problem.
- With the same security level for RSA, DSA, and ECDSA; ECDSA is the one that requires shortest key lenght.
- This makes ECDSA more attractive with less memory and bandwidth requirement.
- public keys: $k_{pub} = (E, P, Q)$
- private keys: $k_{pr} = (d)$
- where E is the elliptic curve, $P = (x_p, y_p)$ is the base point that belongs to the curve and has a prime order, $Q = (x_q, y_q) = d \cdot P$, and d is a random integer $\in [1, n-1]$

ECDSA

Signature:

- Choose a random $k \in \{1, 2, \ldots, n-1\}$
- Compute signature $Sig_{k_{pr}}(x, k) = (\gamma, \delta)$ and $P' = (x', y') = k \cdot P$ and h = H(x)
- $\gamma = x' (\mathrm{mod} n)$
- $\delta = k^{-1}(h + d \cdot \gamma)(\mathrm{mod} n)$

Verification:

- If γ and δ are not $\in [1, n-1]$, signature is invalid, ow:
- Compute $h = H(x)$ and $\omega = \delta^{-1}(\mathrm{mod} n)$ $\upsilon_1 = h \cdot \omega(\mathrm{mod} n)$ $\upsilon_2 = \gamma \cdot \omega(\mathrm{mod} n)$ $R = (x_r, y_r) = \upsilon_1 \cdot P + \upsilon_2 \cdot Q$
- If $\gamma = x_r(\mathrm{mod} n)$, then signature is valid.

A Parallelization of ECDSA Signature Generation

The Original Implementation

1. $e = \text{SHA-1}(t)$
2. choose a random $k \in [1, n-1]$
3. compute $k \cdot G = (x_1, y_1)$ and $\gamma = x_1 (\mathrm{mod} n)$. If $\gamma = 0$, go to step 2.
4. compute $z_0 = (e + d \cdot \gamma)(\mathrm{mod} n)$
5. compute $\delta = k^{-1} \cdot z_0 (\mathrm{mod} n)$. If $\delta = 0$, go to step 2.
6. output the signature (γ, δ)

where t is the message and d is the private key

A Parallelization of ECDSA Signature Generation

- In ECDSA, the point multiplication is the most computationally intensive operation. It is the step 3 in the original algorithm in the form of $k \cdot G$, where G is the *fixed* base point and k is a random integer number $\in [1, n-1]$
- Step 4 has a modular multiplication
- Step 5 has a modular division which can be computed with a modular inversion algortihm.

A Parallelization of ECDSA Signature Generation

- Parallel implementation is a good design choice with the advantage of:
  - improving the performance
  - reducing the requirement for gate count
- Which steps to parallelize?
  - Step 3 (point multiplication)can be parallelized.
  - Some portion of step 4 and 5 (multiprecision arithmetic) are independent of the step 3, therefore could be executed concurrently with step 3

A Parallelization of ECDSA Signature Generation

Parallelization

- Let's have two processing units $\mathcal{P}_1$ and $\mathcal{P}_2$.
- $\mathcal{P}_1$ executes some of the multiprecision arithmetic and also some of the point multiplication operation.
- $\mathcal{P}_2$ executes the remaining of the point multiplication operation.
- $\mathcal{P}_2$ is faster than $\mathcal{P}_1$ by a factor of Θ
- It is assumed that a microprocessor always exists and a hardware accelerator is implemented to improve the ECDSA performance.
- $\mathcal{P}_1$ corresponds to the microprocessor and $\mathcal{P}_2$ corresponds to the hardware accelerator.

A Parallelization of ECDSA Signature Generation

Parallelization

- The idea is one of the processing units can compute $k_1 \cdot P$ and other can compute $k_2 \cdot Q$ and then add these two points to get $k \cdot G$
  - $k \cdot G = k_1 \cdot G + 2^{m_1} \cdot k_2 \cdot G = k_1 \cdot G + k_2 \cdot Q$
  - $k = k_1 + 2^{m_1} \cdot k_2$
  - $Q = 2^{m_1} \cdot G$. Q is precomputed and stored.
  - $k = k_1 + 2^{m_1} \cdot k_2$
  - and $m_1 = \lfloor m/(\Theta + 1) \rfloor$ where m is the size of n in bits
- Since k_2 has Θ times as many bits as k_1 has, $\mathcal{P}_1$ and $\mathcal{P}_2$ is going to take approximately the same time to complete their assigned computations.

A Parallelization of ECDSA Signature Generation

The Parallel Implementation

1. $e = \text{SHA-1}(t)$
2. choose a random $k \in [1, n-1]$
3. compute $z_1 = k^{-1} (\text{mod} n) \|$ compute $P_2 = k_2 \cdot Q$
4. compute $z_2 = z_1 \cdot d (\text{mod} n) \|$
5. compute $z_3 = z_1 \cdot e (\text{mod} n) \|$
6. compute $P_1 = k_1 \cdot G$
7. compute $k \cdot G = P_1 + P_2 = (x_1, y_1)$ and $\gamma = x_1 (\text{mod} n)$
8. compute $\delta = (z_3 + z_2 \cdot \gamma) (\text{mod} n)$. If $\delta = 0$ or $\gamma = 0$, go to step 2.
9. output the signature (γ, δ)

In steps 3 to 6, right side is computed by processing unit $\mathcal{P}_2$. Left side of steps 3 to 6 and the other remaining steps are computed by processing unit $\mathcal{P}_2$. It is preferred that $\mathcal{P}_1$ executes the additional steps therefore gate count of $\mathcal{P}_2$ can be reduced.

SPA Attack on ECDSA

- In Simple Power Analysis attack, the attacker uses the relationship between the power consemed and the instructions performed by the processor to find the k. By using k and the signature, private key can be computed with the formula $d = \gamma^{-1}(k \cdot \delta - e)$
- In the original ECDSA implementation, the point multiplication and modular inversion operations are not resistant to SPA attacks.
- In the parallel implementation, the point multiplication and modular inversion operations are executed concurrently in parallel which makes the power trace of the two executions cover for each other.

Securing the Point Multiplication

Montgomery Point Multiplication

Input: $k = (k_{m-1}, \ldots, k_0)_2$ with $k_{m-1} = 1$ and $P = (x, y) \in E(\mathrm{GF}(2^m))$ and curve parameter b. Output: $k \cdot P$

1. $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_0 \leftarrow x^4 + b$, $Z_0 \leftarrow x^2$.
2. For i from $m-2$ to 0 :
   - 2.1 If $k_i = 1$:
     $T \leftarrow Z_1$, $Z_1 \leftarrow (X_1 \cdot Z_0 + X_0 \cdot T)^2$, $X_1 \leftarrow x \cdot Z_1 + (X_1 \cdot Z_0)(X_0 \cdot T)$
     $T \leftarrow X_0$, $X_0 \leftarrow X_0^4 + b \cdot Z_0^4$, $Z_0 \leftarrow T^2 \cdot Z_0^2$
   - 2.2 Else:
     $T \leftarrow Z_0$, $Z_0 \leftarrow (X_1 \cdot T + X_0 \cdot Z_1)^2$, $X_0 \leftarrow x \cdot Z_0 + (X_1 \cdot T)(X_0 \cdot Z_1)$
     $T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + b \cdot Z_1^4$, $Z_1 \leftarrow T^2 \cdot Z_1^2$
3. $x_2 \leftarrow X_1 / Z_1$.
4. $y_2 \leftarrow$
   $(x + X_1/Z_1)[(X_1 + x \cdot Z_1)(X_0 + x \cdot Z_0) + (x^2 + y)(Z_1 \cdot Z_0)](x \cdot Z_1 \cdot Z_0)^{-1} + y$.
5. return (x_2, y_2)

Securing the Point Multiplication

- Montgomery point multiplication is an efficient algorithm with $6 \cdot m + 10$ field multiplications and 1 field inversion defined over GF(2^m). It also requires no precomputation.
- Although it is not easy, if one can distinguish between steps 2.1 and 2.2 by using power analysis, then the attacker will be able to find k.
- Therefore, this algorithm should be modified such that it is resistant to SPA attacks.
- The modified implementation is ressistant to SPA attacks with the assumption of power trace of swap functions is not distinguishable.
- The modified implementation has same number of field multiplications and inversions with the original Montgomery point multiplication algorithm.

Securing the Point Multiplication

SPA Resistant Implementation of Point Multiplication

- Change Step 2:
- For i from $m-2$ to 0 :
  $R_1 \leftarrow X_1 \cdot Z_0$
  $R_2 \leftarrow X_0 \cdot Z_1$
  $R_3 \leftarrow X_{1-k_i}$
  $Z_{k_i} \leftarrow (R_1 + R_2)^2$
  $X_{k_i} \leftarrow x \cdot Z_{k_i} + R_1 \cdot R_2$
  $X_{1-k_i} \leftarrow R_3^4 + b \cdot Z_{1-k_i}^4$
  $Z_{1-k_i} \leftarrow R_3^2 \cdot Z_{1-k_i}^2$

Securing the Modular Inversion

Binary extended gcd algorithm

Input: Integer k such that $k < n$ where n is a large prime number
Output: $C = k^{-1} (\mathrm{mod} n)$

1. $u \leftarrow k$, $v \leftarrow n$, $A \leftarrow 1$, $C \leftarrow 0$.
2. while u is even:
   - 2.1 $u \leftarrow u/2$
   - 2.2 if A is even, then $A \leftarrow A/2$, else $A \leftarrow (A + n)/2$
3. while v is even:
   - 3.1 $v \leftarrow v/2$
   - 3.2 if C is even, then $C \leftarrow C/2$, else $C \leftarrow (C + n)/2$
4. - 4.1 If $u \geq v$ then $u \leftarrow u - v$, $A \leftarrow A - C$
   - 4.2 else $v \leftarrow v - u$, $C \leftarrow C - A$
5. If $u = 0$, then:
   - if $C > 0$ return C
   - else return $n + C$

   else go to step 2.

Securing the Modular Inversion

- In the binary extended gcd algorithm, step 1 can be modified such that it can perform modular division. Instead of initializing A with 1 it can be initialized with z_0
- In the algorithm, at each iteration either step 2 or step 3 is executed.
- An attacker can find k by using the number of times the steps 2 and 3 are executed and/or the conditionality in step 4
- To make this algorithm resistant to power analysis attacks, we can multiply k with a random number ψ and compute $(k \cdot \psi)^{-1}$ and multiply again with ψ. This can help, but requires 2 additional modular multiplications.
- A better solution exists where finding k is computationally expensive.

Securing the Modular Inversion

Modification

- Change Step 4:
- $auv_+ \leftarrow \text{addr}u$
  $auv_- \leftarrow \text{addr}v$
  $aAC_+ \leftarrow \text{addr}A$
  $aAC_- \leftarrow \text{addr}C$
  if $(u < v)$
  - swap (auv_+, auv_-)
  - swap (aAC_+, aAC_-)

  sub (auv_+, auv_-)
  sub (aAC_+, aAC_-)

Securing the Modular Inversion

Modification Cont.

- Store the addresses of the variables u, v, A, and C in auv_+, auv_-, aAC_+, and aAC_-.
- sub(aAC_+, aAC_-) corresponds to the operation $*aAC_+ \leftarrow *aAC_+ - *aAC_-$ where $*$ stand for the value pointed by that variable.
- Swap functions are the only steps containing conditionality in this algorithm. Therefore, if it is assumed that swap functions are indistinguishable in terms of power trace, then the modified algorithm is resistant to SPA attacks.

Accelerating (EC)DSA Signature Verification

- Computing $u_1 \cdot G + u_2 \cdot Q$ is a time consuming step in ECDSA signature verification.
- The point G is common for many signers, because usually EC parameters are taken from a standard.
- If signature verification is done repeatedly i.e. in an airport immigration office, then some common parameters are computed repeatedly in each verification.
- Intermediate results can be cached and stored to use in some of the future signature verification.
- In the long run, caching can help to reduce the total time for verification of a set of signatures.
- Note that: no assumption of all G's are common. Instead: In a set of signatures, some of them share same G.

Accelerating DSA and ECDSA Signature Verification

Use Caching to speed-up Multi-Exponentiation

- Multi-exponentiation: $\prod_{1 \leq i \leq k} g_i^{e_i}$
- g_i's are fixed base elements for $i \in [2, k]$ where g_1 is a variable and can reoccur in different signatures.
- For an appearing g_1 create the cache entry in the form of $(g_1, (\lambda_1, G_1), \ldots, (\lambda_{s-1}, G_{s-1}))$
- where λ is an integer and $G_i = g_i^{\lambda_i}$

Accelerating DSA and ECDSA Signature Verification

Use Caching to speed-up Multi-Exponentiation

- Whenever a cache entry is available for g_1:
  - First, parse the cache entry
  - Split e_1 into integers E_1 :
    - $d_0 = e_1$
    - for $1 \leq i \leq s-1$:
    - $E_i = \lfloor d_{i-1}/\lambda_i \rfloor$
    - $d_i = d_{i-1} - E_i \cdot \lambda_i$
    - $E_s = d_{s-1}$
  - Apply modular exponent splitting and radix-2 exponent splitting to $g_1^{e_1} = G_1^{E_1} \ldots G_{s-1}^{E_{s-1}} \cdot g_1^{E_s}$:

Accelerating DSA and ECDSA Signature Verification

Use Caching to speed-up Multi-Exponentiation

- whenever a new g_1 value is seen, output some intermediate results.
- whenever an old g_1 value appears, use cached intermediate results to speed-up the verification process.(previous page)
  - old value refers to a g_1 value which has a corresponding entry in the cache.
  - since cache entry takes read/write memory, it is not possible to put all g_1's to cache, some of them can be deleted later which makes that g_1 new again.

EC Software/Hardware Optimization

Point Halving Algorithm

- Point multiplication is a computationally expensive operation of ECDSA.
- Instead of point doubling, one can use point halving and speed-up the point multiplication.
- Let's take a point $P = (x_p, y_p)$, and find a point $Q = (x_q, y_q)$ where $2 \cdot Q = P$.
- Point Halving algorithm speeds up in software by a factor two to three with compared to point doubling algorithm.

EC Software/Hardware Optimization

Point Halving Algorithm over GF(2^m)

Input: $P \in E$
Output: $Q = 1/2 \cdot P$ such that $Q \in E$

1. $M_h = QuadraticSolve(x_P + a)$, a is EC parameter
2. $T = x_P \cdot M_h + y_P$
3. if parity$(T,\ t_m) = 0$: $M_h = M_h + 1$ and $T = T + x_P$
4. $x_Q = \sqrt{T}$
5. $r_Q = M_h + x_Q + 1$
6. $y_Q = x_Q \cdot r_Q$

EC Software/Hardware Optimization

Point Halving Algorithm

- Step 1: *QuadraticSolve* can be implemented in hardware with a small number of *XOR* gates
- Step 3: t_m is a mask and depend on the polynomial modulus.
- When y_Q is not required it can be eliminated which makes the algorithm needs only one field multiplication.
- This algorithm requires approximately 29,000 gates in hardware.

Hash Functions

- Problem with long messages and for security reasons entire message should be signed as a single document. To overcome these problem we use hash functions in digital signatures.

Signature:

- $z = h(x)$
- $Sig_{k_{pr}}(z)$

Verification:

- $z = h(x)$
- $Ver_{k_{pub}}(z)$

Requirements for Hash Functions

- $H(x)$ can be applied to x of any size
- $H(x)$ produces a fixed length output
- $H(x)$ is a relatively easy to compute in software and hardware
- $H(x)$ is one way: for any output z it is impossible to find x such that $H(x) = z$
- $H(x)$ is weak-collision resistant: given x and $H(x)$, it is impossible to find a $x'(\neq x)$ such that $H(x) = H(x')$
- $H(x)$ is strong-collision resistant: it si impossible to find any $x, x'(x \neq x)$ such that $H(x) = H(x')$

Attacks on Digital Signatures Using MD5 Message Digest

- MD5 is used in file integrity checking and as message digest in digital signatures.
- MD5 collision is published by *Wang et al*. It is not secure anymore to check message(file) integrity, because two colliding files can be created while the hash is the same.
- Two different meaningful files can map to same hash which makes the digital signatures vulnerable to possible attacks if used.
- Signer can sign a message m_1 but then the attacker can change the message m_1 with a colliding message m_2. So, verifier can verify that m_2 is signed which is actually not true.

Key Collisions in (EC)DSA

- Non-repudiation: One cannot deny his signature once signed.
- If key collisions occur in signature schemes, then one can verify that a message m is signed by entity e_1 eventhough m is actually signed by another entity e_2, not by e_1
- Key-collision searching is possible in DSA and ECDSA, there is a feasible message-independent algorithm for this purpose.(*T. Rosa*)

References

1. Aravamuthan, S., Rao, V., *A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks*, 2006.
2. Möller, B., Rupp, A., *Faster Multi-Exponentiation through Caching: Accelerating (EC)DSA Signature Verification*, 2007.
3. Schroeppel, R., Beaver, C.L., Gonzales R., Miller, R., Draelos, T., *A Low-Power Design for an Elliptic Curve Digital Signature Chip*, 2002.
4. Rodríguez-Henríquez, F., Saqib, N. A., Díaz-Pèrez, N.A., Koç, Ç.K., *Cryptographic Algorithms on Reconfigurable Hardware*, Springer-Verlag New York, Inc., 2006.
5. Mikle, O., *Practical Attacks on Digital Signatures Using MD5 Message Digest*, 2004.
6. Wang, X., Feng, D., Lai, X., Yu, H., *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, 2004.
7. Rosa, T., *Key-collisions in (EC)DSA: Attacking Non-repudiation*, 2002.