Elements of the public key cryptology DSA implementation

Daniel Trędewicz

April 2018

1 Introduction

The purpose of this document is to provide some insight into The Digital Signature Algorithm (abbreviated and referenced as DSA). Its description is provided in the publication [2].

A digital signature is an electronic analogue of a written signature. The digital signature can be used to provide assurance that the claimed signatory signed the information. In addition, a digital signature may be used to detect whether or not the information was modified after it was signed. For more details about ideas around digital signatures please refer [3] sections 11, 12 and 13.

2 DSA scheme

The scheme itself is described in detail in [2] sections 3 and 4, while approved methods of generating needed parameters are described [2] APPENDIX A and APPENDIX B, so there's no need to duplicate them here.

3 Chosen methods

For generating needed parameters I decided to go with SHA-512 as my hash function for subsequent calculations. I decided to generate probable primes p and q using method described in [2] APPENDIX A.1.1.2., while the generator g is obtained with routine form APPENDIX A.2.3. Per-message secret number k (and thus its inverse $\bmod q$ k^{-1}) is calculated with APPENDIX B.2.1 method.

4 Implementation

DSA is being performed by 3 applications: Signatory_gen_qpg, Signatory_sign_m and Validator. Signatory_gen_qpg generates values of q, p, g and writes them into one file qpg_values.txt. Then it computes private and public key and also stores them in separate files (SignatoryPublicKey.txt and SignatoryPrivateKey.txt). Those values can be used later by Signatory_sign_m to sign multiple messages. Validator can validate those signatures (with an assumption that it doesn't have access to the file containing the private key).

Applications were written in C++ language with usage of libraries: NTL (for multi-precision integer numbers and some number theory routines) and gcrypt (for the hash function). Code was compiled with GCC-C++ v.7.3.1 under Fedora 27 operating system. The source code is available on [1].

5 Example code comparition to specification

Generation of q and p (APPENDIX A.1.1.2):

```
1  void gen_q_and_p(ZZ &q, ZZ &p,
2  ZZ &domainParameterSeed_zz, uint16_t *counter)
3  {
4    do {
5      //obtaining q
6      do {
7        RandomLen(domainParameterSeed_zz, seedlen_bits);
8        BytesFromZZ(domainParameterSeed_str, domainParameterSeed_zz,
9        seedlen_bytes);
10       gcry_md_hash_buffer(chosenHashFunction, U_str,
11       domainParameterSeed_str, seedlen_bytes);
12       ZZFromBytes(U_zz, U_str, outlen_bytes);
13       rem(U_zz, U_zz, power2_ZZ(N-1));
14       q = power(ZZ(2), N - 1) + U_zz + 1 - rem(U_zz, 2);
15       if(NumBits(q) != N) continue;
16       flag_q = ProbPrime(q, MR_iterations);
17     } while(!flag_q);
18
19     //obtaining p
20     ZZ V_zz[n + 1], W = ZZ(0), X = ZZ(0), c = ZZ(0);
21     byte **V_str = (byte **)malloc(sizeof *V_str * (n + 1));
22     for(uint16_t i = 0; i <= n; i++) V_str[i] =
23     (byte *)malloc(sizeof *V_str[i] * outlen_bytes);
24
25     offset = 1;
26     for(*counter = 0; *counter < 4 * L; (*counter)++) {
27       for(uint16_t j = 0; j <= n; j++) {
28         rem(tmp_zz, domainParameterSeed_zz + offset + j,
29         power2_ZZ(seedlen_bits));
30         BytesFromZZ(tmp_str, tmp_zz, seedlen_bytes);
31         gcry_md_hash_buffer(chosenHashFunction, V_str[j],
32         tmp_str, seedlen_bytes);
33         ZZFromBytes(V_zz[j], V_str[j], outlen_bytes);
34       }
35       for(uint16_t j = 0; j <= n; j++)
36         add(W, W, MulMod(V_zz[j], power2_ZZ(j * outlen_bits),
37         power2_ZZ(b)));
38       add(X, W, power2_ZZ(L-1));
39       rem(c, X, 2 * q);
40       sub(p, X, c - 1);
41       if(NumBits(p) != L) {
42         offset += n + 1;
43         continue;
44       }
45       flag_p = ProbPrime(p, MR_iterations);
46       if(flag_p) break;
47       offset += n + 1;
48     }
49   } while(!flag_p);
50 }
```

Examples:
line 14 corresponds to step 7: $q = 2^{N-1} + U + 1 - (U mod 2)$
line 38 corresponds to step 11.3: $X = W + 2^{L-1}$
lines 10-13 correspond to step 6: $U = Hash(domain\_parameter\_seed)\, mod\, 2^{N-1})$

6 Summary

Validator seems to correctly mark signatures as VALID each time they are generated by Signatory_sign_m, while as INVALID each time I mess with message, signature or public key file. Everything seems to run fast and smooth, even with highest standard values for L and N (3072 and 256). In the future I plan to rewrite the code using GNU-GMP instead of NTL, allowing me to compile it as a C code with GCC. I'll measure execution times for generating parameters, signing message, validating signatures and compare them to find out how much faster it would run.

References

[1] `https://github.com/GaloisField94/DSA-studies-`.

[2] FIPS PUB 186-4. `https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf`.

[3] S. A. Vanstone A. J. Menezes, P. C. van Oorschot. Handbook of applied cryptography.