Independent Submission
Request for Comments: 6979
Category: Informational
ISSN: 2070-1721

T. Pornin

August 2013

Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)

Abstract

This document defines a deterministic digital signature generation procedure. Such signatures are compatible with standard Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) digital signatures and can be processed with unmodified verifiers, which need not be aware of the procedure described therein. Deterministic signatures retain the cryptographic security features associated with digital signatures but can be more easily implemented in various environments, since they do not need access to a source of high-quality randomness.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6979.

Table of Contents

1. Introduction

DSA [FIPS-186-4] and ECDSA [X9.62] are two standard digital signature schemes. They provide data integrity and verifiable authenticity in various protocols.

One characteristic of DSA and ECDSA is that they need to produce, for each signature generation, a fresh random value (hereafter designated as k). For effective security, k must be chosen randomly and uniformly from a set of modular integers, using a cryptographically secure process. Even slight biases in that process may be turned into attacks on the signature schemes.

The need for a cryptographically secure source of randomness proves to be a hindrance to deployment of DSA and ECDSA signature schemes in some architectures in which secure random number generation is challenging, in particular, embedded systems such as smartcards. In those systems, the RSA signature algorithm, used as specified in Public-Key Cryptography Standards (PKCS) #1 [RFC3447] (with "type 1" padding, not the Probabilistic Signature Scheme (PSS)) and ISO 9796-2 [ISO-9796-2], is often preferred, even though it is computationally more expensive, because RSA (with such padding schemes) is deterministic and thus does not require a source of randomness.

The randomized nature of DSA and ECDSA also makes implementations harder to test. Automatic tests cannot reliably detect whether the implementation uses a source of randomness of high enough quality. This makes the implementation process more vulnerable to catastrophic failures, often discovered after the system has been deployed and successfully attacked.

It is possible to turn DSA and ECDSA into deterministic schemes by using a deterministic process for generating the "random" value k. That process must fulfill some cryptographic characteristics in order to maintain the properties of verifiability and unforgeability expected from signature schemes; namely, for whoever does not know the signature private key, the mapping from input messages to the corresponding k values must be computationally indistinguishable from what a randomly and uniformly chosen function (from the set of messages to the set of possible k values) would return.

This document describes such a procedure. It has the following features:

- Produced signatures remain fully compatible with plain DSA and ECDSA. Entities that verify the signatures need not be changed or even be aware of the process used to generate k.
- Key pair generation is not altered. Existing private keys can be used with deterministic DSA and ECDSA.
- Using deterministic DSA and ECDSA implies no extra storage requirement of any secret or public value.
- Deterministic DSA and ECDSA can be applied over the same inputs as plain DSA and ECDSA, namely a hash value computed over the message that is to be signed, with a cryptographically secure hash function.

Some relatively arbitrary choices were taken in the definition of deterministic (EC)DSA as specified in this document; this was done in order to make it as universally applicable as possible, so as to maximize usefulness of included test vectors. See Section 3.6 for a discussion of some possible variants.

It shall be noted that key pair generation still requires a source of randomness. In embedded systems where quality of randomness is an issue, it can often be arranged that key pair generation occurs within more controlled conditions (e.g., during a special smartcard initialization procedure or under physical control of sworn agents) or the key might even be generated elsewhere and imported in the device. Deterministic DSA and ECDSA only deal with the need for randomness at the time of signature generation.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. DSA and ECDSA Notations

In this section, we succinctly describe DSA and ECDSA and define our notations. The complete specifications for DSA and ECDSA can be found in [FIPS-186-4] and [X9.62], respectively.

2.1. Key Parameters

DSA and ECDSA work over a large group of prime size, in which the group operation is easy to compute, but the discrete logarithm is computationally infeasible with existing and foreseeable technology. The definition of the group is called the "key parameters". Key parameters may be shared between different key pairs with no ill effect on security; this is the usual case with ECDSA in particular.

DSA uses the following key parameters:

p a large prime number (at least 1024 bits)

q a sufficiently large prime number (at least 160 bits) that is also a divisor of p-1

g a generator for the multiplicative subgroup of order q of integers modulo p

The group on which DSA will be computed consists of the values 'g^j mod p', where '^' denotes exponentiation and j ranges from 0 to q-1 (inclusive). The size of the group is q.

ECDSA uses the following key parameters:

E an elliptic curve, defined over a given finite field

q a sufficiently large prime number (at least 160 bits) that is a divisor of the curve order

G a point of E, of order q

The group on which ECDSA will be computed consists of the curve points jG (multiplication of point G by integer j) where j ranges from 0 to q-1. G is such that qG = 0 (the "point at infinity" on the curve E). The size of the group is q. Note that these notations slightly differ from those described in [X9.62]; we use them in order to match those used for DSA.

2.2. Key Pairs

A DSA or ECDSA private key is an integer x taken modulo q. The relevant standards prescribe that x shall not be 0; hence, x is an integer in the range [1, q-1].

A DSA or ECDSA public key is computed from the private key x and the key parameters:

- For DSA, the public key is the integer: y = g^x mod p
- For ECDSA, the public key is the curve point: U = xG

2.3. Integer Conversions

Let qlen be the binary length of q. qlen is the smallest integer such that q is less than 2^qlen. This is the size of the binary representation of q without a sign bit (note that q, being a big prime, is odd, thus avoiding any ambiguity about the length of any integer equal to a power of 2). We define five conversion functions, which work on strings of bits, octets, and integers modulo q. qlen is the main parameter for these conversions.

In the following subsections, we use two other lengths, called blen and rlen. rlen is equal to qlen, rounded up to the next multiple of 8 (if qlen is already a multiple of 8, then rlen equals qlen; otherwise, rlen is slightly larger, up to qlen+7). Note that rlen is unrelated to the value r, the first half of a generated signature. blen is the length (in bits) of an input sequence of bits and may vary between calls. blen may be smaller than, equal to, or larger than qlen.

2.3.1. Bits and Octets

Formally, all operations are defined on sequences of bits. A sequence is ordered; the first bit is said to be leftmost, while the last bit is rightmost.

On most software systems, bits are grouped into octets (sequences of eight bits). Binary data, e.g., the output of a hash function, is available as a sequence of octets. Whenever applicable, we consider that bits within an octet are ordered from most significant to least significant: the first (leftmost) bit within an octet has numerical value 128, while the last (rightmost) has numerical value 1.

2.3.2. Bit String to Integer

The bits2int transform takes as input a sequence of blen bits and outputs a non-negative integer that is less than 2^qlen. It consists of the following steps:

1. The sequence is first truncated or expanded to length qlen:

   * if qlen < blen, then the qlen leftmost bits are kept, and subsequent bits are discarded;

   * otherwise, qlen-blen bits (of value zero) are added to the left of the sequence (i.e., before the input bits in the sequence order).

2. The resulting sequence is then converted to an integer value using the big-endian convention: if input bits are called b_0 (leftmost) to b_(qlen-1) (rightmost), then the resulting value is:

   b_0*2^(qlen-1) + b_1*2^(qlen-2) + ... + b_(qlen-1)*2^0

The bits2int transform can also be described in the following way: the input bit sequence (of length blen) is transformed into an integer using the big-endian convention. Then, if blen is greater than qlen, the resulting integer is divided by two to the power blen-qlen (Euclidian division: the remainder is discarded); in many software implementations of arithmetics on big integers, that division is equivalent to a "right shift" by blen-qlen bits.

2.3.3. Integer to Octet String

An integer value x less than q (and, in particular, a value that has been taken modulo q) can be converted into a sequence of rlen bits, where rlen = 8*ceil(qlen/8). This is the sequence of bits obtained by big-endian encoding. In other words, the sequence bits x_i (for i ranging from 0 to rlen-1) are such that:

x = x_0*2^(rlen-1) + x_1*2^(rlen-2) + ... + x_(rlen-1)

We call this transform int2octets. Since rlen is a multiple of 8 (the smallest multiple of 8 that is not smaller than qlen), then the resulting sequence of bits is also a sequence of octets, hence the name.

2.3.4. Bit String to Octet String

The bits2octets transform takes as input a sequence of blen bits and outputs a sequence of rlen bits. It consists of the following steps:

1. The input sequence b is converted into an integer value z1 through the bits2int transform:

   z1 = bits2int(b)

2. z1 is reduced modulo q, yielding z2 (an integer between 0 and q-1, inclusive):

   z2 = z1 mod q

   Note that since z1 is less than 2^qlen, that modular reduction can be implemented with a simple conditional subtraction: z2 = z1-q if that value is non-negative; otherwise, z2 = z1.

3. z2 is transformed into a sequence of octets (a sequence of rlen bits) by applying int2octets.

2.3.5. Usage

It is worth noting that int2octets is not the reverse of bits2int, even for input sequences of length qlen: int2octets will add some bits on the left, while bits2int will discard some bits on the right. int2octets is the reverse of bits2int only when qlen is a multiple of 8 and bit sequences already have length qlen.

bits2int is used during signature generation and verification in standard DSA and ECDSA to transform a hash value (computed over the input message) into an integer modulo q. That is, the integer obtained through bits2int is further reduced modulo q; since that integer is less than 2^qlen, that reduction can be performed with at most one subtraction.

int2octets is defined under the name "Integer-to-OctetString" in Section 2.3.7 of SEC 1 [SEC1]. It is used in the specification of the encoding of an ECDSA private key (x) within an ASN.1-based structure.

bits2octets is not used in standard DSA or ECDSA. We will use it in the specification of deterministic (EC)DSA.

2.4. Signature Generation

Signature generation uses a cryptographic hash function H and an input message m. The message is first processed by H, yielding the value H(m), which is a sequence of bits of length hlen. Normally, H is chosen such that its output length hlen is roughly equal to qlen, since the overall security of the signature scheme will depend on the smallest of hlen and qlen; however, the relevant standards support all combinations of hlen and qlen.

The following steps are then applied:

1. H(m) is transformed into an integer modulo q using the bits2int transform and an extra modular reduction:

   h = bits2int(H(m)) mod q

   As was noted in the description of bits2octets, the extra modular reduction is no more than a conditional subtraction.

2. A random value modulo q, dubbed k, is generated. That value shall not be 0; hence, it lies in the [1, q-1] range. Most of the remainder of this document will revolve around the process used to generate k. In plain DSA or ECDSA, k should be selected through a random selection that chooses a value among the q-1 possible values with uniform probability.

3. A value r (modulo q) is computed from k and the key parameters:

   * For DSA:

     r = g^k mod p mod q

     (The exponentiation is performed modulo p, yielding a number between 0 and p-1, which is then further reduced modulo q.)

   * For ECDSA: the point kG is computed; its X coordinate (a member of the field over which E is defined) is converted to an integer, which is reduced modulo q, yielding r.

   If r turns out to be zero, a new k should be selected and r computed again (this is an utterly improbable occurrence).

4. The value s (modulo q) is computed:

   s = (h+x*r)/k mod q

   The pair (r, s) is the signature. How a signature is to be encoded is not covered by the DSA and ECDSA standards themselves; a common way is to use a DER-encoded ASN.1 structure (a SEQUENCE of two INTEGERs, for r and s, in that order).

3. Deterministic DSA and ECDSA

Deterministic (EC)DSA is the process of generating an (EC)DSA signature over an input message m by using the standard (EC)DSA signature generation process (discussed in the previous section), except that the value k, instead of being randomly generated, is obtained through the process described in this section.

We use the notations described in Section 2.

3.1. Building Blocks

3.1.1. HMAC

HMAC [RFC2104] is a construction of a Message Authentication Code using a hash function and a secret key. Here, we use HMAC with the same hash function H as the one used to process the input message prior to signature generation or verification.

We denote the process of applying HMAC with key K over data V by:

```
HMAC_K(V)
```

which returns a sequence of bits of length hlen (the output length of the underlying hash function H).

3.2. Generation of k

Given the input message m, the following process is applied:

a. Process m through the hash function H, yielding:

```
h1 = H(m)
```

(h1 is a sequence of hlen bits).

b. Set:

```
V = 0x01 0x01 0x01 ... 0x01
```

such that the length of V, in bits, is equal to 8*ceil(hlen/8). For instance, on an octet-based system, if H is SHA-256, then V is set to a sequence of 32 octets of value 1. Note that in this step and all subsequent steps, we use the same H function as the one used in step 'a' to process the input message; this choice will be discussed in more detail in Section 3.6.

c. Set:

   K = 0x00 0x00 0x00 ... 0x00

   such that the length of K, in bits, is equal to 8*ceil(hlen/8).

d. Set:

   K = HMAC_K(V || 0x00 || int2octets(x) || bits2octets(h1))

   where '||' denotes concatenation. In other words, we compute HMAC with key K, over the concatenation of the following, in order: the current value of V, a sequence of eight bits of value 0, the encoding of the (EC)DSA private key x, and the hashed message (possibly truncated and extended as specified by the bits2octets transform). The HMAC result is the new value of K. Note that the private key x is in the [1, q-1] range, hence a proper input for int2octets, yielding rlen bits of output, i.e., an integral number of octets (rlen is a multiple of 8).

e. Set:

   V = HMAC_K(V)

f. Set:

   K = HMAC_K(V || 0x01 || int2octets(x) || bits2octets(h1))

   Note that the "internal octet" is 0x01 this time.

g. Set:

   V = HMAC_K(V)

h. Apply the following algorithm until a proper value is found for k:

   1. Set T to the empty sequence. The length of T (in bits) is denoted tlen; thus, at that point, tlen = 0.

   2. While tlen < qlen, do the following:

      V = HMAC_K(V)

      T = T || V

3. Compute:

   k = bits2int(T)

   If that value of k is within the [1,q-1] range, and is suitable for DSA or ECDSA (i.e., it results in an r value that is not 0; see Section 3.4), then the generation of k is finished. The obtained value of k is used in DSA or ECDSA. Otherwise, compute:

   K = HMAC_K(V || 0x00)

   V = HMAC_K(V)

   and loop (try to generate a new T, and so on).

Please note that when k is generated from T, the result of bits2int is compared to q, not reduced modulo q. If the value is not between 1 and q-1, the process loops. Performing a simple modular reduction would induce biases that would be detrimental to signature security.

3.3. Alternate Description of the Generation of k

The process described in the previous section is actually derived from the "HMAC_DRBG" pseudorandom number generator, described in [SP800-90A] and Annex D of [X9.62]. Using the terminology from [SP800-90A], the generation of k can be described as such:

a. Instantiate HMAC_DRBG using HMAC parameterized with the same hash function H as the one used for processing the message that is to be signed. Instantiation parameters are:

   requested_instantiation_security_strength
      Set this parameter to any value that the HMAC_DRBG implementation will accept, when using H as base hash function.

   prediction_resistance_flag
      Set this parameter to "false".

   personalization_string
      Set this parameter to "Null" (the empty bit sequence).

   entropy_input
      Use int2octets(x) as entropy string.

   nonce
      Use bits2octets(H(m)) as nonce.

Note that the last two parameters are not parameters to the HMAC_DRBG instantiation function per se; instead, those values are requested from the internal Get_entropy_input function during instantiation. For deterministic (EC)DSA, we want HMAC_DRBG to run with the entropy string and nonce that we specify, without accessing an actual entropy source.

b. Generate a candidate value for k by requesting qlen bits from HMAC_DRBG and converting the resulting bits into an integer with the bits2int transform. Repeat this step until a value is obtained, which is non-zero, less than q, and suitable for (EC)DSA (see Section 3.4).

Note that we instantiate a new HMAC_DRBG instance for each signature generation process. There is no "personalization string" and no "additional input" when generating bits. The reseed function of HMAC_DRBG is never invoked, neither externally nor as a consequence of the internal HMAC_DRBG processing.

As shown above, we use the encoding of the private key as "entropy string" and the hashed message (truncated and expanded by bits2octets) as "nonce". In HMAC_DRBG, the entropy string and nonce are simply concatenated into the initial seed; hence, the split between "entropy" and "nonce" is quite arbitrary. Using qlen bits for each ought to be compatible with most HMAC_DRBG implementation input requirements.

3.4. Usage Notes

With DSA or ECDSA, the value k is used to compute the first half of the signature, dubbed r (see Section 2.4). The DSA and ECDSA standards mandate that, if r is zero, then a new k should be selected. In that situation, this document specifies that the value k is "unsuitable", and the generation process shall keep on looping.

This occurrence is utterly improbable. Actually, it would require considerable computational effort (similar to breaking preimage resistance of the hash function) to find a private key and a message that lead to a zero value for r; hitting such a case by pure chance is thus deemed implausible, and an attacker cannot force it with carefully crafted messages. In practice, such a code path will not be triggered and thus can be implemented with little optimization.

3.5. Rationale

The process described in the previous sections mimics the "Approved" generation process of k described in Annex D of [X9.62], with the "HMAC_DRBG" pseudorandom number generator. The main difference is

that we use the concatenation of the private key x and the hashed message H(m) as the pseudorandom number generator (PRNG) seed. If using a "security level" of n bits, then HMAC_DRBG should be used with seed entropy at least n+64 bits; however, the key x should also have been generated with that much entropy, and the length of x is qlen, which is at least equal to 2*n and thus larger than n+64 (DSA and ECDSA, as specified by the standards, require qlen >= 160). It can then be argued that deterministic ECDSA fulfills the entropy requirements of Annex D of [X9.62].

We use bits2octets(H(m)) instead of H(m) in order to ease integration. Indeed, many existing signature systems offload the message hashing; the signature engine (which has access to the private key) receives only H(m). In some applications, where data bandwidth is constrained, only the first qlen bits of H(m) are transferred to the signature engine, on the basis that the bits2int transform will ignore subsequent bits anyway. Possibly, in some systems, the truncated H(m) could be externally reduced modulo q, since that is the first thing that (EC)DSA performs on the hashed message. With the definition of bits2octets, deterministic (EC)DSA can be applied with the same input.

3.6. Variants

Many parts of the specification of deterministic (EC)DSA are quite arbitrary. It is possible to define variants that are NOT "deterministic (EC)DSA" but that may nonetheless be useful in some contexts:

- It is possible to use H(m) directly, instead of bits2octets(H(m)), as part of the HMAC input. As explained in Section 3.5, we use bits2octets(H(m)) in order to ease integration into systems that already use an (EC)DSA signature engine by sending it an already-truncated hash value. Using the whole H(m) does not introduce any vulnerability.

- Additional data may be added to the input of HMAC, concatenated after bits2octets(H(m)):

  K = HMAC_K(V || 0x00 || int2octets(x) || bits2octets(h1) || k’)

  A use case may be a protocol that requires a non-deterministic signature algorithm on a system that does not have access to a high-quality random source. It suffices that the additional data k’ is non-repeating (e.g., a signature counter or a monotonic clock) to ensure "random-looking" signatures are indistinguishable, in a cryptographic way, from plain (EC)DSA signatures. In [SP800-90A] terminology, k’ is the "additional

input" that can be set as a parameter when generating pseudorandom bits. This variant can be thought of as a "strengthening" of the randomness of the source of the additional data k’.

- Instead of using x (the private key) as input to HMAC, it is possible to use additional secret data, stored along with the private key with the same security measures. The entropy of that additional data SHALL be at least n bits, preferably n+64 bits or more, where n is the target security level. Having additional secret data may help in formally proving the security of derandomization, but it implies an extra storage cost and incompatibility with already-generated (EC)DSA private keys.

- Similarly, the private key could be a value z, from which both x (the "private key" in the plain (EC)DSA sense) and another value x’, to be used as input to HMAC in the generation of k, would be derived through a suitable Pseudorandom Function (PRF) (such as HMAC_DRBG). This would keep private key storage requirements to a minimum while providing a more easily proven security, but it would impact private key generation and would not be compatible with already-generated key pairs.

- In this document, we use the same hash function H for processing the input message and as a parameter to HMAC. Two distinct hash functions could be used, provided that both are adequately secure. The overall security will be limited by the weaker of the two hash functions, i.e., the one with the smaller output. Using a specific, constant hash function for HMAC may be useful for constrained implementations that accept externally hashed messages, regardless of what hash function was used for that, but have resources for implementing only one hash function for HMAC.

The main disadvantage of any variant is that it ceases to be verifiable against the test vectors published in this document.

4. Security Considerations

Proper implementation and usage of a cryptographic signature algorithm require taking into account many parameters. In particular, private key generation, storage, access control, and disposal are sensitive operations, which this document does not address in any way. Deterministic (EC)DSA shows how to achieve the security characteristics of a standard DSA or ECDSA signature scheme while removing the need for a source of strong randomness, or even any source of randomness, during signature generation.

Private key generation, however, absolutely requires such a strongly random source. In situations where deterministic (EC)DSA is to be used due to the lack of an appropriate source of randomness, one must assume that the private key has been generated externally and imported into the signature generation system or was generated in a context where randomness was available. For instance, one can imagine a smartcard that generates its private key while still in the factory under controlled environmental conditions, but for which random data generation cannot be guaranteed once deployed in the field, when physically in the hands of potential attackers.

Both removal of the random source requirement and the ability to test an implementation against test vectors enhance security of DSA and ECDSA signer implementations, in that they help avoid hard-to-test failure conditions. Deterministic signature schemes may also help in other situations, e.g., to avoid spurious duplicates, when the same data element is signed several times with the same key: with a deterministic signature scheme, the same signature is generated every time, making duplicate detection much easier.

Conversely, lack of randomization may have adverse effects in some advanced protocols, e.g., related to anonymity in some voting schemes. As a rule of thumb, deterministic DSA or ECDSA can be used in lieu of the genuine DSA or ECDSA, with no additional security issues, if the overall protocol would tolerate another deterministic signature scheme, in particular RSA as specified in PKCS #1 [RFC3447] (with "type 1" padding, not PSS) or ISO 9796-2 [ISO-9796-2]. The list of protocols in which deterministic DSA or ECDSA is appropriate includes Transport Layer Security (TLS) [RFC5246], the Secure SHell (SSH) Protocol [RFC4251], Cryptographic Message Syntax (CMS) [RFC5652] and derivatives, X.509 public key infrastructures [RFC5280], and many others.

The construction described in this document is known as a "derandomization". This has been proposed for various signature schemes. Security relies on whether the generation of k is indistinguishable from the output of a random oracle. Roughly speaking, HMAC_DRBG is secure in that role as long as HMAC behaves as a PRF (Pseudorandom Function). For details on the security of HMAC and HMAC_DRBG, please refer to [H2008] and [B2006]. For a more formal treatment of derandomization, see [LN2009].

One remaining issue with deterministic (EC)DSA, as presented in this document, is the "double use" of the private key x, both as the private key in the signature generation algorithm itself and as input to the HMAC_DRBG-based pseudorandom oracle for producing the k value. This requires HMAC_DRBG to keep on being a random oracle, even when

the public key (which is computed from x) is also known. Given the lack of common structure between HMAC and discrete logarithms, this seems a reasonable assumption.

Side-channel attacks are an important consideration whenever an attacker can accurately measure aspects of an implementation such as the length of time that it takes to perform a signing operation or the power consumed at each point of a signing operation. The determinism of the algorithms described in this note may be useful to an attacker in some forms of side-channel attacks, so implementations SHOULD use defensive measures to avoid leaking the private key through a side channel.

5. Intellectual Property Status

To the best of our knowledge, deterministic (EC)DSA is not covered by any active patent. The paper [BDLSY2011] points to two independent publications of the idea of derandomization by Barwood and Wigley, both in early 1997, and also to a patent application by Naccache, M'Raihi, and Levy-dit-Vehel a few months later [NML1997], but the application was withdrawn in 2003. We are not aware of any other patent on the subject.

6. References

6.1. Normative References

[FIPS-186-4] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication (FIPS PUB) 186-4, July 2013.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography (Version 2.0)", May 2009.

[SP800-90A] National Institute of Standards and Technology, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)", NIST Special Publication 800-90A, January 2012.

[X9.62] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-2005, November 2005.

6.2. Informative References

[B2006] Bellare, M., "New Proofs for NMAC and HMAC: Security without Collision-Resistance", Crypto 2006, LNCS 4117, August 2006.

[BDLSY2011] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", Cryptology ePrint Archive Report 2011/368, September 2011.

[FIPS-180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", Federal Information Processing Standards Publication (FIPS PUB) 180-4, March 2012.

[H2008] Hirose, S., "Security Analysis of DRBG Using HMAC in NIST SP 800-90", Information Security Applications (WISA 2008), LNCS 5379, September 2008.

[ISO-9796-2] International Organization for Standardization, "Information technology -- Security techniques -- Digital signature schemes giving message recovery -- Part 2: Integer factorization based mechanisms", ISO/IEC 9796-2:2010, December 2010.

[LN2009] Leurent, G. and P. Nguyen, "How Risky is the Random-Oracle Model?", Cryptology ePrint Archive Report 2008/441, July 2009, <http://eprint.iacr.org/2008/441>.

[NML1997] Naccache, D., M'Raihi, D., and F. Levy-dit-Vehel, "PSEUDO-RANDOM GENERATOR BASED ON A HASH CODING FUNCTION FOR CRYPTOGRAPHIC SYSTEMS REQUIRING RANDOM DRAWING", WIPO patent publication WO/1998/051038, May 1998.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC4251] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

Appendix A. Examples

A.1. Detailed Example

We detail here the intermediate values obtained during the generation of k on an example message and key. We use a binary curve because that specific curve is standard and has a group order length (qlen) that is not a multiple of 8; this illustrates the fine details of how conversions are performed between integers and bit sequences.

A.1.1. Key Pair

We consider ECDSA on the curve K-163 described in [FIPS-186-4] (also known as "ansix9t163k1" in [X9.62]). The curve is defined over a field GF(2^163): field elements are encoded into 163-bit strings. The order of the conventional base point is the prime value:

q = 0x4000000000000000000020108A2E0CC0D99F8A5EF

which has length qlen = 163 bits.

Our private key is:

x = 0x09A4D6792295A7F730FC3F2B49CBC0F62E862272F

The corresponding public key is the curve point U = xG. This point has two coordinates, which are elements of the field GF(2^163). These elements can be converted to integers using the procedure described in Section A.5.6 of [X9.62], yielding the two public point coordinates:

Ux = 0x79AEE090DB05EC252D5CB4452F356BE198A4FF96F

Uy = 0x782E29634DDC9A31EF40386E896BAA18B53AFA5A3

A.1.2. Generation of k

In this example, we use the hash function SHA-256 [FIPS-180-4]. The input message is the UTF-8 encoding of the string "sample" (6 octets, i.e., 48 bits).

The hashed input message h1 = SHA-256(m) is:

h1
AF 2B DB E1 AA 9B 6E C1 E2 AD E1 D6 94 F4 1F C7
1A 83 1D 02 68 E9 89 15 62 11 3D 8A 62 AD D1 BF

(32 octets; each octet value is listed in hexadecimal notation).

We convert the private key x to a sequence of octets using the int2octets transform:

```
int2octets(x)
   00 9A 4D 67 92 29 5A 7F 73 0F C3 F2 B4 9C BC 0F
   62 E8 62 27 2F
```

Note: Although the specific value of x would numerically fit in 160 bits, i.e., 20 octets, we still encode x into 21 octets, because the encoding length is driven by the length of q, which is 163 bits.

We also truncate and/or expand the hashed message using bits2octets:

```
bits2octets(h1)
   01 79 5E DF 0D 54 DB 76 0F 15 6D 0D AC 04 C0 32
   2B 3A 20 42 24
```

The steps b to g (see Section 3.2) then compute the values for the K and V variables. These variables are sequences of 256 bits (the hash function output length, rounded up to a multiple of 8). We reproduce here the successive values:

```
V after step b:
   01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
   01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01

K after step c:
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

K after step d:
   09 99 9A 9B FE F9 72 D3 34 69 11 88 3F AD 79 51
   D2 3F 2C 8B 47 F4 20 22 2D 11 71 EE EE AC 5A B8

V after step e:
   D5 F4 03 0F 75 5E E8 6A A1 0B BA 8C 09 DF 11 4F
   F6 B6 11 1C 23 85 00 D1 3C 73 43 A8 C0 1B EC F7

K after step f:
   0C F2 FE 96 D5 61 9C 9E F5 3C B7 41 7D 49 D3 7E
   A6 8A 4F FE D0 D7 E6 23 E3 86 89 28 99 11 BD 57

V after step g:
   78 34 57 C1 CF 31 48 A8 F2 A9 AE 73 ED 47 2F A9
   8E D9 CD 92 5D 8E 96 4C E0 76 4D EF 3F 84 2B 9A
```

In step h, we perform the final loop. Since we use HMAC with SHA-256, which produces 256 bits worth of output, and we need only 163 bits for T, a single HMAC invocation yields the following T:

```
T (first try)
   93 05 A4 6D E7 FF 8E B1 07 19 4D EB D3 FD 48 AA
   20 D5 E7 65 6C BE 0E A6 9D 2A 8D 4E 7C 67 31 4A
```

which, when converted to an integer with bits2int, yields a first candidate for k:

```
   k1 = 0x4982D236F3FFC758838CA6F5E9FEA455106AF3B2B
```

Since that value is greater than q-1, we have to loop. This first entails computing new values for K and V:

```
new K
   75 CB 5C 05 B2 A7 8C 3D 81 DF 12 D7 4D 7B E0 A0
   E9 4A B1 98 15 78 1D 4D 8E 29 02 A7 9D 0A 66 99

new V
   DC B9 CA 12 61 07 A9 C2 7C E7 7B A5 8E A8 71 C8
   C9 12 D8 35 EA DD C3 05 F2 44 5D 88 F6 6C 4C 43
```

then a new T:

```
T (second try)
   C7 0C 78 60 8A 3B 5B E9 28 9B E9 0E F6 E8 1A 9E
   2C 15 16 D5 75 1D 2F 75 F5 00 33 E4 5F 73 BD EB
```

and a new candidate for k:

```
   k2 = 0x63863C30451DADF4944DF4877B740D4F160A8B6AB
```

Since k2 is also greater than q-1, we loop again:

```
new K (2)
   0A 5A 64 B9 9C 05 95 20 10 36 86 CB 6F 36 BC FC
   A7 88 EB 3B CF 69 BA 66 A5 BB 08 0B 05 93 BA 53

new V (2)
   0B 3B 19 68 11 B1 9F 6C 6F 72 9C 43 F3 5B CF 0D
   FD 72 5F 17 CA 34 30 E8 72 14 53 E5 55 50 A1 8F

T (third try)
   47 5E 80 E9 92 14 05 67 FC C3 A5 0D AB 90 FE 84
   BC D7 BB 03 63 8E 9C 46 56 A0 6F 37 F6 50 8A 7C
```

and we finally get an acceptable value for k:

k = 0x23AF4074C90A02B3FE61D286D5C87F425E6BDD81B

A.1.3. Signature

With our private key and the value of k that we just generated, we can now compute the signature using the standard ECDSA mechanisms. First, the point kG is computed, and the X coordinate of that point is converted to an integer and then reduced modulo q, yielding the first signature half:

r = 0x113A63990598A3828C407C0F4D2438D990DF99A7F

which we use, together with x (the private key), k (which we computed above), and h = bits2int(h1), to compute the second signature half:

s = 0x1313A2E03F5412DDB296A22E2C455335545672D9F

An ECDSA signature is a pair of integers. In many protocols that require a signature to be a sequence of bits (or octets), it is customary to encode the signature as an ASN.1 SEQUENCE of two INTEGER values, with DER rules. This results in the following 48-octet signature:

```
30 2E 02 15 01 13 A6 39 90 59 8A 38 28 C4 07 C0
F4 D2 43 8D 99 0D F9 9A 7F 02 15 01 31 3A 2E 03
F5 41 2D DB 29 6A 22 E2 C4 55 33 55 45 67 2D 9F
```

A.2. Test Vectors

In the following sections, we give test vectors for various key sizes and hash functions, both for DSA and ECDSA.

All numbers are given in hexadecimal notation. Each signature consists of two integers, named r and s; many implementations will encode those integers into a single ASN.1 structure or with some other encoding convention, which is outside of the scope of this document. We also show the k value used internally.

For every key, we list ten signatures, corresponding to two distinct input messages, and five of the SHA [FIPS-180-4] functions: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. The two input messages are the UTF-8 encoding of the strings "sample" and "test" (without the quotes), of length 48 and 32 bits, respectively.

The ECDSA examples use the standard curves described in [FIPS-186-4].

A.2.1. DSA, 1024 Bits

Key pair:

key parameters:

```
p = 86F5CA03DCFEB225063FF830A0C769B9DD9D6153AD91D7CE27F787C43278B447
    E6533B86B18BED6E8A48B784A14C252C5BE0DBF60B86D6385BD2F12FB763ED88
    73ABFD3F5BA2E0A8C0A59082EAC056935E529DAF7C610467899C77ADEDFC846C
    881870B7B19B2B58F9BE0521A17002E3BDD6B86685EE90B3D9A1B02B782B1779

q = 996F967F6C8E388D9E28D01E205FBA957A5698B1

g = 07B0F92546150B62514BB771E2A0C0CE387F03BDA6C56B505209FF25FD3C133D
    89BBCD97E904E09114D9A7DEFDEADFC9078EA544D2E401AEECC40BB9FBBF78FD
    87995A10A1C27CB7789B594BA7EFB5C4326A9FE59A070E136DB77175464ADCA4
    17BE5DCE2F40D10A46A3A3943F26AB7FD9C0398FF8C76EE0A56826A8A88F1DBD
```

private key:

```
x = 411602CB19A6CCC34494D79D98EF1E7ED5AF25F7
```

public key:

```
y = 5DF5E01DED31D0297E274E1691C192FE5868FEF9E19A84776454B100CF16F653
    92195A38B90523E2542EE61871C0440CB87C322FC4B4D2EC5E1E7EC766E1BE8D
    4CE935437DC11C3C8FD426338933EBFE739CB3465F4D3668C5E473508253B1E6
    82F65CBDC4FAE93C2EA212390E54905A86E2223170B44EAA7DA5DD9FFCFB7F3B
```

Signatures:

With SHA-1, message = "sample":

```
k = 7BDB6B0FF756E1BB5D53583EF979082F9AD5BD5B
r = 2E1A0C2562B2912CAAF89186FB0F42001585DA55
s = 29EFB6B0AFF2D7A68EB70CA313022253B9A88DF5
```

With SHA-224, message = "sample":

```
k = 562097C06782D60C3037BA7BE104774344687649
r = 4BC3B686AEA70145856814A6F1BB53346F02101E
s = 410697B92295D994D21EDD2F4ADA85566F6F94C1
```

With SHA-256, message = "sample":

```
k = 519BA0546D0C39202A7D34D7DFA5E760B318BCFB
r = 81F2F5850BE5BC123C43F71A3033E9384611C545
s = 4CDD914B65EB6C66A8AAAD27299BEE6B035F5E89
```

```
With SHA-384, message = "sample":
k = 95897CD7BBB944AA932DBC579C1C09EB6FCFC595
r = 07F2108557EE0E3921BC1774F1CA9B410B4CE65A
s = 54DF70456C86FAC10FAB47C1949AB83F2C6F7595

With SHA-512, message = "sample":
k = 09ECE7CA27D0F5A4DD4E556C9DF1D21D28104F8B
r = 16C3491F9B8C3FBBDD5E7A7B667057F0D8EE8E1B
s = 02C36A127A7B89EDBB72E4FFBC71DABC7D4FC69C

With SHA-1, message = "test":
k = 5C842DF4F9E344EE09F056838B42C7A17F4A6433
r = 42AB2052FD43E123F0607F115052A67DCD9C5C77
s = 183916B0230D45B9931491D4C6B0BD2FB4AAF088

With SHA-224, message = "test":
k = 4598B8EFC1A53BC8AECD58D1ABBB0C0C71E67297
r = 6868E9964E36C1689F6037F91F28D5F2C30610F2
s = 49CEC3ACDC83018C5BD2674ECAAD35B8CD22940F

With SHA-256, message = "test":
k = 5A67592E8128E03A417B0484410FB72C0B630E1A
r = 22518C127299B0F6FDC9872B282B9E70D0790812
s = 6837EC18F150D55DE95B5E29BE7AF5D01E4FE160

With SHA-384, message = "test":
k = 220156B761F6CA5E6C9F1B9CF9C24BE25F98CD89
r = 854CF929B58D73C3CBFDC421E8D5430CD6DB5E66
s = 91D0E0F53E22F898D158380676A871A157CDA622

With SHA-512, message = "test":
k = 65D2C2EEB175E370F28C75BFCDC028D22C7DBE9C
r = 8EA47E475BA8AC6F2D821DA3BD212D11A3DEB9A0
s = 7C670C7AD72B6C050C109E1790008097125433E8
```

A.2.2. DSA, 2048 Bits

Key pair:

key parameters:

```
p = 9DB6FB5951B66BB6FE1E140F1D2CE5502374161FD6538DF1648218642F0B5C48
    C8F7A41AADFA187324B87674FA1822B00F1ECF8136943D7C55757264E5A1A44F
    FE012E9936E00C1D3E9310B01C7D179805D3058B2A9F4BB6F9716BFE6117C6B5
    B3CC4D9BE341104AD4A80AD6C94E005F4B993E14F091EB51743BF33050C38DE2
    35567E1B34C3D6A5C0CEAA1A0F368213C3D19843D0B4B09DCB9FC72D39C8DE41
    F1BF14D4BB4563CA28371621CAD3324B6A2D392145BEBFAC748805236F5CA2FE
    92B871CD8F9C36D3292B5509CA8CAA77A2ADFC7BFD77DDA6F71125A7456FEA15
    3E433256A2261C6A06ED3693797E7995FAD5AABBCFBE3EDA2741E375404AE25B

q = F2C3119374CE76C9356990B465374A17F23F9ED35089BD969F61C6DDE9998C1F

g = 5C7FF6B06F8F143FE8288433493E4769C4D988ACE5BE25A0E24809670716C613
    D7B0CEE6932F8FAA7C44D2CB24523DA53FBE4F6EC3595892D1AA58C4328A06C4
    6A15662E7EAA703A1DECF8BBB2D05DBE2EB956C142A338661D10461C0D135472
    085057F3494309FFA73C611F78B32ADBB5740C361C9F35BE90997DB2014E2EF5
    AA61782F52ABEB8BD6432C4DD097BC5423B285DAFB60DC364E8161F4A2A35ACA
    3A10B1C4D203CC76A470A33AFDCBDD92959859ABD8B56E1725252D78EAC66E71
    BA9AE3F1DD2487199874393CD4D832186800654760E1E34C09E4D155179F9EC0
    DC4473F996BDCE6EED1CABED8B6F116F7AD9CF505DF0F998E34AB27514B0FFE7
```

private key:

```
x = 69C7548C21D0DFEA6B9A51C9EAD4E27C33D3B3F180316E5BCAB92C933F0E4DBC
```

public key:

```
y = 667098C654426C78D7F8201EAC6C203EF030D43605032C2F1FA937E5237DBD94
    9F34A0A2564FE126DC8B715C5141802CE0979C8246463C40E6B6BDAA2513FA61
    1728716C2E4FD53BC95B89E69949D96512E873B9C8F8DFD499CC312882561ADE
    CB31F658E934C0C197F2C4D96B05CBAD67381E7B768891E4DA3843D24D94CDFB
    5126E9B8BF21E8358EE0E0A30EF13FD6A664C0DCE3731F7FB49A4845A4FD8254
    687972A2D382599C9BAC4E0ED7998193078913032558134976410B89D2C171D1
    23AC35FD977219597AA7D15C1A9A428E59194F75C721EBCBCFAE44696A499AFA
    74E04299F132026601638CB87AB79190D4A0986315DA8EEC6561C938996BEADF
```

Signatures:

With SHA-1, message = "sample":

```
k = 888FA6F7738A41BDC9846466ABDB8174C0338250AE50CE955CA16230F9CBD53E
r = 3A1B2DBD7489D6ED7E608FD036C83AF396E290DBD602408E8677DAABD6E7445A
s = D26FCBA19FA3E3058FFC02CA1596CDBB6E0D20CB37B06054F7E36DED0CDBBCCF
```

```
With SHA-224, message = "sample":
k = BC372967702082E1AA4FCE892209F71AE4AD25A6DFD869334E6F153BD0C4D806
r = DC9F4DEADA8D8FF588E98FED0AB690FFCE858DC8C79376450EB6B76C24537E2C
s = A65A9C3BC7BABE286B195D5DA68616DA8D47FA0097F36DD19F517327DC848CEC

With SHA-256, message = "sample":
k = 8926A27C40484216F052F4427CFD5647338B7B3939BC6573AF4333569D597C52
r = EACE8BDBBE353C432A795D9EC556C6D021F7A03F42C36E9BC87E4AC7932CC809
s = 7081E175455F9247B812B74583E9E94F9EA79BD640DC962533B0680793A38D53

With SHA-384, message = "sample":
k = C345D5AB3DA0A5BCB7EC8F8FB7A7E96069E03B206371EF7D83E39068EC564920
r = B2DA945E91858834FD9BF616EBAC151EDBC4B45D27D0DD4A7F6A22739F45C00B
s = 19048B63D9FD6BCA1D9BAE3664E1BCB97F7276C306130969F63F38FA8319021B

With SHA-512, message = "sample":
k = 5A12994431785485B3F5F067221517791B85A597B7A9436995C89ED0374668FC
r = 2016ED092DC5FB669B8EFB3D1F31A91EECB199879BE0CF78F02BA062CB4C942E
s = D0C76F84B5F091E141572A639A4FB8C230807EEA7D55C8A154A224400AFF2351

With SHA-1, message = "test":
k = 6EEA486F9D41A037B2C640BC5645694FF8FF4B98D066A25F76BE641CCB24BA4F
r = C18270A93CFC6063F57A4DFA86024F700D980E4CF4E2CB65A504397273D98EA0
s = 414F22E5F31A8B6D33295C7539C1C1BA3A6160D7D68D50AC0D3A5BEAC2884FAA

With SHA-224, message = "test":
k = 06BD4C05ED74719106223BE33F2D95DA6B3B541DAD7BFBD7AC508213B6DA6670
r = 272ABA31572F6CC55E30BF616B7A265312018DD325BE031BE0CC82AA17870EA3
s = E9CC286A52CCE201586722D36D1E917EB96A4EBDB47932F9576AC645B3A60806

With SHA-256, message = "test":
k = 1D6CE6DDA1C5D37307839CD03AB0A5CBB18E60D800937D67DFB4479AAC8DEAD7
r = 8190012A1969F9957D56FCCAAD223186F423398D58EF5B3CEFD5A4146A4476F0
s = 7452A53F7075D417B4B013B278D1BB8BBD21863F5E7B1CEE679CF2188E1AB19E

With SHA-384, message = "test":
k = 206E61F73DBE1B2DC8BE736B22B079E9DACD974DB00EEBBC5B64CAD39CF9F91C
r = 239E66DDBE8F8C230A3D071D601B6FFBDFB5901F94D444C6AF56F732BEB954BE
s = 6BD737513D5E72FE85D1C750E0F73921FE299B945AAD1C802F15C26A43D34961

With SHA-512, message = "test":
k = AFF1651E4CD6036D57AA8B2A05CCF1A9D5A40166340ECBBDC55BE10B568AA0AA
r = 89EC4BB1400ECCFF8E7D9AA515CD1DE7803F2DAFF09693EE7FD1353E90A68307
s = C9F0BDABCC0D880BB137A994CC7F3980CE91CC10FAF529FC46565B15CEA854E1
```

A.2.3. ECDSA, 192 Bits (Prime Field)

Key pair:

curve: NIST P-192

q = FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831
(qlen = 192 bits)

private key:

x = 6FAB034934E4C0FC9AE67F5B5659A9D7D1FEFD187EE09FD4

public key: U = xG

Ux = AC2C77F529F91689FEA0EA5EFEC7F210D8EEA0B9E047ED56

Uy = 3BC723E57670BD4887EBC732C523063D0A7C957BC97C1C43

Signatures:

With SHA-1, message = "sample":
k = 37D7CA00D2C7B0E5E412AC03BD44BA837FDD5B28CD3B0021
r = 98C6BD12B23EAF5E2A2045132086BE3EB8EBD62ABF6698FF
s = 57A22B07DEA9530F8DE9471B1DC6624472E8E2844BC25B64

With SHA-224, message = "sample":
k = 4381526B3FC1E7128F202E194505592F01D5FF4C5AF015D8
r = A1F00DAD97AEEC91C95585F36200C65F3C01812AA60378F5
s = E07EC1304C7C6C9DEBBE980B9692668F81D4DE7922A0F97A

With SHA-256, message = "sample":
k = 32B1B6D7D42A05CB449065727A84804FB1A3E34D8F261496
r = 4B0B8CE98A92866A2820E20AA6B75B56382E0F9BFD5ECB55
s = CCDB006926EA9565CBADC840829D8C384E06DE1F1E381B85

With SHA-384, message = "sample":
k = 4730005C4FCB01834C063A7B6760096DBE284B8252EF4311
r = DA63BF0B9ABCF948FBB1E9167F136145F7A20426DCC287D5
s = C3AA2C960972BD7A2003A57E1C4C77F0578F8AE95E31EC5E

With SHA-512, message = "sample":
k = A2AC7AB055E4F20692D49209544C203A7D1F2C0BFBC75DB1
r = 4D60C5AB1996BD848343B31C00850205E2EA6922DAC2E4B8
s = 3F6E837448F027A1BF4B34E796E32A811CBB4050908D8F67

```
With SHA-1, message = "test":
k = D9CF9C3D3297D3260773A1DA7418DB5537AB8DD93DE7FA25
r = 0F2141A0EBBC44D2E1AF90A50EBCFCE5E197B3B7D4DE036D
s = EB18BC9E1F3D7387500CB99CF5F7C157070A8961E38700B7

With SHA-224, message = "test":
k = F5DC805F76EF851800700CCE82E7B98D8911B7D510059FBE
r = 6945A1C1D1B2206B8145548F633BB61CEF04891BAF26ED34
s = B7FB7FDFC339C0B9BD61A9F5A8EAF9BE58FC5CBA2CB15293

With SHA-256, message = "test":
k = 5C4CE89CF56D9E7C77C8585339B006B97B5F0680B4306C6C
r = 3A718BD8B4926C3B52EE6BBE67EF79B18CB6EB62B1AD97AE
s = 5662E6848A4A19B1F1AE2F72ACD4B8BBE50F1EAC65D9124F

With SHA-384, message = "test":
k = 5AFEFB5D3393261B828DB6C91FBC68C230727B030C975693
r = B234B60B4DB75A733E19280A7A6034BD6B1EE88AF5332367
s = 7994090B2D59BB782BE57E74A44C9A1C700413F8ABEFE77A

With SHA-512, message = "test":
k = 0758753A5254759C7CFBAD2E2D9B0792EEE44136C9480527
r = FE4F4AE86A58B6507946715934FE2D8FF9D95B6B098FE739
s = 74CF5605C98FBA0E1EF34D4B5A1577A7DCF59457CAE52290
```

A.2.4. ECDSA, 224 Bits (Prime Field)

Key pair:

curve: NIST P-224

q = FFFFFFFFFFFFFFFFFFFFFFFFFFFF16A2E0B8F03E13DD29455C5C2A3D
(qlen = 224 bits)

private key:

x = F220266E1105BFE3083E03EC7A3A654651F45E37167E88600BF257C1

public key: U = xG

Ux = 00CF08DA5AD719E42707FA431292DEA11244D64FC51610D94B130D6C

Uy = EEAB6F3DEBE455E3DBF85416F7030CBD94F34F2D6F232C69F3C1385A

Signatures:

With SHA-1, message = "sample":
k = 7EEFADD91110D8DE6C2C470831387C50D3357F7F4D477054B8B426BC
r = 22226F9D40A96E19C4A301CE5B74B115303C0F3A4FD30FC257FB57AC
s = 66D1CDD83E3AF75605DD6E2FEFF196D30AA7ED7A2EDF7AF475403D69

With SHA-224, message = "sample":
k = C1D1F2F10881088301880506805FEB4825FE09ACB6816C36991AA06D
r = 1CDFE6662DDE1E4A1EC4CDEDF6A1F5A2FB7FBD9145C12113E6ABFD3E
s = A6694FD7718A21053F225D3F46197CA699D45006C06F871808F43EBC

With SHA-256, message = "sample":
k = AD3029E0278F80643DE33917CE6908C70A8FF50A411F06E41DEDFCDC
r = 61AA3DA010E8E8406C656BC477A7A7189895E7E840CDFE8FF42307BA
s = BC814050DAB5D23770879494F9E0A680DC1AF7161991BDE692B10101

With SHA-384, message = "sample":
k = 52B40F5A9D3D13040F494E83D3906C6079F29981035C7BD51E5CAC40
r = 0B115E5E36F0F9EC81F1325A5952878D745E19D7BB3EABFABA77E953
s = 830F34CCDFE826CCFDC81EB4129772E20E122348A2BBD889A1B1AF1D

With SHA-512, message = "sample":
k = 9DB103FFEDEDF9CFDBA05184F925400C1653B8501BAB89CEA0FBEC14
r = 074BD1D979D5F32BF958DDC61E4FB4872ADCAFEB2256497CDAC30397
s = A4CECA196C3D5A1FF31027B33185DC8EE43F288B21AB342E5D8EB084

```
With SHA-1, message = "test":
k = 2519178F82C3F0E4F87ED5883A4E114E5B7A6E374043D8EFD329C253
r = DEAA646EC2AF2EA8AD53ED66B2E2DDAA49A12EFD8356561451F3E21C
s = 95987796F6CF2062AB8135271DE56AE55366C045F6D9593F53787BD2

With SHA-224, message = "test":
k = DF8B38D40DCA3E077D0AC520BF56B6D565134D9B5F2EAE0D34900524
r = C441CE8E261DED634E4CF84910E4C5D1D22C5CF3B732BB204DBEF019
s = 902F42847A63BDC5F6046ADA114953120F99442D76510150F372A3F4

With SHA-256, message = "test":
k = FF86F57924DA248D6E44E8154EB69F0AE2AEBAEE9931D0B5A969F904
r = AD04DDE87B84747A243A631EA47A1BA6D1FAA059149AD2440DE6FBA6
s = 178D49B1AE90E3D8B629BE3DB5683915F4E8C99FDF6E666CF37ADCFD

With SHA-384, message = "test":
k = 7046742B839478C1B5BD31DB2E862AD868E1A45C863585B5F22BDC2D
r = 389B92682E399B26518A95506B52C03BC9379A9DADF3391A21FB0EA4
s = 414A718ED3249FF6DBC5B50C27F71F01F070944DA22AB1F78F559AAB

With SHA-512, message = "test":
k = E39C2AA4EA6BE2306C72126D40ED77BF9739BB4D6EF2BBB1DCB6169D
r = 049F050477C5ADD858CAC56208394B5A55BAEBBE887FDF765047C17C
s = 077EB13E7005929CEFA3CD0403C7CDCC077ADF4E44F3C41B2F60ECFF
```

A.2.5. ECDSA, 256 Bits (Prime Field)

Key pair:

curve: NIST P-256

q = FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
(qlen = 256 bits)

private key:

x = C9AFA9D845BA75166B5C215767B1D6934E50C3DB36E89B127B8A622B120F6721

public key: U = xG

Ux = 60FED4BA255A9D31C961EB74C6356D68C049B8923B61FA6CE669622E60F29FB6

Uy = 7903FE1008B8BC99A41AE9E95628BC64F2F1B20C2D7E9F5177A3C294D4462299

Signatures:

With SHA-1, message = "sample":
k = 882905F1227FD620FBF2ABF21244F0BA83D0DC3A9103DBBEE43A1FB858109DB4
r = 61340C88C3AAEBEB4F6D667F672CA9759A6CCAA9FA8811313039EE4A35471D32
s = 6D7F147DAC089441BB2E2FE8F7A3FA264B9C475098FDCF6E00D7C996E1B8B7EB

With SHA-224, message = "sample":
k = 103F90EE9DC52E5E7FB5132B7033C63066D194321491862059967C715985D473
r = 53B2FFF5D1752B2C689DF257C04C40A587FABABB3F6FC2702F1343AF7CA9AA3F
s = B9AFB64FDC03DC1A131C7D2386D11E349F070AA432A4ACC918BEA988BF75C74C

With SHA-256, message = "sample":
k = A6E3C57DD01ABE90086538398355DD4C3B17AA873382B0F24D6129493D8AAD60
r = EFD48B2AACB6A8FD1140DD9CD45E81D69D2C877B56AAF991C34D0EA84EAF3716
s = F7CB1C942D657C41D436C7A1B6E29F65F3E900DBB9AFF4064DC4AB2F843ACDA8

With SHA-384, message = "sample":
k = 09F634B188CEFD98E7EC88B1AA9852D734D0BC272F7D2A47DECC6EBEB375AAD4
r = 0EAFEA039B20E9B42309FB1D89E213057CBF973DC0CFC8F129EDDDC800EF7719
s = 4861F0491E6998B9455193E34E7B0D284DDD7149A74B95B9261F13ABDE940954

With SHA-512, message = "sample":
k = 5FA81C63109BADB88C1F367B47DA606DA28CAD69AA22C4FE6AD7DF73A7173AA5
r = 8496A60B5E9B47C825488827E0495B0E3FA109EC4568FD3F8D1097678EB97F00
s = 2362AB1ADBE2B8ADF9CB9EDAB740EA6049C028114F2460F96554F61FAE3302FE

```
With SHA-1, message = "test":
k = 8C9520267C55D6B980DF741E56B4ADEE114D84FBFA2E62137954164028632A2E
r = 0CBCC86FD6ABD1D99E703E1EC50069EE5C0B4BA4B9AC60E409E8EC5910D81A89
s = 01B9D7B73DFAA60D5651EC4591A0136F87653E0FD780C3B1BC872FFDEAE479B1

With SHA-224, message = "test":
k = 669F4426F2688B8BE0DB3A6BD1989BDAEFFF84B649EEB84F3DD26080F667FAA7
r = C37EDB6F0AE79D47C3C27E962FA269BB4F441770357E114EE511F662EC34A692
s = C820053A05791E521FCAAD6042D40AEA1D6B1A540138558F47D0719800E18F2D

With SHA-256, message = "test":
k = D16B6AE827F17175E040871A1C7EC3500192C4C92677336EC2537ACAEE0008E0
r = F1ABB023518351CD71D881567B1EA663ED3EFCF6C5132B354F28D3B0B7D38367
s = 019F4113742A2B14BD25926B49C649155F267E60D3814B4C0CC84250E46F0083

With SHA-384, message = "test":
k = 16AEFFA357260B04B1DD199693960740066C1A8F3E8EDD79070AA914D361B3B8
r = 83910E8B48BB0C74244EBDF7F07A1C5413D61472BD941EF3920E623FBCCEBEB6
s = 8DDBEC54CF8CD5874883841D712142A56A8D0F218F5003CB0296B6B509619F2C

With SHA-512, message = "test":
k = 6915D11632ACA3C40D5D51C08DAF9C555933819548784480E93499000D9F0B7F
r = 461D93F31B6540894788FD206C07CFA0CC35F46FA3C91816FFF1040AD1581A04
s = 39AF9F15DE0DB8D97E72719C74820D304CE5226E32DEDAE67519E840D1194E55
```

A.2.6. ECDSA, 384 Bits (Prime Field)

Key pair:

curve: NIST P-384

q = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC7634D81F4372DDF
581A0DB248B0A77AECEC196ACCC52973
(qlen = 384 bits)

private key:

x = 6B9D3DAD2E1B8C1C05B19875B6659F4DE23C3B667BF297BA9AA47740787137D8
96D5724E4C70A825F872C9EA60D2EDF5

public key: U = xG

Ux = EC3A4E415B4E19A4568618029F427FA5DA9A8BC4AE92E02E06AAE5286B300C64
DEF8F0EA9055866064A254515480BC13

Uy = 8015D9B72D7D57244EA8EF9AC0C621896708A59367F9DFB9F54CA84B3F1C9DB1
288B231C3AE0D4FE7344FD2533264720

Signatures:

With SHA-1, message = "sample":
k = 4471EF7518BB2C7C20F62EAE1C387AD0C5E8E470995DB4ACF694466E6AB09663
0F29E5938D25106C3C340045A2DB01A7
r = EC748D839243D6FBEF4FC5C4859A7DFFD7F3ABDDF72014540C16D73309834FA3
7B9BA002899F6FDA3A4A9386790D4EB2
s = A3BCFA947BEEF4732BF247AC17F71676CB31A847B9FF0CBC9C9ED4C1A5B3FACF
26F49CA031D4857570CCB5CA4424A443

With SHA-224, message = "sample":
k = A4E4D2F0E729EB786B31FC20AD5D849E304450E0AE8E3E341134A5C1AFA03CAB
8083EE4E3C45B06A5899EA56C51B5879
r = 42356E76B55A6D9B4631C865445DBE54E056D3B3431766D0509244793C3F9366
450F76EE3DE43F5A125333A6BE060122
s = 9DA0C81787064021E78DF658F2FBB0B042BF304665DB721F077A4298B095E483
4C082C03D83028EFBF93A3C23940CA8D

With SHA-256, message = "sample":
k = 180AE9F9AEC5438A44BC159A1FCB277C7BE54FA20E7CF404B490650A8ACC414E
375572342863C899F9F2EDF9747A9B60
r = 21B13D1E013C7FA1392D03C5F99AF8B30C570C6F98D4EA8E354B63A21D3DAA33
BDE1E888E63355D92FA2B3C36D8FB2CD
s = F3AA443FB107745BF4BD77CB3891674632068A10CA67E3D45DB2266FA7D1FEEB
EFDC63ECCD1AC42EC0CB8668A4FA0AB0

```
   With SHA-384, message = "sample":
   k = 94ED910D1A099DAD3254E9242AE85ABDE4BA15168EAF0CA87A555FD56D10FBCA
       2907E3E83BA95368623B8C4686915CF9
   r = 94EDBB92A5ECB8AAD4736E56C691916B3F88140666CE9FA73D64C4EA95AD133C
       81A648152E44ACF96E36DD1E80FABE46
   s = 99EF4AEB15F178CEA1FE40DB2603138F130E740A19624526203B6351D0A3A94F
       A329C145786E679E7B82C71A38628AC8

   With SHA-512, message = "sample":
   k = 92FC3C7183A883E24216D1141F1A8976C5B0DD797DFA597E3D7B32198BD35331
       A4E966532593A52980D0E3AAA5E10EC3
   r = ED0959D5880AB2D869AE7F6C2915C6D60F96507F9CB3E047C0046861DA4A799C
       FE30F35CC900056D7C99CD7882433709
   s = 512C8CCEEE3890A84058CE1E22DBC2198F42323CE8ACA9135329F03C068E5112
       DC7CC3EF3446DEFCEB01A45C2667FDD5

   With SHA-1, message = "test":
   k = 66CC2C8F4D303FC962E5FF6A27BD79F84EC812DDAE58CF5243B64A4AD8094D47
       EC3727F3A3C186C15054492E30698497
   r = 4BC35D3A50EF4E30576F58CD96CE6BF638025EE624004A1F7789A8B8E43D0678
       ACD9D29876DAF46638645F7F404B11C7
   s = D5A6326C494ED3FF614703878961C0FDE7B2C278F9A65FD8C4B7186201A29916
       95BA1C84541327E966FA7B50F7382282

   With SHA-224, message = "test":
   k = 18FA39DB95AA5F561F30FA3591DC59C0FA3653A80DAFFA0B48D1A4C6DFCBFF6E
       3D33BE4DC5EB8886A8ECD093F2935726
   r = E8C9D0B6EA72A0E7837FEA1D14A1A9557F29FAA45D3E7EE888FC5BF954B5E624
       64A9A817C47FF78B8C11066B24080E72
   s = 07041D4A7A0379AC7232FF72E6F77B6DDB8F09B16CCE0EC3286B2BD43FA8C614
       1C53EA5ABEF0D8231077A04540A96B66

   With SHA-256, message = "test":
   k = 0CFAC37587532347DC3389FDC98286BBA8C73807285B184C83E62E26C401C0FA
       A48DD070BA79921A3457ABFF2D630AD7
   r = 6D6DEFAC9AB64DABAFE36C6BF510352A4CC27001263638E5B16D9BB51D451559
       F918EEDAF2293BE5B475CC8F0188636B
   s = 2D46F3BECBCC523D5F1A1256BF0C9B024D879BA9E838144C8BA6BAEB4B53B47D
       51AB373F9845C0514EEFB14024787265

   With SHA-384, message = "test":
   k = 015EE46A5BF88773ED9123A5AB0807962D193719503C527B031B4C2D225092AD
       A71F4A459BC0DA98ADB95837DB8312EA
   r = 8203B63D3C853E8D77227FB377BCF7B7B772E97892A80F36AB775D509D7A5FEB
       0542A7F0812998DA8F1DD3CA3CF023DB
   s = DDD0760448D42D8A43AF45AF836FCE4DE8BE06B485E9B61B827C2F13173923E0
       6A739F040649A667BF3B828246BAA5A5
```

```
With SHA-512, message = "test":
k = 3780C4F67CB15518B6ACAE34C9F83568D2E12E47DEAB6C50A4E4EE5319D1E8CE
    0E2CC8A136036DC4B9C00E6888F66B6C
r = A0D5D090C9980FAF3C2CE57B7AE951D31977DD11C775D314AF55F76C676447D0
    6FB6495CD21B4B6E340FC236584FB277
s = 976984E59B4C77B0E8E4460DCA3D9F20E07B9BB1F63BEEFAF576F6B2E8B22463
    4A2092CD3792E0159AD9CEE37659C736
```

A.2.7. ECDSA, 521 Bits (Prime Field)

Key pair:

curve: NIST P-521

q = 1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFA51868783BF2F966B7FCC0148F709A5D03BB5C9B8899C47AEBB6FB71E91386
409
(qlen = 521 bits)

private key:

x = 0FAD06DAA62BA3B25D2FB40133DA757205DE67F5BB0018FEE8C86E1B68C7E75C
AA896EB32F1F47C70855836A6D16FCC1466F6D8FBEC67DB89EC0C08B0E996B83
538

public key: U = xG

Ux = 1894550D0785932E00EAA23B694F213F8C3121F86DC97A04E5A7167DB4E5BCD3
71123D46E45DB6B5D5370A7F20FB633155D38FFA16D2BD761DCAC474B9A2F502
3A4

Uy = 0493101C962CD4D2FDDF782285E64584139C2F91B47F87FF82354D6630F746A2
8A0DB25741B5B34A828008B22ACC23F924FAAFBD4D33F81EA66956DFEAA2BFDF
CF5

Signatures:

With SHA-1, message = "sample":
k = 089C071B419E1C2820962321787258469511958E80582E95D8378E0C2CCDB3CB
42BEDE42F50E3FA3C71F5A76724281D31D9C89F0F91FC1BE4918DB1C03A5838D
0F9
r = 0343B6EC45728975EA5CBA6659BBB6062A5FF89EEA58BE3C80B619F322C87910
FE092F7D45BB0F8EEE01ED3F20BABEC079D202AE677B243AB40B5431D497C55D
75D
s = 0E7B0E675A9B24413D448B8CC119D2BF7B2D2DF032741C096634D6D65D0DBE3D
5694625FB9E8104D3B842C1B0E2D0B98BEA19341E8676AEF66AE4EBA3D5475D5
D16

```
   With SHA-224, message = "sample":
   k = 121415EC2CD7726330A61F7F3FA5DE14BE9436019C4DB8CB4041F3B54CF31BE0
       493EE3F427FB906393D895A19C9523F3A1D54BB8702BD4AA9C99DAB2597B9211
       3F3
   r = 1776331CFCDF927D666E032E00CF776187BC9FDD8E69D0DABB4109FFE1B5E2A3
       0715F4CC923A4A5E94D2503E9ACFED92857B7F31D7152E0F8C00C15FF3D87E2E
       D2E
   s = 050CB5265417FE2320BBB5A122B8E1A32BD699089851128E360E620A30C7E17B
       A41A666AF126CE100E5799B153B60528D5300D08489CA9178FB610A2006C254B
       41F

   With SHA-256, message = "sample":
   k = 0EDF38AFCAAECAB4383358B34D67C9F2216C8382AAEA44A3DAD5FDC9C3257576
       1793FEF24EB0FC276DFC4F6E3EC476752F043CF01415387470BCBD8678ED2C7E
       1A0
   r = 1511BB4D675114FE266FC4372B87682BAECC01D3CC62CF2303C92B3526012659
       D16876E25C7C1E57648F23B73564D67F61C6F14D527D54972810421E7D87589E
       1A7
   s = 04A171143A83163D6DF460AAF61522695F207A58B95C0644D87E52AA1A347916
       E4F7A72930B1BC06DBE22CE3F58264AFD23704CBB63B29B931F7DE6C9D949A7E
       CFC

   With SHA-384, message = "sample":
   k = 1546A108BC23A15D6F21872F7DED661FA8431DDBD922D0DCDB77CC878C8553FF
       AD064C95A920A750AC9137E527390D2D92F153E66196966EA554D9ADFCB109C4
       211
   r = 1EA842A0E17D2DE4F92C15315C63DDF72685C18195C2BB95E572B9C5136CA4B4
       B576AD712A52BE9730627D16054BA40CC0B8D3FF035B12AE75168397F5D50C67
       451
   s = 1F21A3CEE066E1961025FB048BD5FE2B7924D0CD797BABE0A83B66F1E35EEAF5
       FDE143FA85DC394A7DEE766523393784484BDF3E00114A1C857CDE1AA203DB65
       D61

   With SHA-512, message = "sample":
   k = 1DAE2EA071F8110DC26882D4D5EAE0621A3256FC8847FB9022E2B7D28E6F1019
       8B1574FDD03A9053C08A1854A168AA5A57470EC97DD5CE090124EF52A2F7ECBF
       FD3
   r = 0C328FAFCBD79DD77850370C46325D987CB525569FB63C5D3BC53950E6D4C5F1
       74E25A1EE9017B5D450606ADD152B534931D7D4E8455CC91F9B15BF05EC36E37
       7FA
   s = 0617CCE7CF5064806C467F678D3B4080D6F1CC50AF26CA209417308281B68AF2
       82623EAA63E5B5C0723D8B8C37FF0777B1A20F8CCB1DCCC43997F1EE0E44DA4A
       67A
```

```
   With SHA-1, message = "test":
   k = 0BB9F2BF4FE1038CCF4DABD7139A56F6FD8BB1386561BD3C6A4FC818B20DF5DD
       BA80795A947107A1AB9D12DAA615B1ADE4F7A9DC05E8E6311150F47F5C57CE8B
       222
   r = 13BAD9F29ABE20DE37EBEB823C252CA0F63361284015A3BF430A46AAA80B87B0
       693F0694BD88AFE4E661FC33B094CD3B7963BED5A727ED8BD6A3A202ABE009D0
       367
   s = 1E9BB81FF7944CA409AD138DBBEE228E1AFCC0C890FC78EC8604639CB0DBDC90
       F717A99EAD9D272855D00162EE9527567DD6A92CBD629805C0445282BBC91679
       7FF

   With SHA-224, message = "test":
   k = 040D09FCF3C8A5F62CF4FB223CBBB2B9937F6B0577C27020A99602C25A011369
       87E452988781484EDBBCF1C47E554E7FC901BC3085E5206D9F619CFF07E73D6F
       706
   r = 1C7ED902E123E6815546065A2C4AF977B22AA8EADDB68B2C1110E7EA44D42086
       BFE4A34B67DDC0E17E96536E358219B23A706C6A6E16BA77B65E1C595D43CAE1
       7FB
   s = 177336676304FCB343CE028B38E7B4FBA76C1C1B277DA18CAD2A8478B2A9A9F5
       BEC0F3BA04F35DB3E4263569EC6AADE8C92746E4C82F8299AE1B8F1739F8FD51
       9A4

   With SHA-256, message = "test":
   k = 01DE74955EFAABC4C4F17F8E84D881D1310B5392D7700275F82F145C61E84384
       1AF09035BF7A6210F5A431A6A9E81C9323354A9E69135D44EBD2FCAA7731B909
       258
   r = 00E871C4A14F993C6C7369501900C4BC1E9C7B0B4BA44E04868B30B41D807104
       2EB28C4C250411D0CE08CD197E4188EA4876F279F90B3D8D74A3C76E6F1E4656
       AA8
   s = 0CD52DBAA33B063C3A6CD8058A1FB0A46A4754B034FCC644766CA14DA8CA5CA9
       FDE00E88C1AD60CCBA759025299079D7A427EC3CC5B619BFBC828E7769BCD694
       E86

   With SHA-384, message = "test":
   k = 1F1FC4A349A7DA9A9E116BFDD055DC08E78252FF8E23AC276AC88B1770AE0B5D
       CEB1ED14A4916B769A523CE1E90BA22846AF11DF8B300C38818F713DADD85DE0
       C88
   r = 14BEE21A18B6D8B3C93FAB08D43E739707953244FDBE924FA926D76669E7AC8C
       89DF62ED8975C2D8397A65A49DCC09F6B0AC62272741924D479354D74FF60755
       78C
   s = 133330865C067A0EAF72362A65E2D7BC4E461E8C8995C3B6226A21BD1AA78F0E
       D94FE536A0DCA35534F0CD1510C41525D163FE9D74D134881E35141ED5E8E95B
       979
```

```
   With SHA-512, message = "test":
   k = 16200813020EC986863BEDFC1B121F605C1215645018AEA1A7B215A564DE9EB1
       B38A67AA1128B80CE391C4FB71187654AAA3431027BFC7F395766CA988C964DC
       56D
   r = 13E99020ABF5CEE7525D16B69B229652AB6BDF2AFFCAEF38773B4B7D08725F10
       CDB93482FDCC54EDCEE91ECA4166B2A7C6265EF0CE2BD7051B7CEF945BABD47E
       E6D
   s = 1FBD0013C674AA79CB39849527916CE301C66EA7CE8B80682786AD60F98F7E78
       A19CA69EFF5C57400E3B3A0AD66CE0978214D13BAF4E9AC60752F7B155E2DE4D
       CE3
```

A.2.8. ECDSA, 163 Bits (Binary Field, Koblitz Curve)

Key pair:

curve: NIST K-163

q = 4000000000000000000020108A2E0CC0D99F8A5EF
(qlen = 163 bits)

private key:

x = 09A4D6792295A7F730FC3F2B49CBC0F62E862272F

public key: U = xG

Ux = 79AEE090DB05EC252D5CB4452F356BE198A4FF96F

Uy = 782E29634DDC9A31EF40386E896BAA18B53AFA5A3

Signatures:

With SHA-1, message = "sample":
k = 09744429FA741D12DE2BE8316E35E84DB9E5DF1CD
r = 30C45B80BA0E1406C4EFBBB7000D6DE4FA465D505
s = 38D87DF89493522FC4CD7DE1553BD9DBBA2123011

With SHA-224, message = "sample":
k = 323E7B28BFD64E6082F5B12110AA87BC0D6A6E159
r = 38A2749F7EA13BD5DA0C76C842F512D5A65FFAF32
s = 064F841F70112B793FD773F5606BFA5AC2A04C1E8

With SHA-256, message = "sample":
k = 23AF4074C90A02B3FE61D286D5C87F425E6BDD81B
r = 113A63990598A3828C407C0F4D2438D990DF99A7F
s = 1313A2E03F5412DDB296A22E2C455335545672D9F

With SHA-384, message = "sample":
k = 2132ABE0ED518487D3E4FA7FD24F8BED1F29CCFCE
r = 34D4DE955871BB84FEA4E7D068BA5E9A11BD8B6C4
s = 2BAAF4D4FD57F175C405A2F39F9755D9045C820BD

With SHA-512, message = "sample":
k = 00BBCC2F39939388FDFE841892537EC7B1FF33AA3
r = 38E487F218D696A7323B891F0CCF055D895B77ADC
s = 0972D7721093F9B3835A5EB7F0442FA8DCAA873C4

```
With SHA-1, message = "test":
k = 14CAB9192F39C8A0EA8E81B4B87574228C99CD681
r = 1375BEF93F21582F601497036A7DC8014A99C2B79
s = 254B7F1472FFFEE9002D081BB8CE819CCE6E687F9

With SHA-224, message = "test":
k = 091DD986F38EB936BE053DD6ACE3419D2642ADE8D
r = 110F17EF209957214E35E8C2E83CBE73B3BFDEE2C
s = 057D5022392D359851B95DEC2444012502A5349CB

With SHA-256, message = "test":
k = 193649CE51F0CFF0784CFC47628F4FA854A93F7A2
r = 0354D5CD24F9C41F85D02E856FA2B0001C83AF53E
s = 020B200677731CD4FE48612A92F72A19853A82B65

With SHA-384, message = "test":
k = 37C73C6F8B404EC83DA17A6EBCA724B3FF1F7EEBA
r = 11B6A84206515495AD8DBB2E5785D6D018D75817E
s = 1A7D4C1E17D4030A5D748ADEA785C77A54581F6D0

With SHA-512, message = "test":
k = 331AD98D3186F73967B1E0B120C80B1E22EFC2988
r = 148934745B351F6367FF5BB56B1848A2F508902A9
s = 36214B19444FAB504DBA61D4D6FF2D2F9640F4837
```

A.2.9. ECDSA, 233 Bits (Binary Field, Koblitz Curve)

Key pair:

curve: NIST K-233

q = 8000000000000000000000000000069D5BB915BCD46EFB1AD5F173ABDF
(qlen = 232 bits)

private key:

x = 103B2142BDC2A3C3B55080D09DF1808F79336DA2399F5CA7171D1BE9B0

public key: U = xG

Ux = 0682886F36C68473C1A221720C2B12B9BE13458BA907E1C4736595779F2

Uy = 1B20639B41BE0927090999B7817A3B3928D20503A39546044EC13A10309

Signatures:

With SHA-1, message = "sample":
k = 273179E3E12C69591AD3DD9C7CCE3985820E3913AB6696EB14486DDBCF
r = 5474541C988A9A1F73899F55EF28963DFFBBF0C2B1A1EE787C6A76C6A4
s = 46301F9EC6624257BFC70D72186F17898EDBD0A3522560A88DD1B7D45A

With SHA-224, message = "sample":
k = 71626A309D9CD80AD0B975D757FE6BF4B84E49F8F34C780070D7746F19
r = 667F2FCE3E1C497EBD8E4B7C6372A8234003FE4ED6D4515814E7E11430
s = 6A1C41340DAA730320DB9475F10E29A127D7AE3432F155E1F7954E1B57

With SHA-256, message = "sample":
k = 73552F9CAC5774F74F485FA253871F2109A0C86040552EAA67DBA92DC9
r = 38AD9C1D2CB29906E7D63C24601AC55736B438FB14F4093D6C32F63A10
s = 647AAD2599C21B6EE89BE7FF957D98F684B7921DE1FD3CC82C079624F4

With SHA-384, message = "sample":
k = 17D726A67539C609BD99E29AA3737EF247724B71455C3B6310034038C8
r = 0C6510F57559C36FBCFF8C7BA4B81853DC618AD0BAAB03CFFDF3FD09FD
s = 0AD331EE1C9B91A88BA77997235769C60AD07EE69E11F7137E17C5CF67

With SHA-512, message = "sample":
k = 0E535C328774CDE546BE3AF5D7FCD263872F107E807435105BA2FDC166
r = 47C4AC1B344028CC740BA7BB9F8AA59D6390E3158153D4F2ADE4B74950
s = 26CE0CDE18A1B884B3EE1A879C13B42F11BB7C85F7A3745C8BECEC8E6E

```
With SHA-1, message = "test":
k = 1D8BBF5CB6EFFA270A1CDC22C81E269F0CC16E27151E0A460BA9B51AFF
r = 4780B2DE4BAA5613872179AD90664249842E8B96FCD5653B55DD63EED4
s = 6AF46BA322E21D4A88DAEC1650EF38774231276266D6A45ED6A64ECB44

With SHA-224, message = "test":
k = 67634D0ABA2C9BF7AE54846F26DCD166E7100654BCE6FDC96667631AA2
r = 61D9CC8C842DF19B3D9F4BDA0D0E14A957357ADABC239444610FB39AEA
s = 66432278891CB594BA8D08A0C556053D15917E53449E03C2EF88474CF6

With SHA-256, message = "test":
k = 2CE5AEDC155ACC0DDC5E679EBACFD21308362E5EFC05C5E99B2557A8D7
r = 05E4E6B4DB0E13034E7F1F2E5DBAB766D37C15AE4056C7EE607C8AC7F4
s = 5FC46AA489BF828B34FBAD25EC432190F161BEA8F60D3FCADB0EE3B725

With SHA-384, message = "test":
k = 1B4BD3903E74FD0B31E23F956C70062014DFEFEE21832032EA5352A055
r = 50F1EFEDFFEC1088024620280EE0D7641542E4D4B5D61DB32358FC571B
s = 4614EAE449927A9EB2FCC42EA3E955B43D194087719511A007EC9217A5

With SHA-512, message = "test":
k = 1775ED919CA491B5B014C5D5E86AF53578B5A7976378F192AF665CB705
r = 6FE6D0D3A953BB66BB01BC6B9EDFAD9F35E88277E5768D1B214395320F
s = 7C01A236E4BFF0A771050AD01EC1D24025D3130BBD9E4E81978EB3EC09
```

A.2.10. ECDSA, 283 Bits (Binary Field, Koblitz Curve)

Key pair:

curve: NIST K-283

q = 1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE9AE2ED07577265DFF7F94451E061
E163C61
(qlen = 281 bits)

private key:

x = 06A0777356E87B89BA1ED3A3D845357BE332173C8F7A65BDC7DB4FAB3C4CC79A
CC8194E

public key: U = xG

Ux = 25330D0A651D5A20DC6389BC02345117725640AEC3C126612CE444EDD19649BD
ECC03D6

Uy = 505BD60A4B67182474EC4D1C668A73140F70504A68F39EFCD972487E9530E050
8A76193

Signatures:

With SHA-1, message = "sample":
k = 0A96F788DECAF6C9DBE24DC75ABA6EAAE85E7AB003C8D4F83CB1540625B2993B
F445692
r = 1B66D1E33FBDB6E107A69B610995C93C744CEBAEAF623CB42737C27D60188BD1
D045A68
s = 02E45B62C9C258643532FD536594B46C63B063946494F95DAFF8759FD5525023
24295C5

With SHA-224, message = "sample":
k = 1B4C4E3B2F6B08B5991BD2BDDE277A7016DA527AD0AAE5BC61B64C5A0EE63E8B
502EF61
r = 018CF2F371BE86BB62E02B27CDE56DDAC83CCFBB3141FC59AEE022B66AC1A60D
BBD8B76
s = 1854E02A381295EA7F184CEE71AB7222D6974522D3B99B309B1A8025EB84118A
28BF20E

With SHA-256, message = "sample":
k = 1CEB9E8E0DFF53CE687DEB81339ACA3C98E7A657D5A9499EF779F887A934408E
CBE5A38
r = 19E90AA3DE5FB20AED22879F92C6FED278D9C9B9293CC5E94922CD952C9DBF20
DF1753A
s = 135AA7443B6A25D11BB64AC482E04D47902D017752882BD72527114F46CF8BB5
6C5A8C3

```
   With SHA-384, message = "sample":
   k = 1460A5C41745A5763A9D548AE62F2C3630BBED71B6AA549D7F829C22442A728C
       5D965DA
   r = 0F8C1CA9C221AD9907A136F787D33BA56B0495A40E86E671C940FD767EDD75EB
       6001A49
   s = 1071A56915DEE89E22E511975AA09D00CDC4AA7F5054CBE83F5977EE6F8E1CC3
       1EC43FD

   With SHA-512, message = "sample":
   k = 00F3B59FCB5C1A01A1A2A0019E98C244DFF61502D6E6B9C4E957EDDCEB258EF4
       DBEF04A
   r = 1D0008CF4BA4A701BEF70771934C2A4A87386155A2354140E2ED52E18553C35B
       47D9E50
   s = 0D15F4FA1B7A4D41D9843578E22EF98773179103DC4FF0DD1F74A6B5642841B9
       1056F78

   With SHA-1, message = "test":
   k = 168B5F8C0881D4026C08AC5894A2239D219FA9F4DA0600ADAA56D5A1781AF81F
       08A726E
   r = 140932FA7307666A8CCB1E1A09656CC40F5932965841ABD5E8E43559D93CF231
       1B02767
   s = 16A2FD46DA497E5E739DED67F426308C45C2E16528BF2A17EB5D65964FD88B77
       0FBB9C6

   With SHA-224, message = "test":
   k = 045E13EA645CE01D9B25EA38C8A8A170E04C83BB7F231EE3152209FE10EC8B2E
       565536C
   r = 0E72AF7E39CD72EF21E61964D87C838F977485FA6A7E999000AFA97A381B2445
       FCEE541
   s = 1644FF7D848DA1A040F77515082C27C763B1B4BF332BCF5D08251C6B57D80631
       9778208

   With SHA-256, message = "test":
   k = 0B585A7A68F51089691D6EDE2B43FC4451F66C10E65F134B963D4CBD4EB844B0
       E1469A6
   r = 158FAEB2470B306C57764AFC8528174589008449E11DB8B36994B607A65956A5
       9715531
   s = 0521BC667CA1CA42B5649E78A3D76823C678B7BB3CD58D2E93CD791D53043A6F
       83F1FD1

   With SHA-384, message = "test":
   k = 1E88738E14482A09EE16A73D490A7FE8739DF500039538D5C4B6C8D6D7F208D6
       CA56760
   r = 1CC4DC5479E0F34C4339631A45AA690580060BF0EB518184C983E0E618C3B93A
       AB14BBE
   s = 0284D72FF8AFA83DE364502CBA0494BB06D40AE08F9D9746E747EA87240E589B
       A0683B7
```

```
With SHA-512, message = "test":
k = 00E5F24A223BD459653F682763C3BB322D4EE75DD89C63D4DC61518D543E7658
    5076BBA
r = 1E7912517C6899732E09756B1660F6B96635D638283DF9A8A11D30E008895D7F
    5C9C7F3
s = 0887E75CBD0B7DD9DE30ED79BDB3D78E4F1121C5EAFF5946918F594F88D36364
    4789DA7
```

A.2.11. ECDSA, 409 Bits (Binary Field, Koblitz Curve)

Key pair:

curve: NIST K-409

q = 7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE5F83B2D4EA20
400EC4557D5ED3E3E7CA5B4B5C83B8E01E5FCF
(qlen = 407 bits)

private key:

x = 29C16768F01D1B8A89FDA85E2EFD73A09558B92A178A2931F359E4D70AD853E5
69CDAF16DAA569758FB4E73089E4525D8BBFCF

public key: U = xG

Ux = 0CF923F523FE34A6E863D8BA45FB1FE6D784C8F219C414EEF4DB8362DBBD3CA7
1AEB28F568668D5D7A0093E2B84F6FAD759DB42

Uy = 13B1C374D5132978A1B1123EBBE9A5C54D1A9D56B09AFDB4ADE93CCD7C4D332E
2916F7D4B9D18578EE3C2E2DE4D2ECE0DE63549

Signatures:

With SHA-1, message = "sample":
k = 7866E5247F9A3556F983C86E81EDA696AC8489DB40A2862F278603982D304F08
B2B6E1E7848534BEAF1330D37A1CF84C7994C1
r = 7192EE99EC7AFE23E02CB1F9850D1ECE620475EDA6B65D04984029408EC1E5A6
476BC940D81F218FC31D979814CAC6E78340FA
s = 1DE75DE97CBE740FC79A6B5B22BC2B7832C687E6960F0B8173D5D8BE2A75AC6C
A43438BAF69C669CE6D64E0FB93BC5854E0F81

With SHA-224, message = "sample":
k = 512340DB682C7B8EBE407BF1AA54194DFE85D49025FE0F632C9B8A06A996F2FC
D0D73C752FB09D23DB8FBE50605DC25DF0745C
r = 41C8EDF39D5E4E76A04D24E6BFD4B2EC35F99CD2483478FD8B0A03E99379576E
DACC4167590B7D9C387857A5130B1220CB771F
s = 659652EEAC9747BCAD58034B25362B6AA61836E1BA50E2F37630813050D43457
E62EAB0F13AE197E6CFE0244F983107555E269

With SHA-256, message = "sample":
k = 782385F18BAF5A36A588637A76DFAB05739A14163BF723A4417B74BD1469D37A
C9E8CCE6AEC8FF63F37B815AAF14A876EED962
r = 49EC220D6D24980693E6D33B191532EAB4C5D924E97E305E2C1CCFE6F1EAEF96
C17F6EC27D1E06191023615368628A7E0BD6A9
s = 1A4AB1DD9BAAA21F77C503E1B39E770FFD44718349D54BA4CF08F688CE89D7D7
C5F7213F225944BE5F7C9BA42B8BEE382F8AF9

```
   With SHA-384, message = "sample":
   k = 4DA637CB2E5C90E486744E45A73935DD698D4597E736DA332A06EDA8B26D5ABC
       6153EC2ECE14981CF3E5E023F36FFA55EEA6D7
   r = 562BB99EE027644EC04E493C5E81B41F261F6BD18FB2FAE3AFEAD91FAB8DD44A
       FA910B13B9C79C87555225219E44E72245BB7C
   s = 25BA5F28047DDDBDA7ED7E49DA31B62B20FD9C7E5B8988817BBF738B3F4DFDD2
       DCD06EE6DF2A1B744C850DAF952C12B9A56774

   With SHA-512, message = "sample":
   k = 57055B293ECFDFE983CEF716166091E573275C53906A39EADC25C89C5EC8D7A7
       E5629FCFDFAD514E1348161C9A34EA1C42D58C
   r = 16C7E7FB33B5577F7CF6F77762F0F2D531C6E7A3528BD2CF582498C1A48F2007
       89E9DF7B754029DA0D7E3CE96A2DC760932606
   s = 2729617EFBF80DA5D2F201AC7910D3404A992C39921C2F65F8CF4601392DFE93
       3E6457EAFDBD13DFE160D243100378B55C290A

   With SHA-1, message = "test":
   k = 545453D8DC05D220F9A12EF322D0B855E664C72835FABE8A41211453EB8A7CFF
       950D80773839D0043A46852DDA5A536E02291F
   r = 565648A5BAD24E747A7D7531FA9DBDFCB184ECFEFDB00A319459242B68D0989E
       52BED4107AED35C27D8ECA10E876ACA48006C9
   s = 7420BA6FF72ECC5C92B7CA0309258B5879F26393DB22753B9EC5DF905500A042
       28AC08880C485E2AC8834E13E8FA44FA57BF18

   With SHA-224, message = "test":
   k = 3C5352929D4EBE3CCE87A2DCE380F0D2B33C901E61ABC530DAF3506544AB0930
       AB9BFD553E51FCDA44F06CD2F49E17E07DB519
   r = 251DFE54EAEC8A781ADF8A623F7F36B4ABFC7EE0AE78C8406E93B5C3932A8120
       AB8DFC49D8E243C7C30CB5B1E021BADBDF9CA4
   s = 77854C2E72EAA6924CC0B5F6751379D132569843B1C7885978DBBAA6678967F6
       43A50DBB06E6EA6102FFAB7766A57C3887BD22

   With SHA-256, message = "test":
   k = 251E32DEE10ED5EA4AD7370DF3EFF091E467D5531CA59DE3AA791763715E1169
       AB5E18C2A11CD473B0044FB45308E8542F2EB0
   r = 58075FF7E8D36844EED0FC3F78B7CFFDEEF6ADE5982D5636552A081923E24841
       C9E37DF2C8C4BF2F2F7A174927F3B7E6A0BEB2
   s = 0A737469D013A31B91E781CE201100FDE1FA488ABF2252C025C678462D715AD3
       078C9D049E06555CABDF37878CFB909553FF51

   With SHA-384, message = "test":
   k = 11C540EA46C5038FE28BB66E2E9E9A04C9FE9567ADF33D56745953D44C1DC8B5
       B92922F53A174E431C0ED8267D919329F19014
   r = 1C5C88642EA216682244E46E24B7CE9AAEF9B3F97E585577D158C3CBC3C59825
       0A53F6D46DFB1E2DD9DC302E7DA4F0CAAFF291
   s = 1D3FD721C35872C74514359F88AD983E170E5DE5B31AFC0BE12E9F4AB2B2538C
       7797686BA955C1D042FD1F8CDC482775579F11
```

```
With SHA-512, message = "test":
k = 59527CE953BC09DF5E85155CAE7BB1D7F342265F41635545B06044F844ECB4FA
    6476E7D47420ADC8041E75460EC0A4EC760E95
r = 1A32CD7764149DF79349DBF79451F4585BB490BD63A200700D7111B45DDA4140
    00AE1B0A69AEACBA1364DD7719968AAD123F93
s = 582AB1076CAFAE23A76244B82341AEFC4C6D8D8060A62A352C33187720C8A37F
    3DAC227E62758B11DF1562FD249941C1679F82
```

A.2.12. ECDSA, 571 Bits (Binary Field, Koblitz Curve)

Key pair:

curve: NIST K-571

q = 2000000000000000000000000000000000000000000000000000000000000000
0000000131850E1F19A63E4B391A8DB917F4138B630D84BE5D639381E91DEB45
CFE778F637C1001
(qlen = 570 bits)

private key:

x = 0C16F58550D824ED7B95569D4445375D3A490BC7E0194C41A39DEB732C29396C
DF1D66DE02DD1460A816606F3BEC0F32202C7BD18A32D87506466AA92032F131
4ED7B19762B0D22

public key: U = xG

Ux = 6CFB0DF7541CDD4C41EF319EA88E849EFC8605D97779148082EC991C463ED323
19596F9FDF4779C17CAF20EFD9BEB57E9F4ED55BFC52A2FA15CA23BC62B7BF01
9DB59793DD77318

Uy = 1CFC91102F7759A561BD8D5B51AAAEEC7F40E659D67870361990D6DE29F6B4F7
E18AE13BDE5EA5C1F77B23D676F44050C9DBFCCDD7B3756328DDA059779AAE84
46FC5158A75C227

Signatures:

With SHA-1, message = "sample":
k = 17F7E360B21BEAE4A757A19ACA77FB404D273F05719A86EAD9D7B3F4D5ED7B46
30584BB153CF7DCD5A87CCA101BD7EA9ECA0CE5EE27CA985833560000BB52B6B
BE068740A45B267
r = 0767913F96C82E38B7146A505938B79EC07E9AA3214377651BE968B52C039D3E
4837B4A2DE26C481C4E1DE96F4D9DE63845D9B32E26D0D332725678E3CE57F66
8A5E3108FB6CEA5
s = 109F89F55FA39FF465E40EBCF869A9B1DB425AEA53AB4ECBCE3C310572F79315
F5D4891461372A0C36E63871BEDDBB3BA2042C6410B67311F1A185589FF4C987
DBA02F9D992B9DF

```
   With SHA-224, message = "sample":
   k = 0B599D068A1A00498EE0B9AD6F388521F594BD3F234E47F7A1DB6490D7B57D60
       B0101B36F39CC22885F78641C69411279706F0989E6991E5D5B53619E43EFB39
       7E25E0814EF02BC
   r = 010774B9F14DE6C9525131AD61531FA30987170D43782E9FB84FF0D70F093946
       DF75ECB69D400FE39B12D58C67C19DCE96335CEC1D9AADE004FE5B498AB8A940
       D46C8444348686A
   s = 06DFE9AA5FEA6CF2CEDC06EE1F9FD9853D411F0B958F1C9C519C90A85F6D24C1
       C3435B3CDF4E207B4A67467C87B7543F6C0948DD382D24D1E48B3763EC27D4D3
       2A0151C240CC5E0

   With SHA-256, message = "sample":
   k = 0F79D53E63D89FB87F4D9E6DC5949F5D9388BCFE9EBCB4C2F7CE497814CF40E8
       45705F8F18DBF0F860DE0B1CC4A433EF74A5741F3202E958C082E0B76E16ECD5
       866AA0F5F3DF300
   r = 1604BE98D1A27CEC2D3FA4BD07B42799E07743071E4905D7DCE7F6992B21A27F
       14F55D0FE5A7810DF65CF07F2F2554658817E5A88D952282EA1B8310514C0B40
       FFF46F159965168
   s = 18249377C654B8588475510F7B797081F68C2F8CCCE49F730353B2DA3364B1CD
       3E984813E11BB791824038EA367BA74583AB97A69AF2D77FA691AA694E348E15
       DA76F5A44EC1F40

   With SHA-384, message = "sample":
   k = 0308253C022D25F8A9EBCD24459DD6596590BDEC7895618EEE8A2623A98D2A2B
       2E7594EE6B7AD3A39D70D68CB4ED01CB28E2129F8E2CC0CC8DC7780657E28BCD
       655F0BE9B7D35A2
   r = 1E6D7FB237040EA1904CCBF0984B81B866DE10D8AA93B06364C4A46F6C9573FA
       288C8BDDCC0C6B984E6AA75B42E7BF82FF34D51DFFBD7C87FDBFAD971656185B
       D12E4B8372F4BF1
   s = 04F94550072ADA7E8C82B7E83577DD39959577799CDABCEA60E267F36F1BEB98
       1ABF24E722A7F031582D2CC5D80DAA7C0DEEBBE1AC5E729A6DBB34A5D645B698
       719FCA409FBA370

   With SHA-512, message = "sample":
   k = 0C5EE7070AF55F84EBC43A0D481458CEDE1DCEBB57720A3C92F59B4941A044FE
       CFF4F703940F3121773595E880333772ACF822F2449E17C64DA286BCD65711DD
       5DA44D7155BF004
   r = 086C9E048EADD7D3D2908501086F3AF449A01AF6BEB2026DC381B39530BCDDBE
       8E854251CBD5C31E6976553813C11213E4761CB8CA2E5352240AD9FB9C635D55
       FAB13AE42E4EE4F
   s = 09FEE0A68F322B380217FCF6ABFF15D78C432BD8DD82E18B6BA877C01C860E24
       410F5150A44F979920147826219766ECB4E2E11A151B6A15BB8E2E825AC95BCC
       A228D8A1C9D3568
```

```
   With SHA-1, message = "test":
   k = 1D056563469E933E4BE064585D84602D430983BFBFD6885A94BA484DF9A7AB03
       1AD6AC090A433D8EEDC0A7643EA2A9BC3B6299E8ABA933B4C1F2652BB49DAEE8
       33155C8F1319908
   r = 1D055F499A3F7E3FC73D6E7D517B470879BDCB14ABC938369F23643C7B96D024
       2C1FF326FDAF1CCC8593612ACE982209658E73C24C9EC493B785608669DA74A5
       B7C9A1D8EA843BC
   s = 1621376C53CFE3390A0520D2C657B1FF0EBB10E4B9C2510EDC39D04FEBAF12B8
       502B098A8B8F842EA6E8EB9D55CFEF94B7FF6D145AC3FFCE71BD978FEA3EF819
       4D4AB5293A8F3EA

   With SHA-224, message = "test":
   k = 1DA875065B9D94DBE75C61848D69578BCC267935792624F9887B53C9AF9E43CA
       BFC42E4C3F9A456BA89E717D24F1412F33CFD297A7A4D403B18B5438654C74D5
       92D5022125E0C6B
   r = 18709BDE4E9B73D046CE0D48842C97063DA54DCCA28DCB087168FA37DA2BF5FD
       BE4720EE48D49EDE4DD5BD31AC0149DB8297BD410F9BC02A11EB79B60C8EE63A
       F51B65267D71881
   s = 12D8B9E98FBF1D264D78669E236319D8FFD8426C56AFB10C76471EE88D7F0AB1
       B158E685B6D93C850D47FB1D02E4B24527473DB60B8D1AEF26CEEBD3467B65A7
       0FFDDC0DBB64D5F

   With SHA-256, message = "test":
   k = 04DDD0707E81BB56EA2D1D45D7FAFDBDD56912CAE224086802FEA1018DB306C4
       FB8D93338DBF6841CE6C6AB1506E9A848D2C0463E0889268843DEE4ACB552CFF
       CB858784ED116B2
   r = 1F5BF6B044048E0E310309FFDAC825290A69634A0D3592DBEE7BE71F69E45412
       F766AC92E174CC99AABAA5C9C89FCB187DFDBCC7A26765DB6D9F1EEC8A6127BB
       DFA5801E44E3BEC
   s = 1B44CBFB233BFA2A98D5E8B2F0B2C27F9494BEAA77FEB59CDE3E7AE9CB2E385B
       E8DA7B80D7944AA71E0654E5067E9A70E88E68833054EED49F28283F02B22912
       3995AF37A6089F0

   With SHA-384, message = "test":
   k = 0141B53DC6E569D8C0C0718A58A5714204502FDA146E7E2133E56D19E905B794
       13457437095DE13CF68B5CF5C54A1F2E198A55D974FC3E507AFC0ACF95ED391C
       93CC79E3B3FE37C
   r = 11F61A6EFAB6D83053D9C52665B3542FF3F63BD5913E527BDBA07FBAF34BC766
       C2EC83163C5273243AA834C75FDDD1BC8A2BEAD388CD06C4EBA1962D645EEB35
       E92D44E8F2E081D
   s = 16BF6341876F051DF224770CC8BA0E4D48B3332568A2B014BC80827BAA89DE18
       D1AEBC73E3BE8F85A8008C682AAC7D5F0E9FB5ECBEFBB637E30E4A0F226D2C2A
       A3E569BB54AB72B
```

```
   With SHA-512, message = "test":
   k = 14842F97F263587A164B215DD0F912C588A88DC4AB6AF4C530ADC1226F16E086
       D62C14435E6BFAB56F019886C88922D2321914EE41A8F746AAA2B964822E4AC6
       F40EE2492B66824
   r = 0F1E50353A39EA64CDF23081D6BB4B2A91DD73E99D3DD5A1AA1C49B4F6E34A66
       5EAD24FD530B9103D522609A395AF3EF174C85206F67EF84835ED1632E0F6BAB
       718EA90DF9E2DA0
   s = 0B385004D7596625028E3FDE72282DE4EDC5B4CE33C1127F21CC37527C90B730
       7AE7D09281B840AEBCECAA711B00718103DDB32B3E9F6A9FBC6AF23E224A73B9
       435F619D9C62527
```

A.2.13. ECDSA, 163 Bits (Binary Field, Pseudorandom Curve)

Key pair:

curve: NIST B-163

q = 4000000000000000000292FE77E70C12A4234C33
(qlen = 163 bits)

private key:

x = 35318FC447D48D7E6BC93B48617DDDEDF26AA658F

public key: U = xG

Ux = 126CF562D95A1D77D387BA75A3EA3A1407F23425A

Uy = 7D7CB5273C94DA8CA93049AFDA18721C24672BD71

Signatures:

With SHA-1, message = "sample":
k = 0707A94C3D352E0A9FE49FB12F264992152A20004
r = 153FEBD179A69B6122DEBF5BC61EB947B24C93526
s = 37AC9C670F8CF18045049BAE7DD35553545C19E49

With SHA-224, message = "sample":
k = 3B24C5E2C2D935314EABF57A6484289B291ADFE3F
r = 0A379E69C44F9C16EA3215EA39EB1A9B5D58CC955
s = 04BAFF5308DA2A7FE2C1742769265AD3ED1D24E74

With SHA-256, message = "sample":
k = 3D7086A59E6981064A9CDB684653F3A81B6EC0F0B
r = 134E00F78FC1CB9501675D91C401DE20DDF228CDC
s = 373273AEC6C36CB7BAFBB1903A5F5EA6A1D50B624

With SHA-384, message = "sample":
k = 3B1E4443443486C7251A68EF184A936F05F8B17C7
r = 29430B935AF8E77519B0CA4F6903B0B82E6A21A66
s = 1EA1415306E9353FA5AA54BC7C2581DFBB888440D

With SHA-512, message = "sample":
k = 2EDF5CFCAC7553C17421FDF54AD1D2EF928A879D2
r = 0B2F177A99F9DF2D51CCAF55F015F326E4B65E7A0
s = 0DF1FB4487E9B120C5E970EFE48F55E406306C3A1

```
With SHA-1, message = "test":
k = 10024F5B324CBC8954BA6ADB320CD3AB9296983B4
r = 256D4079C6C7169B8BC92529D701776A269D56308
s = 341D3FFEC9F1EB6A6ACBE88E3C86A1C8FDEB8B8E1

With SHA-224, message = "test":
k = 34F46DE59606D56C75406BFB459537A7CC280AA62
r = 28ECC6F1272CE80EA59DCF32F7AC2D861BA803393
s = 0AD4AE2C06E60183C1567D2B82F19421FE3053CE2

With SHA-256, message = "test":
k = 38145E3FFCA94E4DDACC20AD6E0997BD0E3B669D2
r = 227DF377B3FA50F90C1CB3CDCBBDBA552C1D35104
s = 1F7BEAD92583FE920D353F368C1960D0E88B46A56

With SHA-384, message = "test":
k = 375813210ECE9C4D7AB42DDC3C55F89189CF6DFFD
r = 11811DAFEEA441845B6118A0DFEE8A0061231337D
s = 36258301865EE48C5C6F91D63F62695002AB55B57

With SHA-512, message = "test":
k = 25AD8B393BC1E9363600FDA1A2AB6DF40079179A3
r = 3B6BB95CA823BE2ED8E3972FF516EB8972D765571
s = 13DC6F420628969DF900C3FCC48220B38BE24A541
```

A.2.14. ECDSA, 233 Bits (Binary Field, Pseudorandom Curve)

Key pair:

curve: NIST B-233

q = 1000000000000000000000000000013E974E72F8A6922031D2603CFE0D7
(qlen = 233 bits)

private key:

x = 07ADC13DD5BF34D1DDEEB50B2CE23B5F5E6D18067306D60C5F6FF11E5D3

public key: U = xG

Ux = 0FB348B3246B473AA7FBB2A01B78D61B62C4221D0F9AB55FC72DB3DF478

Uy = 1162FA1F6C6ACF7FD8D19FC7D74BDD9104076E833898BC4C042A6E6BEBF

Signatures:

With SHA-1, message = "sample":
k = 0A4E0B67A3A081C1B35D7BECEB5FE72A918B422B907145DB5416ED751CE
r = 015CC6FD78BB06E0878E71465515EA5A21A2C18E6FC77B4B158DBEB3944
s = 0822A4A6C2EB2DF213A5E90BF40377956365EE8C4B4A5A4E2EB9270CB6A

With SHA-224, message = "sample":
k = 0F2B1C1E80BEB58283AAA79857F7B83BDF724120D0913606FD07F7FFB2C
r = 05D9920B53471148E10502AB49AB7A3F11084820A074FD89883CF51BC1A
s = 04D3938900C0A9AAA7080D1DFEB56CFB0FADABE4214536C7ED5117ED13A

With SHA-256, message = "sample":
k = 034A53897B0BBDB484302E19BF3F9B34A2ABFED639D109A388DC52006B5
r = 0A797F3B8AEFCE7456202DF1E46CCC291EA5A49DA3D4BDDA9A4B62D5E0D
s = 01F6F81DA55C22DA4152134C661588F4BD6F82FDBAF0C5877096B070DC2

With SHA-384, message = "sample":
k = 04D4670B28990BC92EEB49840B482A1FA03FE028D09F3D21F89C67ECA85
r = 015E85A8D46225DD7E314A1C4289731FC14DECE949349FE535D11043B85
s = 03F189D37F50493EFD5111A129443A662AB3C6B289129AD8C0CAC85119C

With SHA-512, message = "sample":
k = 0DE108AAADA760A14F42C057EF81C0A31AF6B82E8FBCA8DC86E443AB549
r = 03B62A4BF783919098B1E42F496E65F7621F01D1D466C46940F0F132A95
s = 0F4BE031C6E5239E7DAA014CBBF1ED19425E49DAEB426EC9DF4C28A2E30

```
With SHA-1, message = "test":
k = 0250C5C90A4E2A3F8849FEBA87F0D0AE630AB18CBABB84F4FFFB36CEAC0
r = 02F1FEDC57BE203E4C8C6B8C1CEB35E13C1FCD956AB41E3BD4C8A6EFB1F
s = 05738EC8A8EDEA8E435EE7266AD3EDE1EEFC2CEBE2BE1D614008D5D2951

With SHA-224, message = "test":
k = 07BDB6A7FD080D9EC2FC84BFF9E3E15750789DC04290C84FED00E109BBD
r = 0CCE175124D3586BA7486F7146894C65C2A4A5A1904658E5C7F9DF5FA5D
s = 08804B456D847ACE5CA86D97BF79FD6335E5B17F6C0D964B5D0036C867E

With SHA-256, message = "test":
k = 00376886E89013F7FF4B5214D56A30D49C99F53F211A3AFE01AA2BDE12D
r = 035C3D6DFEEA1CFB29B93BE3FDB91A7B130951770C2690C16833A159677
s = 0600F7301D12AB376B56D4459774159ADB51F97E282FF384406AFD53A02

With SHA-384, message = "test":
k = 03726870DE75613C5E529E453F4D92631C03D08A7F63813E497D4CB3877
r = 061602FC8068BFD5FB86027B97455D200EC603057446CCE4D76DB8EF42C
s = 03396DD0D59C067BB999B422D9883736CF9311DFD6951F91033BD03CA8D

With SHA-512, message = "test":
k = 09CE5810F1AC68810B0DFFBB6BEEF2E0053BB937969AE7886F9D064A8C4
r = 07E12CB60FDD614958E8E34B3C12DDFF35D85A9C5800E31EA2CC2EF63B1
s = 0E8970FD99D836F3CC1C807A2C58760DE6EDAA23705A82B9CB1CE93FECC
```

A.2.15. ECDSA, 283 Bits (Binary Field, Pseudorandom Curve)

Key pair:

curve: NIST B-283

q = 3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEF90399660FC938A90165B042A7CE
FADB307
(qlen = 282 bits)

private key:

x = 14510D4BC44F2D26F4553942C98073C1BD35545CEABB5CC138853C5158D2729E
A408836

public key: U = xG

Ux = 17E3409A13C399F0CA8A192F028D46E3446BCFFCDF51FF8A905ED2DED786E74F
9C3E8A9

Uy = 47EFCBCC31C01D86D1992F7BFAC0277DBD02A6D289274099A2C0F039C8F59F31
8371B0E

Signatures:

With SHA-1, message = "sample":
k = 277F389559667E8AE4B65DC056F8CE2872E1917E7CC59D17D485B0B98343206F
BCCD441
r = 201E18D48C6DB3D5D097C4DCE1E25587E1501FC3CF47BDB5B4289D79E273D6A9
ACB8285
s = 151AE05712B024CE617358260774C8CA8B0E7A7E72EF8229BF2ACE7609560CB3
0322C4F

With SHA-224, message = "sample":
k = 14CC8FCFEECD6B999B4DC6084EBB06FDED0B44D5C507802CC7A5E9ECF36E69DA
6AE23C6
r = 143E878DDFD4DF40D97B8CD638B3C4706501C2201CF7108F2FB91478C11D6947
3246925
s = 0CBF1B9717FEEA3AABB09D9654110144267098E0E1E8D0289A6211BE0EEDFDD8
6A3DB79

With SHA-256, message = "sample":
k = 38C9D662188982943E080B794A4CFB0732DBA37C6F40D5B8CFADED6FF31C5452
BA3F877
r = 29FD82497FB3E5CEF65579272138DE59E2B666B8689466572B3B69A172CEE83B
E145659
s = 05A89D9166B40795AF0FE5958201B9C0523E500013CA12B4840EA2BC53F25F9B
3CE87C0

```
   With SHA-384, message = "sample":
   k = 21B7265DEBF90E6F988CFFDB62B121A02105226C652807CC324ED6FB119A287A
       72680AB
   r = 2F00689C1BFCD2A8C7A41E0DE55AE182E6463A152828EF89FE3525139B660329
       4E69353
   s = 1744514FE0A37447250C8A329EAAADA81572226CABA16F39270EE5DD03F27B1F
       665EB5D

   With SHA-512, message = "sample":
   k = 20583259DC179D9DA8E5387E89BFF2A3090788CF1496BCABFE7D45BB120B0C81
       1EB8980
   r = 0DA43A9ADFAA6AD767998A054C6A8F1CF77A562924628D73C62761847AD8286E
       0D91B47
   s = 1D118733AE2C88357827CAFC6F68ABC25C80C640532925E95CFE66D40F8792F3
       AC44C42

   With SHA-1, message = "test":
   k = 0185C57A743D5BA06193CE2AA47B07EF3D6067E5AE1A6469BCD3FC510128BA56
       4409D82
   r = 05A408133919F2CDCDBE5E4C14FBC706C1F71BADAFEF41F5DE4EC27272FC1CA9
       366FBB2
   s = 012966272872C097FEA7BCE64FAB1A81982A773E26F6E4EF7C99969846E67CA9
       CBE1692

   With SHA-224, message = "test":
   k = 2E5C1F00677A0E015EC3F799FA9E9A004309DBD784640EAAF5E1CE64D3045B9F
       E9C1FA1
   r = 08F3824E40C16FF1DDA8DC992776D26F4A5981AB5092956C4FDBB4F1AE0A711E
       EAA10E5
   s = 0A64B91EFADB213E11483FB61C73E3EF63D3B44EEFC56EA401B99DCC60CC28E9
       9F0F1FA

   With SHA-256, message = "test":
   k = 018A7D44F2B4341FEFE68F6BD8894960F97E08124AAB92C1FFBBE90450FCC935
       6C9AAA5
   r = 3597B406F5329D11A79E887847E5EC60861CCBB19EC61F252DB7BD549C699951
       C182796
   s = 0A6A100B997BC622D91701D9F5C6F6D3815517E577622DA69D3A0E8917C1CBE6
       3ACD345

   With SHA-384, message = "test":
   k = 3C75397BA4CF1B931877076AF29F2E2F4231B117AB4B8E039F7F9704DE1BD352
       2F150B6
   r = 1BB490926E5A1FDC7C5AA86D0835F9B994EDA315CA408002AF54A298728D422E
       BF59E4C
   s = 36C682CFC9E2C89A782BFD3A191609D1F0C1910D5FD6981442070393159D65FB
       CC0A8BA
```

```
With SHA-512, message = "test":
k = 14E66B18441FA54C21E3492D0611D2B48E19DE3108D915FD5CA08E786327A267
    5F11074
r = 19944AA68F9778C2E3D6E240947613E6DA60EFCE9B9B2C063FF5466D72745B5A
    0B25BA2
s = 03F1567B3C5B02DF15C874F0EE22850824693D5ADC4663BAA19E384E550B1DD4
    1F31EE6
```

A.2.16. ECDSA, 409 Bits (Binary Field, Pseudorandom Curve)

Key pair:

curve: NIST B-409

q = 1000000000000000000000000000000000000000000000000001E2AAD6A612F
33307BE5FA47C3C9E052F838164CD37D9A21173
(qlen = 409 bits)

private key:

x = 0494994CC325B08E7B4CE038BD9436F90B5E59A2C13C3140CD3AE07C04A01FC4
89F572CE0569A6DB7B8060393DE76330C624177

public key: U = xG

Ux = 1A7055961CF1DA4B9A015B18B1524EF01FDD9B93FAEFC26FB1F2F828A7227B70
31925DA0AC1A8A075C3B33554B222EA859C17E7

Uy = 18105C042F290736088F30AEC7AE7732A45DE47BCE0940113AB8132516D1E059
B0F581FD581A9A3CB3A0AC42A1962738ADB86E6

Signatures:

With SHA-1, message = "sample":
k = 042D8A2B34402757EB2CCFDDC3E6E96A7ADD3FDA547FC10A0CB77CFC720B4F9E
16EEAAA2A8CC4E4A4B5DBF7D8AC4EA491859E60
r = 0D8783188E1A540E2022D389E1D35B32F56F8C2BB5636B8ABF7718806B27A713
EBAE37F63ECD4B61445CEF5801B62594EF3E982
s = 03A6B4A80E204DB0DE12E7415C13C9EC091C52935658316B4A0C591216A38791
54BEB1712560E346E7EF26517707435B55C3141

With SHA-224, message = "sample":
k = 0C933F1DC4C70838C2AD16564715ACAF545BCDD8DC203D25AF3EC63949C65CB2
E68AC1F60CA7EACA2A823F4E240927AA82CEEC5
r = 0EE4F39ACC2E03CE96C3D9FCBAFA5C22C89053662F8D4117752A9B10F09ADFDA
59DB061E247FE5321D6B170EE758ACE1BE4D157
s = 00A2B83265B456A430A8BF27DCC8A9488B3F126C10F0D6D64BF7B8A218FAAF20
E51A295A3AE78F205E5A4A6AE224C3639F1BB34

With SHA-256, message = "sample":
k = 08EC42D13A3909A20C41BEBD2DFED8CACCE56C7A7D1251DF43F3E9E289DAE00E
239F6960924AC451E125B784CB687C7F23283FD
r = 02D8B1B31E33E74D7EB46C30FDE5AD2CA04EC8FE08FBA0E73BA5E568953AC5EA
307C072942238DFC07F4A4D7C7C6A9F86436D17
s = 079F7D471E6CB73234AF7F7C381D2CE15DE35BAF8BB68393B73235B3A26EC2DF
4842CE433FB492D6E074E604D4870024D42189A

```
   With SHA-384, message = "sample":
   k = 0DA881BCE3BA851485879EF8AC585A63F1540B9198ECB8A1096D70CB25A104E2
       F8A96B108AE76CB49CF34491ABC70E9D2AAD450
   r = 07BC638B7E7CE6FEE5E9C64A0F966D722D01BB4BC3F3A35F30D4CDDA92DFC5F7
       F0B4BBFE8065D9AD452FD77A1914BE3A2440C18
   s = 06D904429850521B28A32CBF55C7C0FDF35DC4E0BDA2552C7BF68A171E970E67
       88ACC0B9521EACB4796E057C70DD9B95FED5BFB

   With SHA-512, message = "sample":
   k = 0750926FFAD7FF5DE85DF7960B3A4F9E3D38CF5A049BFC89739C48D42B34FBEE
       03D2C047025134CC3145B60AFD22A68DF0A7FB2
   r = 05D178DECAFD2D02A3DA0D8BA1C4C1D95EE083C760DF782193A9F7B4A8BE6FC5
       C21FD60613BCA65C063A61226E050A680B3ABD4
   s = 013B7581E98F6A63FBBCB3E49BCDA60F816DB230B888506D105DC229600497C3
       B46588C784BE3AA9343BEF82F7C9C80AEB63C3B

   With SHA-1, message = "test":
   k = 017E167EAB1850A3B38EE66BFE2270F2F6BFDAC5E2D227D47B20E75F0719161E
       6C74E9F23088F0C58B1E63BC6F185AD2EF4EAE6
   r = 049F54E7C10D2732B4638473053782C6919218BBEFCEC8B51640FC193E832291
       F05FA12371E9B448417B3290193F08EE9319195
   s = 0499E267DEC84E02F6F108B10E82172C414F15B1B7364BE8BFD66ADC0C5DE23F
       EE3DF0D811134C25AFE0E05A6672F98889F28F1

   With SHA-224, message = "test":
   k = 01ADEB94C19951B460A146B8275D81638C07735B38A525D76023AAF26AA8A058
       590E1D5B1E78AB3C91608BDA67CFFBE6FC8A6CC
   r = 0B1527FFAA7DD7C7E46B628587A5BEC0539A2D04D3CF27C54841C2544E1BBDB4
       2FDBDAAF8671A4CA86DFD619B1E3732D7BB56F2
   s = 0442C68C044868DF4832C807F1EDDEBF7F5052A64B826FD03451440794063F52
       B022DF304F47403D4069234CA9EB4C964B37C02

   With SHA-256, message = "test":
   k = 06EBA3D58D0E0DFC406D67FC72EF0C943624CF40019D1E48C3B54CCAB0594AFD
       5DEE30AEBAA22E693DBCFECAD1A85D774313DAD
   r = 0BB27755B991D6D31757BCBF68CB01225A38E1CFA20F775E861055DD108ED7EA
       455E4B96B2F6F7CD6C6EC2B3C70C3EDDEB9743B
   s = 0C5BE90980E7F444B5F7A12C9E9AC7A04CA81412822DD5AD1BE7C45D5032555E
       A070864245CF69266871FEB8CD1B7EDC30EF6D5

   With SHA-384, message = "test":
   k = 0A45B787DB44C06DEAB846511EEDBF7BFCFD3BD2C11D965C92FC195F67328F36
       A2DC83C0352885DAB96B55B02FCF49DCCB0E2DA
   r = 04EFEB7098772187907C87B33E0FBBA4584226C50C11E98CA7AAC6986F8D3BE0
       44E5B52D201A410B852536527724CA5F8CE6549
   s = 09574102FEB3EF87E6D66B94119F5A6062950FF4F902EA1E6BD9E2037F33FF99
       1E31F5956C23AFE48FCDC557FD6F088C7C9B2B3
```

```
With SHA-512, message = "test":
k = 0B90F8A0E757E81D4EA6891766729C96A6D01F9AEDC0D334932D1F81CC4E1973
    A4F01C33555FF08530A5098CADB6EDAE268ABB5
r = 07E0249C68536AE2AEC2EC30090340DA49E6DC9E9EEC8F85E5AABFB234B6DA7D
    2E9524028CF821F21C6019770474CC40B01FAF6
s = 08125B5A03FB44AE81EA46D446130C2A415ECCA265910CA69D55F2453E16CD7B
    2DFA4E28C50FA8137F9C0C6CEE4CD37ABCCF6D8
```

A.2.17. ECDSA, 571 Bits (Binary Field, Pseudorandom Curve)

Key pair:

curve: NIST B-571

q = 3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFE661CE18FF55987308059B186823851EC7DD9CA1161DE93D5174D66E8
382E9BB2FE84E47
(qlen = 570 bits)

private key:

x = 028A04857F24C1C082DF0D909C0E72F453F2E2340CCB071F0E389BCA2575DA19
124198C57174929AD26E348CF63F78D28021EF5A9BF2D5CBEAF6B7CCB6C4DA82
4DD5C82CFB24E11

public key: U = xG

Ux = 4B4B3CE9377550140B62C1061763AA524814DDCEF37B00CD5CDE94F7792BB0E9
6758E55DA2E9FEA8FF2A8B6830AE1D57A9CA7A77FCB0836BF43EA5454CDD9FEA
D5CCFE7375C6A83

Uy = 4453B18F261E7A0E7570CD72F235EA750438E43946FBEBD2518B696954767AA7
849C1719E18E1C51652C28CA853426F15C09AA4B579487338ABC7F33768FADD6
1B5A3A6443A8189

Signatures:

With SHA-1, message = "sample":
k = 2669FAFEF848AF67D437D4A151C3C5D3F9AA8BB66EDC35F090C9118F95BA0041
B0993BE2EF55DAAF36B5B3A737C40DB1F6E3D93D97B8419AD6E1BB8A5D4A0E9B
2E76832D4E7B862
r = 147D3EB0EDA9F2152DFD014363D6A9CE816D7A1467D326A625FC4AB0C786E1B7
4DDF7CD4D0E99541391B266C704BB6B6E8DCCD27B460802E0867143727AA4155
55454321EFE5CB6
s = 17319571CAF533D90D2E78A64060B9C53169AB7FC908947B3EDADC54C79CCF0A
7920B4C64A4EAB6282AFE9A459677CDA37FD6DD50BEF18709590FE18B923BDF7
4A66B189A850819

```
With SHA-224, message = "sample":
k = 2EAFAD4AC8644DEB29095BBAA88D19F31316434F1766AD4423E0B54DD2FE0C05
    E307758581B0DAED2902683BBC7C47B00E63E3E429BA54EA6BA3AEC33A94C9A2
    4A6EF8E27B7677A
r = 10F4B63E79B2E54E4F4F6A2DBC786D8F4A143ECA7B2AD97810F6472AC6AE2085
    3222854553BE1D44A7974599DB7061AE8560DF57F2675BE5F9DD94ABAF3D47F1
    582B318E459748B
s = 3BBEA07C6B269C2B7FE9AE4DDB118338D0C2F0022920A7F9DCFCB7489594C03B
    536A9900C4EA6A10410007222D3DAE1A96F291C4C9275D75D98EB290DC0EEF17
    6037B2C7A7A39A3

With SHA-256, message = "sample":
k = 15C2C6B7D1A070274484774E558B69FDFA193BDB7A23F27C2CD24298CE1B22A6
    CC9B7FB8CABFD6CF7C6B1CF3251E5A1CDDD16FBFED28DE79935BB2C631B8B8EA
    9CC4BCC937E669E
r = 213EF9F3B0CFC4BF996B8AF3A7E1F6CACD2B87C8C63820000800AC787F17EC99
    C04BCEDF29A8413CFF83142BB88A50EF8D9A086AF4EB03E97C567500C21D8657
    14D832E03C6D054
s = 3D32322559B094E20D8935E250B6EC139AC4AAB77920812C119AF419FB62B332
    C8D226C6C9362AE3C1E4AABE19359B8428EA74EC8FBE83C8618C2BCCB6B43FBA
    A0F2CCB7D303945

With SHA-384, message = "sample":
k = 0FEF0B68CB49453A4C6ECBF1708DBEEFC885C57FDAFB88417AAEFA5B1C35017B
    4B498507937ADCE2F1D9EFFA5FE8F5AEB116B804FD182A6CF1518FDB62D53F60
    A0FF6EB707D856B
r = 375D8F49C656A0BBD21D3F54CDA287D853C4BB1849983CD891EF6CD6BB56A62B
    687807C16685C2C9BCA2663C33696ACCE344C45F3910B1DF806204FF731ECB28
    9C100EF4D1805EC
s = 1CDEC6F46DFEEE44BCE71D41C60550DC67CF98D6C91363625AC2553E4368D2DF
    B734A8E8C72E118A76ACDB0E58697940A0F3DF49E72894BD799450FC9E550CC0
    4B9FF9B0380021C

With SHA-512, message = "sample":
k = 3FF373833A06C791D7AD586AFA3990F6EF76999C35246C4AD0D519BFF180CA18
    80E11F2FB38B764854A0AE3BECDDB50F05AC4FCEE542F207C0A6229E2E19652F
    0E647B9C4882193
r = 1C26F40D940A7EAA0EB1E62991028057D91FEDA0366B606F6C434C361F04E545
    A6A51A435E26416F6838FFA260C617E798E946B57215284182BE55F29A355E60
    24FE32A47289CF0
s = 3691DE4369D921FE94EDDA67CB71FBBEC9A436787478063EB1CC778B3DCDC1C4
    162662752D28DEEDF6F32A269C82D1DB80C87CE4D3B662E03AC347806E3F19D1
    8D6D4DE7358DF7E
```

```
   With SHA-1, message = "test":
   k = 019B506FD472675A7140E429AA5510DCDDC21004206EEC1B39B28A688A8FD324
       138F12503A4EFB64F934840DFBA2B4797CFC18B8BD0B31BBFF3CA66A4339E4EF
       9D771B15279D1DC
   r = 133F5414F2A9BC41466D339B79376038A64D045E5B0F792A98E5A7AA87E0AD01
       6419E5F8D176007D5C9C10B5FD9E2E0AB8331B195797C0358BA05ECBF24ACE59
       C5F368A6C0997CC
   s = 3D16743AE9F00F0B1A500F738719C5582550FEB64689DA241665C4CE4F328BA0
       E34A7EF527ED13BFA5889FD2D1D214C11EB17D6BC338E05A56F41CAFF1AF7B8D
       574DB62EF0D0F21

   With SHA-224, message = "test":
   k = 333C711F8C62F205F926593220233B06228285261D34026232F6F729620C6DE1
       2220F282F4206D23226705608688B20B8BA86D8DFE54F07A37EC48F253283AC
       33C3F5102C8CC3E
   r = 3048E76506C5C43D92B2E33F62B33E3111CEEB87F6C7DF7C7C01E3CDA28FA5E8
       BE04B5B23AA03C0C70FEF8F723CBCEBFF0B7A52A3F5C8B84B741B4F6157E69A5
       FB0524B48F31828
   s = 2C99078CCFE5C82102B8D006E3703E020C46C87C75163A2CD839C885550BA5CB
       501AC282D29A1C26D26773B60FBE05AAB62BFA0BA32127563D42F7669C97784C
       8897C22CFB4B8FA

   With SHA-256, message = "test":
   k = 328E02CF07C7B5B6D3749D8302F1AE5BFAA8F239398459AF4A2C859C7727A812
       3A7FE9BE8B228413FC8DC0E9DE16AF3F8F43005107F9989A5D97A5C4455DA895
       E81336710A3FB2C
   r = 184BC808506E11A65D628B457FDA60952803C604CC7181B59BD25AEE1411A66D
       12A777F3A0DC99E1190C58D0037807A95E5080FA1B2E5CCAA37B50D401CFFC34
       17C005AEE963469
   s = 27280D45F81B19334DBDB07B7E63FE8F39AC7E9AE14DE1D2A6884D2101850289
       D70EE400F26ACA5E7D73F534A14568478E59D00594981ABE6A1BA18554C13EB5
       E03921E4DC98333

   With SHA-384, message = "test":
   k = 2A77E29EAD9E811A9FDA0284C14CDFA1D9F8FA712DA59D530A06CDE54187E250
       AD1D4FB5788161938B8DE049616399C5A56B0737C9564C9D4D845A4C6A7CDFCB
       FF0F01A82BE672E
   r = 319EE57912E7B0FAA1FBB145B0505849A89C6DB1EC06EA20A6A7EDE072A6268A
       F6FD9C809C7E422A5F33C6C3326EAD7402467DF3272A1B2726C1C20975950F0F
       50D8324578F13EC
   s = 2CF3EA27EADD0612DD2F96F46E89AB894B01A10DF985C5FC099CFFE0EA083EB4
       4BE682B08BFE405DAD5F37D0A2C59015BA41027E24B99F8F75A70B6B7385BF39
       BBEA02513EB880C
```

```
With SHA-512, message = "test":
k = 21CE6EE4A2C72C9F93BDB3B552F4A633B8C20C200F894F008643240184BE57BB
    282A1645E47FBBE131E899B4C61244EFC2486D88CDBD1DD4A65EBDD837019D02
    628D0DCD6ED8FB5
r = 2AA1888EAB05F7B00B6A784C4F7081D2C833D50794D9FEAF6E22B8BE728A2A90
    BFCABDC803162020AA629718295A1489EE7ED0ECB8AAA197B9BDFC49D18DDD78
    FC85A48F9715544
s = 0AA5371FE5CA671D6ED9665849C37F394FED85D51FEF72DA2B5F28EDFB2C6479
    CA63320C19596F5E1101988E2C619E302DD05112F47E8823040CE540CD3E90DC
    F41DBC461744EE9
```

A.3. Sample Code

We include here a sample implementation of deterministic DSA. It is meant for illustration purposes; for instance, this code makes no attempt at avoiding side-channel leakage of the private key. It is written in the Java programming language. The actual generation of the "random" value k is done in the computek() method. The Java virtual machine (JVM) is assumed to provide the implementation of the hash function and of HMAC.

```
// ================================================================

import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

/**
 * Deterministic DSA signature generation.  This is a sample
 * implementation designed to illustrate how deterministic DSA
 * chooses the pseudorandom value k when signing a given message.
 * This implementation was NOT optimized or hardened against
 * side-channel leaks.
 *
 * An instance is created with a hash function name, which must be
 * supported by the underlying Java virtual machine ("SHA-1" and
 * "SHA-256" should work everywhere).  The data to sign is input
 * through the {@code update()} methods.  The private key is set with
 * {@link #setPrivateKey}.  The signature is obtained by calling
 * {@link #sign}; alternatively, {@link #signHash} can be used to
 * sign some data that has been externally hashed.  The private key
 * MUST be set before generating the signature itself, but message
 * data can be input before setting the key.
 *
 * Instances are NOT thread-safe.  However, once a signature has
 * been generated, the same instance can be used again for another
 * signature; {@link #setPrivateKey} need not be called again if the
 * private key has not changed.  {@link #reset} can also be called to
 * cancel previously input data.  Generating a signature with {@link
 * #sign} (not {@link #signHash}) also implicitly causes a
 * reset.
 *
 * ----------------------------------------------------------------
 * Copyright (c) 2013 IETF Trust and the persons identified as
 * authors of the code.  All rights reserved.
 *
```

```
 * Redistribution and use in source and binary forms, with or without
 * modification, is permitted pursuant to, and subject to the license
 * terms contained in, the Simplified BSD License set forth in Section
 * 4.c of the IETF Trust's Legal Provisions Relating to IETF Documents
 * (http://trustee.ietf.org/license-info).
 *
 * Technical remarks and questions can be addressed to:
 * pornin@bolet.org
 * ----------------------------------------------------------------------
 */

public class DeterministicDSA {

        private String macName;
        private MessageDigest dig;
        private Mac hmac;
        private BigInteger p, q, g, x;
        private int qlen, rlen, rolen, holen;
        private byte[] bx;

        /**
         * Create an instance, using the specified hash function.
         * The name is used to obtain from the JVM an implementation
         * of the hash function and an implementation of HMAC.
         *
         * @param hashName   the hash function name
         * @throws IllegalArgumentException  on unsupported name
         */
        public DeterministicDSA(String hashName)
        {
                try {
                        dig = MessageDigest.getInstance(hashName);
                } catch (NoSuchAlgorithmException nsae) {
                        throw new IllegalArgumentException(nsae);
                }
                if (hashName.indexOf('-') < 0) {
                        macName = "Hmac" + hashName;
                } else {
                        StringBuilder sb = new StringBuilder();
                        sb.append("Hmac");
                        int n = hashName.length();
                        for (int i = 0; i < n; i ++) {
                                char c = hashName.charAt(i);
                                if (c != '-') {
                                        sb.append(c);
                                }
                        }
                        macName = sb.toString();
```

```
                }
                try {
                        hmac = Mac.getInstance(macName);
                } catch (NoSuchAlgorithmException nsae) {
                        throw new IllegalArgumentException(nsae);
                }
                holen = hmac.getMacLength();
        }

        /**
         * Set the private key.
         *
         * @param p   key parameter: field modulus
         * @param q   key parameter: subgroup order
         * @param g   key parameter: generator
         * @param x   private key
         */
        public void setPrivateKey(BigInteger p, BigInteger q,
                BigInteger g, BigInteger x)
        {
                /*
                 * Perform some basic sanity checks.  We do not
                 * check primality of p or q because that would
                 * be too expensive.
                 *
                 * We reject keys where q is longer than 999 bits,
                 * because it would complicate signature encoding.
                 * Normal DSA keys do not have a q longer than 256
                 * bits anyway.
                 */
                if (p == null || q == null || g == null || x == null
                        || p.signum() <= 0 || q.signum() <= 0
                        || g.signum() <= 0 || x.signum() <= 0
                        || x.compareTo(q) >= 0 || q.compareTo(p) >= 0
                        || q.bitLength() > 999
                        || g.compareTo(p) >= 0 || g.bitLength() == 1
                        || g.modPow(q, p).bitLength() != 1) {
                        throw new IllegalArgumentException(
                                "invalid DSA private key");
                }
                this.p = p;
                this.q = q;
                this.g = g;
                this.x = x;
                qlen = q.bitLength();
                if (q.signum() <= 0 || qlen < 8) {
                        throw new IllegalArgumentException(
                                "bad group order: " + q);
```

```
                }
                rolen = (qlen + 7) >>> 3;
                rlen = rolen * 8;

                /*
                 * Convert the private exponent (x) into a sequence
                 * of octets.
                 */
                bx = int2octets(x);
        }

        private BigInteger bits2int(byte[] in)
        {
                BigInteger v = new BigInteger(1, in);
                int vlen = in.length * 8;
                if (vlen > qlen) {
                        v = v.shiftRight(vlen - qlen);
                }
                return v;
        }

        private byte[] int2octets(BigInteger v)
        {
                byte[] out = v.toByteArray();
                if (out.length < rolen) {
                        byte[] out2 = new byte[rolen];
                        System.arraycopy(out, 0,
                                out2, rolen - out.length,
                                out.length);
                        return out2;
                } else if (out.length > rolen) {
                        byte[] out2 = new byte[rolen];
                        System.arraycopy(out, out.length - rolen,
                                out2, 0, rolen);
                        return out2;
                } else {
                        return out;
                }
        }

        private byte[] bits2octets(byte[] in)
        {
                BigInteger z1 = bits2int(in);
                BigInteger z2 = z1.subtract(q);
                return int2octets(z2.signum() < 0 ? z1 : z2);
        }

        /**
```

```
         * Set (or reset) the secret key used for HMAC.
         *
         * @param K   the new secret key
         */
        private void setHmacKey(byte[] K)
        {
                try {
                        hmac.init(new SecretKeySpec(K, macName));
                } catch (InvalidKeyException ike) {
                        throw new IllegalArgumentException(ike);
                }
        }

        /**
         * Compute the pseudorandom k for signature generation,
         * using the process specified for deterministic DSA.
         *
         * @param h1   the hashed message
         * @return  the pseudorandom k to use
         */
        private BigInteger computek(byte[] h1)
        {
                /*
                 * Convert hash value into an appropriately truncated
                 * and/or expanded sequence of octets.  The private
                 * key was already processed (into field bx[]).
                 */
                byte[] bh = bits2octets(h1);

                /*
                 * HMAC is always used with K as key.
                 * Whenever K is updated, we reset the
                 * current HMAC key.
                 */

                /* step b. */
                byte[] V = new byte[holen];
                for (int i = 0; i < holen; i ++) {
                        V[i] = 0x01;
                }

                /* step c. */
                byte[] K = new byte[holen];
                setHmacKey(K);

                /* step d. */
                hmac.update(V);
                hmac.update((byte)0x00);
```

```
hmac.update(bx);
hmac.update(bh);
K = hmac.doFinal();
setHmacKey(K);

/* step e. */
hmac.update(V);
V = hmac.doFinal();

/* step f. */
hmac.update(V);
hmac.update((byte)0x01);
hmac.update(bx);
hmac.update(bh);
K = hmac.doFinal();
setHmacKey(K);

/* step g. */
hmac.update(V);
V = hmac.doFinal();

/* step h. */
byte[] T = new byte[rolen];
for (;;) {
        /*
         * We want qlen bits, but we support only
         * hash functions with an output length
         * multiple of 8;acd hence, we will gather
         * rlen bits, i.e., rolen octets.
         */
        int toff = 0;
        while (toff < rolen) {
                hmac.update(V);
                V = hmac.doFinal();
                int cc = Math.min(V.length,
                        T.length - toff);
                System.arraycopy(V, 0, T, toff, cc);
                toff += cc;
        }
        BigInteger k = bits2int(T);
        if (k.signum() > 0 && k.compareTo(q) < 0) {
                return k;
        }

        /*
         * k is not in the proper range; update
         * K and V, and loop.
         */
```

```
                    hmac.update(V);
                    hmac.update((byte)0x00);
                    K = hmac.doFinal();
                    setHmacKey(K);
                    hmac.update(V);
                    V = hmac.doFinal();
            }
        }

        /**
         * Process one more byte of input data (message to sign).
         *
         * @param in   the extra input byte
         */
        public void update(byte in)
        {
                dig.update(in);
        }

        /**
         * Process some extra bytes of input data (message to sign).
         *
         * @param in   the extra input bytes
         */
        public void update(byte[] in)
        {
                dig.update(in, 0, in.length);
        }

        /**
         * Process some extra bytes of input data (message to sign).
         *
         * @param in    the extra input buffer
         * @param off   the extra input offset
         * @param len   the extra input length (in bytes)
         */
        public void update(byte[] in, int off, int len)
        {
                dig.update(in, off, len);
        }

        /**
         * Produce the signature.  {@link #setPrivateKey} MUST have
         * been called.  The signature is computed over the data
         * that was input through the {@code update*()} methods.
         * This engine is then reset (made ready for a new
         * signature generation).
         *
```

```
     * @return  the signature
     */
    public byte[] sign()
    {
            return signHash(dig.digest());
    }

    /**
     * Produce the signature.  {@link #setPrivateKey} MUST
     * have been called.  The signature is computed over the
     * provided hash value (data is assumed to have been hashed
     * externally).  The data that was input through the
     * {@code update*()} methods is ignored, but kept.
     *
     * If the hash output is longer than the subgroup order
     * (the length of q, in bits, denoted 'qlen'), then the
     * provided value {@code h1} can be truncated, provided that
     * at least qlen leading bits are preserved.  In other words,
     * bit values in {@code h1} beyond the first qlen bits are
     * ignored.
     *
     * @param h1   the hash value
     * @return  the signature
     */
    public byte[] signHash(byte[] h1)
    {
            if (p == null) {
                    throw new IllegalStateException(
                            "no private key set");
            }
            try {
                    BigInteger k = computek(h1);
                    BigInteger r = g.modPow(k, p).mod(q);
                    BigInteger s = k.modInverse(q).multiply(
                            bits2int(h1).add(x.multiply(r)))
                            .mod(q);

                    /*
                     * Signature encoding: ASN.1 SEQUENCE of
                     * two INTEGERs.  The conditions on q
                     * imply that the encoded version of r and
                     * s is no longer than 127 bytes for each,
                     * including DER tag and length.
                     */
                    byte[] br = r.toByteArray();
                    byte[] bs = s.toByteArray();
                    int ulen = br.length + bs.length + 4;
                    int slen = ulen + (ulen >= 128 ? 3 : 2);
```

```
                        byte[] sig = new byte[slen];
                        int i = 0;
                        sig[i ++] = 0x30;
                        if (ulen >= 128) {
                                sig[i ++] = (byte)0x81;
                                sig[i ++] = (byte)ulen;
                        } else {
                                sig[i ++] = (byte)ulen;
                        }
                        sig[i ++] = 0x02;
                        sig[i ++] = (byte)br.length;
                        System.arraycopy(br, 0, sig, i, br.length);
                        i += br.length;
                        sig[i ++] = 0x02;
                        sig[i ++] = (byte)bs.length;
                        System.arraycopy(bs, 0, sig, i, bs.length);
                        return sig;

                } catch (ArithmeticException ae) {
                        throw new IllegalArgumentException(
                                "DSA error (bad key ?)", ae);
                }
        }

        /**
         * Reset this engine.  Data input through the {@code
         * update*()} methods is discarded.  The current private key,
         * if one was set, is kept unchanged.
         */
        public void reset()
        {
                dig.reset();
        }
}

// ========================================================================
```

Author’s Address

Thomas Pornin
Quebec, QC
Canada

EMail: pornin@bolet.org