Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 15, 2013

M. Campagna
Certicom Corp.

October 12, 2012

A Cryptographic Suite for Embedded Systems (SuiteE)
draft-campagna-suitee-04

Abstract

This document describes a cryptographic suite of algorithms ideal for constrained embedded systems. It uses the existing IEEE 802.15.4 standard as a starting point, builds upon existing embedded cryptographic primitives and suggests additional elliptic curve cryptography (ECC) algorithms and curves. The goal is to define a complete suite of algorithms ideal for embedded systems.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 15, 2013.

Copyright Notice

described in the Simplified BSD License.

Table of Contents

1. Introduction

Constrained embedded systems and in particular devices for wireless personal and body area networks (WPAN and BAN respectively), have unique computation, power and bandwidth constraints. These systems are seeing wider deployment in Smart Energy, Home Automation, Personal Home and Health Care, and more broadly the so-called Internet of Things. The environments in which they are being deployed require varying degrees of security.

The Cryptographic Suite for Embedded Systems (SuiteE) aims to optimally meet the wide variety of cryptographic requirements, by providing a compact and complete collection of cryptographic algorithms having minimal code space, computational requirements and bandwidth usage. Additionally the selection of these algorithms are tuned to minimize overall system costs in mass production by selecting easily embeddable algorithms which will further reduce code space, energy usage and increase computational performance. It is expected that this suite of algorithms can be used to provide security solutions in the 6lowpan and CoRE space.

Mass production economics see many benefits of placing fixed routines in hardware. The benefits are in code space, performance, battery life, and overall cost of the device. This is the fundamental reason why most IEEE 802.15.4 devices implement AES in hardware today. Considering the projected scale of the so-called Internet of things (Cisco estimates the smart grid alone to be 100 to 1000 times the size of the Internet today), efficiencies and cost savings realized in embedding more of the lower level operations in hardware transforms into a basic requirement - technology selection should afford benefits to embedding in hardware.

Many of the environments in which these new embedded systems are being deployed have a life expectancy of 20+ years. This requires the selection of key lifecycle management mechanisms at a security level adequate to deliver the desired security services for the lifespan of the system. [NIST57] provides recommendations on general key management and security levels. We summarize the comparable strengths table and recommended minimum sizes:

Algorithm Lifetime	Security Strength	Symmetric Key Size	Integer Factorization Cryptography Key Size (e.g., RSA) Size	Elliptic Curve Cryptography (ECC) Key Size
Through 2010	80 bits	80	1024	160
Through 2030	112 bit	112	2048	256
Beyond 2030	128 bit	128	3072	256
>Beyond 2030	192 bit	192	7680	384
>>Beyond 2030	256 bit	256	15360	512

[NIST57] does not provide guidance on life span for security strengths for 192 and 256 bit presumably because of the uncertainty in forecasting technology 30+ years out.

Considering the expected life span of many of these systems and best industry practice we target the 128 bit security strengths for SuiteE.

The design goals of SuiteE are:

Provide re-usable primitives

Reduce code size

To be suitable for hardware implementation

Reduce computational costs

Reduce energy usage and increase battery lifespan

A complete cryptographic cipher suite should consist of primitives from which the security services of identification and authentication, confidentiality, data integrity and non-repudiation can be provided. We prescribe an encryption scheme with authentication, a deterministic random number generator, a hash

function, a key-agreement scheme, a digital signature scheme, and a certificate scheme that achieves a 128-bit security level, and achieves the goals identified above.

The remainder of this document is organized as follows. Section 2 provides an authenticated encryption mode AES-CCM*. Section 3 provides a deterministic random number generator. Section 4 describes a hashing algorithm using the existing AES core in an AES-MMO mode. Section 6 indicates why elliptic curve technology is selected and the specific curve selection sect283k1. Section 7 specifies the use of an elliptic curve signature scheme with partial message recovery. Section 8 provides an implicit certificate scheme. Section 9 describes an elliptic curve based mutual authenticated key agreement scheme.

2. Encryption

IEEE 802.15.4 [IEEE-802.15.4-2003] specifies the use of AES-CCM*, a variation of the Counter Mode with Cipher Block Chaining MAC (CBC-MAC) using AES-128. AES-128 is specified in [FIPS-197]. Using [IEEE-802.15.4-2003] as the normative reference we present a description of AES-CCM* here.

In the sections that follow we will assume that the basic block cipher is AES-128 as specified in [FIPS-197]. We will represent AES-128 as a function E taking two 128-bit inputs, a message block M, and key K, with 128-bit output C = E(M, K).

2.1. AES-CTR Mode

CTR mode or Counter Mode is a block cipher mode for providing confidentiality. It has some specific advantages in that both the encryption and decryption routines only require the block cipher to operate in a forward, or encrypt-only, direction.

Input: a 128-bit symmetric key K, and a plaintext message P of length Plen, and initial counter value CTR

1. Compute m = ceiling(Plen/128).

2. Apply the counter generator function to CTR to compute CTR_1, CTR_2, ..., CTR_m.

3. Compute S_j = E(CTR_j, K), for j = 1,...,m.

4. Let S = S_1||S_2||...||S_m, where || indicates concatenation.

5. Compute C = P XOR MSB_Plen(S), where MSB_X( ) takes the X most significant bits.

Output: C

The Decryption routine for CTR mode is symmetric.

Input: a 128-bit symmetric key K, and a ciphertext message C of length Clen, and initial counter value CTR

1. Compute m = ceiling(Clen/128).

2. Apply the counter generator function to CTR to compute CTR_1, CTR_2, ..., CTR_m.

3. Compute S_j = E(CTR_j, K), for j = 1,...,m.

4. Let S = S_1||S_2||...||S_m, where || indicates concatenation.

5. Compute P = C XOR MSB_Clen(S), where MSB_X( ) takes the X most significant bits.

Output: P

2.2. AES-CBC-MAC Mode

Cipher Block Chaining MAC Mode, CBC-MAC mode uses a block cipher to provide data integrity. Unlike CTR mode, that operates on arbitrary length strings CBC-MAC requires message padding to be on a multiple of the block length. The last message block will be padded out using zero bytes.

Input: a 128-bit symmetric key K, a message M of length Mlen

1. Form B by padding message M on the right with 0-bytes to be on a byte boundary of the block length (16-bytes).

2. Form B = B_1||B_2||...||B_m.

3. Set O_0 to a zeroized block-length byte string.

4. Compute O_j = E(O_j-1 XOR B_j, K).

5. Set MAC T = O_m.

Output: T

Verification is done by identical computation on input K, message M, and purported MAC T' and an additional check where the computed MAC value T is compared to the received MAC value T' and accepted only if T = T'.

2.3. AES-CCM* Mode

CCM mode is an authenticate and encrypt mode for block ciphers defined in [NIST-800-38C]. It is defined on a 128-bit block size block cipher. CCM* modifies this description to allow for modes that require only authentication, as well as variable length authentication tags.

2.3.1. AES-CCM* Encrypt

We break this section up into 3 subsections, input transformation, authentication transformation, and encryption transformation. We will assume that the following inputs are provided to the routines.

Input:

a. A 128-bit symmetric key K.

b. A value 1 < L < 9.

c. A nonce N of 15 - L octets, unique within the usage of the key K.

d. An octet string m of length l(m), where 0 <= l(m) < 2^8L.

e. An octet string a of length l(a), where 0 <= l(a) < 2^64.

2.3.1.1. Input Transformation

Input:

a. An octet string m of length l(m), where 0 <= l(m) < 2^8L.

b. An octet string a of length l(a), where 0 <= l(a) < 2^64.

1. Represent the length l(a) as an octet string L(a).

   a. If l(a) = 0, then L(a) is an empty string.

   b. If 0 < l(a) < 2^16 - 2^8, then L(a) is the 2-octet representation of l(a).

   c. If 2^16 - 2^8 <= l(a) < 2^32, then L(a) is the right-concatenation of the octet 0xff, the octet 0xfe, and the 4-octet encoding of l(a).

   d. If 2^32 <= l(a) < 2^64, then L(a) is the right-concatenation of the octet 0xff, the octet 0xff, and the 8-octet encoding of l(a).

2. Form AddAuthData = L(a)||a||0^t, where 0^t is the smallest non-negative string of t zero octets so that the resulting AddAuthData octet length is a multiple of 16.

3. Form PlaintextData = m || 0^s, where 0^s is the smallest non-negative string of s zero octets so that the resulting

PlaintextData octet length is a multiple of 16.

4. Form AuthData = AddAuthData || PlaintextData.

Output: AuthData

2.3.1.2. Authentication Transformation

Input:

a. A 128-bit symmetric key K.

b. The octet string AuthData, created in the input transformation.

c. A nonce N of 15 - L octets, unique within the usage of the key K.

d. The length l(m), where 0 <= l(m) < 2^8L.

1. Form the byte Flags = Reserved || Adata || M || L, where the 1-bit Reserved field is reserved for future expansions and shall be set to '0'. The 1-bit Adata field is set to '0' if l(a) = 0 and set to '1' if l(a) > 0. The M field is the 3-bit representation of the integer (M - 2)/2 if M > 0 and of the integer 0 if M = 0, in most-significant-bit-first order. The L field is the 3-bit representation of the integer L - 1, in most-significant-bit-first order.

2. Form B0 = Flags || Nonce N || l(m)

3. Parse the message AuthData as B1 || B2 || ... ||Bt, where each message block Bi is a 16-octet string.

4. The CBC-MAC value T = AES-CBC-MAC(B_0 || AuthData, K) as defined by Section 2.2.

Output: T

2.3.1.3. Encryption Transformation

Input:

a. A 128-bit symmetric key K.

b. PlaintextData from Section 2.3.1.1.

c. The authentication tag output T from Section 2.3.1.2.

d. A nonce N of length 15-L bytes.

1. Form Flags = Reserved0 || Reserved1 || 000 || L', where the reserved bits Reserved0 and Reserved1 is '0', and L' is the 3-bit representation of the integer L - 1.

2. Form the A_i = Flags || Nonce N || Counter_i, where Counter_i is an L-octet representation of the integer i = 0, 1, 2, ..., t.

3. Parse the PlaintextData from Section 2.3.1.1 into 16-octet blocks M_1, ..., M_t.

4. Compute C_i = E(A_i, K) XOR M_i for i = 1, ..., t.

5. Compute Ciphertext as the leftmost l(m) bits of C_1 || ... || C_t.

6. Compute S_0 = E(A_0, K) XOR T.

7. Compute AuthTag as the leftmost M octets of S_0.

Output: Ciphertext and AuthTag

2.3.2. AES-CCM* Decrypt

We break this section up into 2 subsections, decryption and authentication verification. The AES-CCM* Decryption process should both decrypt any encrypted portion of the message, and authenticate the decrypted message. It will return an error code, or plaintext data. We will assume that the following inputs are provided to the routines.

Input:

a. A 128-bit symmetric key K.

b. A nonce N of 15 - L octets, unique within the usage of the key K.

c. An M-octet tag AuthTag.

d. An octet string Ciphertext of length l(c), where 0 <= l(c) - M < 2^8L.

2.3.2.1. Decryption Transformation

Input:

a. A 128-bit symmetric key K.

b. A nonce N of 15 - L octets, unique within the usage of the key K.

c. An M-octet tag AuthTag.

d. An octet string Ciphertext of length l(c), where 0 <= l(c) - M < 2^8L.

1. Form C_0 from AuthTag by padding on the right by the least number of 0 octets so that C_0 is an octet string of length 16.

2. Form CiphertextData by right concatenation of C with the smallest number of 0 octets so the resulting string is of octet length divisible by 16.

3. Form Flags = Reserved0 || Reserved1 || 000 || L', where the reserved bits Reserved0 and Reserved1 are '0', and L' is the 3-bit representation of the integer L - 1.

4. Form the A_i = Flags || Nonce N || Counter_i, where Counter_i is an L-octet representation of the integer i = 0, 1, 2, ..., t.

5. Parse the C_0||CiphertextData into 16-octet blocks C_0, C_1, ..., C_t.

6. Compute P_i = E(A_i, K) XOR C_i for i = 0, ..., t.

7. Form U as the rightmost M-octets of P_0.

8. Form Plaintext leftmost l(c) octets of P_1||...||P_t.

Output: U and Plaintext

2.3.2.2. Verification Transformation

Input:

a. A 128-bit symmetric key K.

b. Plaintext of length l(c), where 0 <= l(c) < 2^8L.

c. U, the purported MAC of Plaintext.

d. A nonce N of 15 - L octets, unique within the usage of the key K.

1. Compute T as the output of the authentication transformation Section 2.3.1.2 with the inputs, key K, Plaintext, nonce N, and length value l(c).

2. Form T' as the leftmost M octets of T.

3. If T' = U output Plaintext, otherwise output error code INVALID.

Output: Plaintext or INVALID

3. Deterministic Random Number Generator

This section provides a reduced set of options to the CTR_DRBG definition using AES as defined in [NIST-800-90].

Restrict the CTR_DRBG to the use of AES-128 delivering 128-bit security.

General assumption of a full-entropy seed, removing the extra coding needed for the block_cipher_df and the BCC function as defined in [NIST-800-90]

The personalization_string and additional_input options are not supported.

Set maximum_number_of_bits_per_request = 2^16. (Based on convenient word boundary.)

Set reseed_interval = 2^48. (Based on maximizing life of a device that may not have an entropy source.)

We give a basic description of the AES block-cipher-based DRBG in the next few subsections. The description includes an update function, an initialization function, and a generate function. We will leave it up to implementers to consider other optional functions such as reseed.

We define the state of the CTR_DRBG using the following structure.

```
struct{
  uint64  counter;  //  64-bit counter
  uchar   V[16];    //  state
  uchar   K[16];    //  key
}ctr_drbg;
```

3.1. CTR_Update

The update function CTR_Update() modifies the internal state variables V and K of the DRBG structure using provided data.

Input:

a. The current state V.

b. The current key K.

c. A 32-byte input, data.

1. temp = AES(V+1 (mod 2^128), K) || AES(V+2 (mod 2^128), K).

2. temp = temp XOR data.

3. K = leftmost 16 bytes of temp.

4. V = rightmost 16 bytes of temp.

Output:

a. The updated state V.

b. The updated key K.

Functionally we write (V, K) = CTR_Update(V, K, data).

3.2. CTR_Init

The update function CTR_Update() modifies the internal state variables V and K of the DRBG structure using provided data.

Input:

a. A full entropy 32-byte seed.

1. Set V = 0^128, K = 0^128, zeroize state V, and K.

2. (V, K) = CTR_Update(V, K, seed).

3. Set counter = 1.

Output:

a. State (counter, V, K)

Functionally we write (counter V, K) = CTR_Init(seed).

NOTE: The CTR_Init() function can be simplified by making the observation that the resulting state is simply an XOR of the seed value with the fixed output values of AES(0^127||1_2, 0^128)|| AES(0^126||10_2, 0^128) = 58E2FCCEFA7E3061367F1D57A4E7455A0388DACE60B 6A392F328C2B971B2FE78_{16}, or (K||V) = 58E2FCCEFA7E3061367F1D57A4E7455A0388DACE60B6A392F328C2B971B2FE78 XOR seed.

3.3. CTR_Generate

The function CTR_Generate() modifies the internal state variables V and K of the DRBG structure, and generates a random output of the requested length rlen bytes. If the counter has exceeded 2^48 or rlen exceeds 2^16, an ERROR is returned, and the state is not modified.

Input:

a. Current state (counter, V, K).

b. Requested number of bytes, rlen.

1. If counter > 2^48 return ERROR.

2. If rlen > 2^16 return ERROR.

3. Set output = NULL

4. while(length(temp) < rlen)

   a. V = V + 1 (mod 2^128).

   b. output ||= AES(V, K).

5. output = leftmost rlen bytes of output.

6. (V, K) = CTR_Update(V, K, 0^256).

7. Set counter = counter + 1.

Output:

a. State (counter, V, K)

b. Either NULL and ERROR, or output and SUCCESS

Functionally we write ((counter V, K), (output, RESULT_CODE)) = CTR_Generate(V, K, data).

4. Hash

[ISO-10118-2] specifies hash functions using an n-bit block cipher. The first function from [ISO-10118-2] ("Hash-function one") maps arbitrary length inputs to n-bit outputs. This is the Matyas-Meyer-Oseas (MMO) construction described in [MOV96]. In the first subsection we specify a family of Merkle-Damgard strengthening (or MD-strengthening) functions that aims to account for existing deployments in ZigBee Smart Energy, and provide a gradual MD-strengthening that reduces padding on small messages.

The following subsection details an MMO construction that utilizes two input pre-processing steps. The first is a prefix-free encoding: the bitlength of the message encoded as a 128-bit integer is prepended to the input message. The second is the more common MD-strengthening transform, which essentially appends an encoding of the length and ensures the output is a multiple of the block length.

The hash function defined here is a not a general purpose hash function at the 128-bit security level, as it does not provide collision resistance. Application of this hash function should conform to the usages defined in this specification. The use of this hash function elsewhere requires careful consideration.

4.1. Prefix-Free Encoding

The prefix-free encoding step is defined as follows:

Input: message M of bitlength Mlen

Output: M' of bitlength Mlen + 128

1. If Mlen >= 2^64 return ERROR.

2. Convert Mlen to a bit string Mblen of length 128 bits, where the first 64 where Mblen is the 128-bit little-endian integer representation of Mlen. (Note that the leftmost 64 bits will always be zero.)

3. Prepend (left-pad) and output Mblen to M, i.e., output M' = Mblen||M.

4.2. MD-strengthening

This section defines an escalating MD-strengthening padding scheme to account for minimizing additional blocks for small messages, and expanding length encoding for larger messages. Additionally it allows for compliance with existing applications of MMO hashing

defined in ZigBee Smart Energy 1.0. For messages less than 2^16 bits we use MD-strengthening-1, for messages of length less than 2^32 bits but greater than 2^16 - 1 bits we use MD-strengthening-2, and for messages of length less than 2^64 bits but greater than 2^32 - 1 bits we use MD-strengthening-3

4.2.1. MD-strengthening-1

Input: A message M of bitlength Mlen.

1. If Mlen >= 2^16 return ERROR.

2. Pad the message M:

   a. Right concatenate the message M with a '1' bit followed by k '0' bits where k is the least non-negative number such that Mlen + 1 + k = 112 mod 128.

   b. Form the padded message M' by right concatenating the resulting string with a 16-bit string that is equal to the value Mlen.

Output: M' = M_1, M_2, ..., M_m a padded message in 128-bit blocks.

4.2.2. MD-strengthening-2

Input: A message M of bitlength Mlen.

1. If Mlen < 2^16 or Mlen >= 2^32 return ERROR.

2. Pad the message M:

   a. Right concatenate the message M with a '1' bit followed by k '0' bits where k is the least non-negative number such that Mlen + 1 + k = 80 mod 128.

   b. Form the padded message M'' by right concatenating the resulting string with a 32-bit string that is equal the value Mlen.

   c. Form the padded message M' by right concatenating M'' with a 16-bit 0 string.

Output: M' = M_1, M_2, ..., M_m a padded message in 128-bit blocks.

4.2.3. MD-strengthening-3

Input: A message M of bitlength Mlen.

1. If Mlen < 2^32 or Mlen= >= 2^64 return ERROR.

2. Pad the message M:

   a. Right concatenate the message M with a '1' bit followed by k '0' bits where k is the least non-negative number such that Mlen + 1 + k = 16 mod 128.

   b. Form the padded message M'' by right concatenating the resulting string with a 64-bit string that is equal the value Mlen.

   c. Form the padded message M' by right concatenating M'' with a 48-bit 0 string.

Output: M' = M_1, M_2, ..., M_m a padded message in 128-bit blocks.

4.3. MMO Construction

We describe the basic MMO construction on a message M that has been padded and MD-strengthened to be of length a multiple of the bit length of AES, 128 bits.

Input: A message M of bitlength Mlen.

1. Apply the prefix-free encoding to M to form a bitstring M', with bitlength Mlen' = Mlen + 128.

2. Pad and format M' in 128-bit blocks using one of the MD routines specified above or return ERROR.

3. Set H_0 = 0^128, a zeroized 128-bit block.

4. Compute H_j = E(M_j, H_(j-1)) XOR M_j, for j = 1, ..., m.

Output: H = H_m as the hash value.

5. Key Derivation Function

At this time SuiteE does not recommend a particular key derivation function (KDF) for compliance. That said, some of the SuiteE primitives do require use of a KDF. One possible choice are the pseudorandom function-based constructions of KDFs given in [NIST-800-108] which are suitable, and may be instantiated with AES-CMAC as a pseudo-random function.

6. Curve Selection

We will assume basic familiarity with Elliptic Curve Cryptography (ECC) notation and specifications. Wherever possible we will refer to freely available standards and indicate alignment of these standards with other well known standards. We will assume the notation in the Standards for Efficient Cryptography Group SEC 1, [SEC1]. This specification defines the basic ECC operations as well as the most widely standardized and used techniques. The overall goal of [SEC1] is to provide a free standard that ensures conformance with other standard specifications, and is instructive to implementers on how to develop the necessary routines to build an elliptic curve cryptosystem.

Elliptic curve cryptography is a natural public key technology to select given a goal of providing public key operations at a 128-bit security level on embedded systems. ECC, like RSA and other public key cryptosystems provides security by the use of key pairs: a private key and a public key. Security properties are built upon the assumption that the private key is securely generated and maintained within the confines of a cryptographic boundary. Ideally, this means that private keys should be generated on the device in which the private keys will be used operationally.

RSA key generation at the 128-bit security level requires the generation 1536-bit prime numbers, and performing integer operations on 3072-bit numbers. The inability of embedded systems to generate these keys on devices today, and the future requirements to implement this in hardware makes RSA a poor choice.

ECC key generation at the 128-bit security level requires the generation of a 256-bit random number and scalar point multiplication, which can be done efficiently in embedded systems today.

Elliptic curves over binary fields have a distinct advantage over elliptic curves over prime fields in that they are more efficient and cost effective in hardware. The binary curve sect283k1 as specified in [SEC2] is the most widely standardized and specified binary curve at the 128-bit security level. It is a Koblitz curve, which has an advantage over random curves in performing point multiplication algorithms.

For a basic description of the algorithms the reader is referred to [SEC1]. For a more complete background and optimizations for elliptic curves the reader is referred to [HMV04].

Throughout the following sections on elliptic curve cryptography we

will use the following notation

Fq - is a finite field with q elements.

E - an elliptic curve over Fq, given by the equation y^2 + xy = x^3 + ax^2 + b.

E(Fq) - the group of elliptic curve points over Fq.

G - is a base point in E(Fq) of a large prime order.

n - is a large prime, and the order of the subgroup generated by G.

h - is the cofactor, h = #E(Fq)/n.

Further we will assume it is known how to generate elliptic curve key pairs, and will refer the reader to the freely available specification [SEC1]. In general a key pair is generated by taking a random integer d, 1 < d < n, and computing Q = dG, the addition of the point G to itself d times. In general all parties will have some assurances that the elliptic curve domain parameters are valid, and in particular are fixed to the selected sect283k1 curve parameters.

6.1. SECT283K1 Curve Parameters

Fq - is a finite field with q = 2^283. Fq = F_2[X]/(f(x)), where f(x) = x^283 + x^12 + x^7 + x^5 + 1.

E - the elliptic curve is defined as E:y^2 + xy = x^3 + b.

G - is a base point in E of a large prime order, presented here in uncompressed octet encoding 04 0503213F 78CA4488 3F1A3B81 62F188E5 53CD265F 23C1567A 16876913 B0C2AC24 58492836 01CCDA38 0F1C9E31 8D90F95D 07E5426F E87E45C0 E8184698 E4596236 4E341161 77DD2259.

n - is a large prime, and the order of the subgroup generated by G, n = 01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFE9AE 2ED07577 265DFF7F 94451E06 1E163C61.

h - is the cofactor, #E(Fq)/n, h = 4.

7. Elliptic Curve Signatures with Partial Message Recovery

Elliptic Curve Pintsov-Vanstone Signatures (ECPVS) is an elliptic curve variant of Nyberg-Rueppel signatures. ECPVS is standardized in [X9.92.1] and [IEEE1363-A]. It has three distinct advantages over ECDSA when it comes to constrained environments. The first being that it allows for smaller signature sizes by the incorporation of part of the message into a signature field. The second is a simplification of the integer arithmetic. The signing transformation does not require a modular inverse, improving both code size and computational performance. The verification transformation requires no integer arithmetic and so also removes a modular inverse and modular multiplies in the verification transformation. The third is that it is a Schnorr signature scheme which loosens the collision resistance requirement on the underlying hash function [NSW07]. Thus the AES-MMO hash primitive in SuiteE (see Section Section 4)is sufficient for security. A second consequence is a performance increase in the signature verification, where a scalar multiply with a 256-bit integer (ECDSA) is replaced by a 128-bit integer (ECPVS).

We assume all parties possess the domain parameters for the elliptic curve, as chosen in Section 6, and that the public keys of signers are validated as described in [SEC1], Section 3.2.2.

The scheme presented here is a variant of ECPVS as presented in [X9.92.1], it removes the redundancy requirement on the input message by replacing the use of symmetric key encryption with the authenticated encryption functionality of SuiteE, AES-CCM*.

ECPVS uses encoding and decoding routines to process signatures.

7.1. Message Encoding and Decoding Methods

The following subsections specify the encoding and decoding operation primitives that shall be used to generate ECPVS signatures and to verify ECPVS signatures.

7.1.1. Message Encoding

Input: The input to the encoding operation is:

a. A recoverable message part, which is a bit string M of length Mlen bits.

b. A bit string Z (used to derive a key).

Steps

1. Form T = 00||M, by prepending a zero byte to the front of M.

2. Apply the key derivation function KDF to the bit string Z to produce a 128-bit key K.

3. Let (C, MAC) = AES-CCM*-ENCRYPT(T, K, 128).

4. Form r = C||MAC, the concatenation of the cipher text value C with the computed MAC value.

Output: r.

We will denote this process as r = EM(M, Z), for encode message.

7.1.2. Decoding Messages

Input: The input to the decoding operation is:

a. The message representative, which is a bit string r of length rlen bits.

b. A bit string Z.

Steps

1. Apply the key derivation function KDF to the bit string Z to produce a 128-bit key K.

2. Parse the message r = C||MAC, where MAC is the last 16 bytes, or return ERROR.

3. Compute M' = AES-CCM*-DECRYPT(C, MAC, K, 128).

4. If M' is ERROR value return ERROR.

5. Parse M' = 00||M, where 00 is a zeroized byte, or return ERROR.

6. return M.

Output: ERROR or message value M.

We will denote this process as M = EM^-1(r, Z), for decode message.

7.2. Elliptic Curve Pintsov-Vanstone Signatures (ECPVS)

This section contains two subsections, signature generation and signature verification.

7.2.1. Signature Generation

Input: The input to the signature generation transformation is:

a. A message (M, V), which is a pair of bit strings to be signed. M is the recoverable portion of the message, and V is the visible or plaintext portion of the message.

b. An elliptic curve private key d.

Steps:

1. Generate an ephemeral key pair (k, R) with R = (x, y). (see: [SEC1] Section 3.2.1).

2. Convert the field element x to a bit string Z. (see: [SEC1] Section 2.3.5 and 2.3.2).

3. Encode the recoverable portion of the message r = EM(M, Z).

4. Compute H = AES-MMO(r||V), where || represents concatenation.

5. Convert the bit string H to an integer e. (see: [SEC1] Section 2.3.1 and 2.3.8).

6. Compute s = k - de (mod n).

Output: (r, s) as the signature with partial message recovery on V.

7.2.2. Signature Verification

Input: The input to verification is:

a. A message V.

b. A signature with partial message recovery (r, s).

c. The elliptic curve public key Q belonging to the signer.

Steps:

1. If s is not an integer in the interval [1, n - 1], return ERROR.

2. Compute H = AES-MMO(r||V).

3. Convert the bit string H to an integer e. (see: [SEC1] Section 2.3.1 and 2.3.8).

4. Compute the elliptic curve point R = sG + eQ. Let x denote the x-coordinate of R.

5. Convert the field element x to a bit string Z. (see: [SEC1] Section 2.3.5 and 2.3.2).

6. Use message decoding to compute: M = EM^-1(r, Z).

7. If M is an ERROR value, return ERROR, else return M and VALID.

Output: Either an ERROR, or a recovered message M and indication of a VALID signature.

8. Elliptic Curve Implicit Certificates (ECQV)

In this section we specify the Elliptic Curve Qu-Vanstone implicit certificate scheme (ECQV).

A traditional certificate scheme consists of a certificate issuer with a key pair (d, Q) to produce a triplet binding an identity, I, and a public key, P, using a signature, sig, by invoking the issuer's private key d. We can represent this certificate as (I, P, sig). An implicit certificate binds an identity element with a public key without an explicit signature by creating a pair of elements; an identity, I, and a public key reconstruction value B. We can represent this certificate as (I, B).

Verification of the certificate is implicit in the use of the public key. The public key is bound to the entity identified in the implicit certificate, and the certification process ensures that only this entity may recover the associated secret key. In effect, certificate verification is combined with the cryptographic primitive for which the signed key is being used. This provides two advantages over traditional certificates. The first is a cost savings on memory and bandwidth. Implicit certificates do not require an explicit signature, reducing the size of the cryptographic material to roughly 1/3rd of a traditional certificate. The second is on computation, the public key reconstruction operation is computationally more efficient than a signature verification, and may be combined with other operations.

The next five sections describe an implicit certificate scheme, defined in [SEC4]. As in the previous section the elliptic curve domain parameters and hash functions are assumed. It is also assumed that the CA has established a key pair (dCA, QCA), and all communicating parties have access to the public key QCA.

We will assume that certificate validation routines have been established in regards to verifying certificate validity, key usage and certificate formatting. These checks will be assumed in steps in which certificate parsing takes place where possible error values may be returned.

8.1. Certificate Request

This section describes the basic operations by which an entity A generates a certificate request. It is assumed that A and CA have an authenticated channel, but the channel may not be private.

Input: none

Steps:

1. Generate an elliptic curve key pair (kA, RA).(see: [SEC1] Section 3.2.1).

Output: a key pair (kA, RA).

The entity A, requesting the certificate, sends the public key RA along with purported identity of A to the Certificate Authority, and stores (kA, RA) for future use, keeping kA secret.

8.2. Certificate Generation

This section describes the certificate generation process executed by a certificate authority. It is assumed that A and CA have an authenticated channel, but the channel may not be private.

Input: the following items or equivalent

a. The CA private key dCA.

b. A certificate request consisting of a public key RA and an identifier for A.

Steps:

1. Validate the public key value RA, if it fails output ERROR. (see: [SEC1] Section 3.2.2).

2. Generate an elliptic curve key pair (k, kG) (see: [SEC1] Section 3.2.1).

3. Compute the elliptic curve point BA = RA + kG.

4. Convert BA to an octet string BAS (see: [SEC1] Section 2.3.3).

5. Encode the certificate CertA as an octet string consisting of BAS, and the identity element I, consisting of the identifier for A and other certificate validity and key usage information.

6. Compute H = AES-MMO(CertA).

7. Convert the bit string H to an integer e (see: [SEC1] Section 2.3.1 and 2.3.8).

8. Compute the integer r = ek + dCA (mod n).

Output: Either an implicit certificate CertA, and private key

contribution value r, or an ERROR value.

8.3. Certificate Reception

This section describes the certificate reception process by which a certificate requester computes its private key.

Input: the following items or equivalent

a. The CA public key QCA.

b. The key pair generated during the certificate request (kA, RA).

c. The implicit certificate CertA.

d. The private key contribution value r.

Steps:

1. Parse the certificate CertA into an octet string BAS and an identity element I or return ERROR.

2. Convert BAS to a public key BA or return ERROR (see: [SEC1] Section 2.3.4).

3. Validate the public key value BA or return ERROR (see: [SEC1] Section 3.2.2).

4. Compute H = AES-MMO(CertA).

5. Convert the bit string H to an integer e (see: [SEC1] Section 2.3.1 and 2.3.8).

6. Compute the private key dA = r + ekA (mod n).

7. Compute QA = eBA + QCA.

8. Verify QA = (dA)G otherwise halt and return ERROR.

Output: Either a key pair (dA, QA), or an ERROR value.

8.4. Certificate Public Key Extraction

This section describes the certificate public key reconstruction operation. This would be done by any entity desiring to authenticate a cryptographic operation with entity A.

Input: the following items or equivalent

a. The CA public key QCA.

b. The implicit certificate CertA.

Steps:

1. Parse the certificate CertA into an octet string BAS and an identity element I or return ERROR.

2. Convert BAS to a public key BA or return ERROR.(see: [SEC1] Section 2.3.4).

3. Validate the public key value BA or return ERROR.(see: [SEC1] Section 3.2.2).

4. Compute H = AES-MMO(CertA).

5. Convert the bit string H to an integer e. (see: [SEC1] Section 2.3.1 and 2.3.8).

6. Compute QA = eBA + QCA.

Output: Either an alleged public key QA, or an ERROR value.

We say the output is "alleged" since the party that extracted it does not at this point know that it belongs to the party identified in CertA. However, the scheme does guarantee that only the identified party possesses the secret key associated to QA.

9. Elliptic Curve Key Agreement Scheme

The elliptic curve MQV, or ECMQV scheme is a key agreement scheme based on ECC. We give a description here, but will refer to [SEC1] as the definitive normative reference. ECMQV can provide a variety of security properties and we refer the reader to [NIST-800-56] to use an instantiation of ECMQV that best satisfies their application security goals. We select ECMQV for SuiteE because of its well studied security properties, wide standardization, existing deployment in the constrained environment space and the computational and bandwidth savings it can provide over competing methods to provide an authenticated key agreement scheme.

As in previous ECC sections we will assume all parties have agreed upon and access to validated elliptic curve parameters, and for the purpose of SuiteE this is the sect283k1 curve as defined in [SEC2].

9.1. ECMQV

This section defines the basic operations of the ECMQV primitive. We will assume the two communicating parties A and B have established two key pairs (dA1, QA1) and (dA2, QA2), and (dB1, QB1) and (dB2, QB2) respectively. The description is presented from A's reference point, B would perform the same operations with analogous key pairs. The first of each public key may be a static public key derived from an implicit certificate and used to authenticate the entity.

Input: The following or equivalent

a. Two key pairs (dA1, QA1) and (dA2, QA2).

b. Two partially validated public keys QB1, QB2 purportedly owned by B.

c. A key derivation function KDF, and an agreed upon key data length klength.

d. [Optional] shared information, SHARED_INFO for the KDF.

Steps:

1. Compute an integer QA2bar from QA2:

    a. Convert the x-coordinate of QA2 to an integer x.

    b. Set xbar = x (mod 2^(ceiling(log_2(n)/2)). (For SuiteE, xbar = x mod 2^142.)

c. Set QA2bar = xbar + 2^(ceiling(log_2(n)/2). (For SuiteE QA2bar = xbar + 2^142.)

2. Compute the integer s = dA2 + QA2bar*dA2 (mod n)

3. Compute an integer QB2bar from QB2:

   a. Convert the x-coordinate of QB2 to an integer x'.

   b. Set xbar' = x' (mod 2^(ceiling(log_2(n)/2)). (For SuiteE, xbar' = x' mod 2^142.)

   c. Set QB2bar = xbar' + 2^(ceiling(log_2(n)/2). (For SuiteE QB2bar = xbar' + 2^142.)

4. Compute the point P = h*s(QB2 + QB2bar*QB1), where h is the cofactor defined as #E(Fq)/n, which is 4 for the SuiteE curve sect283k1.

5. If P = point at infinity output ERROR and exit.

6. Convert the x-coordinate of P to an octet string Z.

7. Use the KDF function with input Z, klength, and the optional SHARED_INFO, to generate either the shared key value K or return an ERROR.

Output: Shared secret key K of klength, or an ERROR value.

10. References

10.1. Normative References

[FIPS-180-3]
National Institute of Standards and Technology, "Secure Hash Standard", FIPS 180-3, October 2008, <http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf>.

[FIPS-197]
National Institute of Standards and Technology, "Advanced Encryption Standard", FIPS 197, November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[IEEE-802.15.4-2003]
IEEE Computer Society, "IEEE Standard for Information technology -- Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)", IEEE Standard for Information technology 802.15.4, October 2003.

[ISO-10118-2]
International Organization for Standards and the International Electrotechnical Commission, "ISO/IEC 10118-2, Information technology - Security techniques - Hash functions - Part 2: Hash-functions using an n-bit block cipher", Information technology - Security techniques 10118-2, December 2000.

[NIST-800-108]
National Institute of Standards and Technology, "NIST SP 800-108, Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, October 2009, <http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf>.

[NIST-800-38C]
National Institute of Standards and Technology, "NIST SP 800-38C, Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality", NIST Special Publication 800-38C, July 2007, <http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf>.

[NIST-800-56]
National Institute of Standards and Technology, "Recommendation for Pair-wise Key Establishment Schemes

Using Discrete Logarithm Cryptography (Revised)", NIST Special Publication 800-56a, March 2007, <http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf>.

[NIST-800-57] National Institute of Standards and Technology, "Recommendation for Key Management - Part 1: General (Revised)", NIST Special Publication 800-57, March 2007, <http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf>.

[NIST-800-90] National Institute of Standards and Technology, "NIST SP 800-90, Recommendation for Random Number Generation Using Deterministic Random Bit Generators(Revised)", NIST Special Publication 800-90, March 2007, <http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf>.

[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.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, September 2003.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, April 2004.

[SEC1] Standards for Efficient Cryptography Group, "SEC1: Elliptic Curve Cryptography", SEC 1, May 2009, <http://www.secg.org/download/aid-780/sec1-v2.pdf>.

[SEC2] Standards for Efficient Cryptography Group, "SEC2: Recommended Elliptic Curve Domain Parameters", SEC 2, September 2010, <http://www.secg.org/download/aid-784/sec2-v2.pdf>.

[SEC4] Standards for Efficient Cryptography Group, "Elliptic Curve Qu-Vanstone Implicit Certificate Scheme (ECQV), v0.97", SEC 4, March 2011, <http://www.secg.org/download/aid-785/sec4-0.97.pdf>.

[X9.92.1] Accredited Standards Committee X9, Inc., "Public Key Cryptography for the Financial Services Industry - Digital Signature Algorithms Giving Partial Message Recovery - Part 1: Elliptic Curve Pintsov-Vanstone Signatures (ECPVS)", X9 92-1, 2009, <http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.92-1-2009>.

[ZigBee] ZigBee Standards Organization, "ZigBee Specification, revision 17", October 2007, <http://www.zigbee.org/ZigBeeSpecificationDownloadRequest/tabid/311/Default.aspx>.

Registration required.

10.2. Informative References

[HMV04] Hankerson, D., Menezes, A., and S. Vanstone, "Guide to Elliptic Curve Cryptography", 2004.

Springer, ISBN 038795273X.

[IEEE1363] Institute of Electrical and Electronics Engineers, "Standard Specifications for Public Key Cryptography", IEEE 1363, 2000.

[IEEE1363-A] Institute of Electrical and Electronics Engineers, "Standard Specifications for Public Key Cryptography - Amendment 1: Additional Techniques", IEEE 1363a, 2004.

[LMQSV98] Law, L., Menezes, A., Qu, M., Solinas, J., and S. Vanstone, "An Efficient Protocol for Authenticated Key Agreement", University of Waterloo Technical Report CORR 98-05, August 1998, <http://www.cacr.math.uwaterloo.ca/techreports/1998/corr98-05.pdf>.

[MOV96] Menezes, A., van Oorschot, P., and S. Vanstone, "Handbook of Applied Cryptography", 1996, <http://www.cacr.math.uwaterloo.ca/hac>.

CRC Press, ISBN 0-8493-8523-7. Available online.

[NIST57] Barker, E., Barker, W., Burr, W., Polk, W., and M. Smid, "Recommendation for Key Management - Part 1: General (revised)", March 2007, <http://csrc.nist.gov/publications/nistpubs/800-57/

sp800-57-Part1-revised2_Mar08-2007.pdf>.

NIST Special Publication 800-57

[NSW07] Neven, G., Smart, N., and B. Warinschi, "Hash function requirements for Schnorr signatures", Journal of Mathematical Cryptology Volume 3 Number 1, May 2009.

[X9.123] Accredited Standards Committee X9, Inc., "Elliptic Curve Qu-Vanstone Implicit Certificates (Draft)", X9 123, 2011.

[ZigBeeSE] ZigBee Standards Organization, "ZigBee Smart Energy Profile Specification, revision 15", December 2008, <http://www.zigbee.org/ZigBeeSmartEnergyPublicApplicationProfile/tabid/312/Default.aspx>.

Registration required.

Appendix A. Acknowledgments

Author's Address

Matthew Campagna
Certicom Corp.
4701 Tahoe Boulevard
Mississauga, Ontario L4W 0B5
Canada

Email: mcampagna@certicom.com