The PASSERINE Public Key Encryption and Authentication Mechanism

Markku-Juhani O. Saarinen

Aalto University
Department of Communications and Networking
P.O.Box 13000, 00076 Aalto, FINLAND
m.saarinen@tkk.fi

Abstract. PASSERINE[1] is a lightweight public key encryption mechanism which is based on a hybrid, randomized variant of the Rabin public key encryption scheme. Its design is targeted for extremely low-resource applications such as wireless sensor networks, RFID tags, embedded systems, and smart cards. As is the case with the Rabin scheme, the security of PASSERINE can be shown to be equivalent to factoring the public modulus. On most low-resource implementation platforms PASSERINE offers smaller transmission latency, hardware and software footprint and better encryption speed when compared to RSA or Elliptic Curve Cryptography. This is mainly due to the fact that PASSERINE implementations can avoid expensive big integer arithmetic in favor of a fully parallelizable CRT randomized-square operation. In order to reduce latency and memory requirements, PASSERINE uses Naccache-Shamir randomized multiplication, which is implemented with a system of simultaneous congruences modulo small coprime numbers. The PASSERINE private key operation is of comparable computational complexity to the RSA private key operation. The private key operation is typically performed by a computationally superior recipient such as a base station. The PASSERINE project is entirely open source (hardware and software).

Keywords: Rabin Cryptosystem, Randomized Multiplication, RFID, Wireless Sensor Networks.

1 Introduction

Public key encryption is often viewed as unimplementable for extremely low-resource devices such as sensor network nodes and RFID tags. However, public key cryptography offers clear security advantages as fixed secret keys do not have to be shared between the two communicating parties. The PASSERINE public key encryption operation is very light, but the private key operation is approximately as computationally demanding as the private key operation of RSA.

1.1 Usage Scenarios

We will now describe two typical scenarios where it suffices that a lightweight device is able to perform only the public key operation.

[1] Version: August 10, 2010. This is work in progress.

Authentication and establishing a secure session. In case of authentication, the main attack scenario is that an adversary eavesdrops on the communication link between the authenticating station and the lightweight device. If the device leaks its authentication code, then the device can be easily cloned.

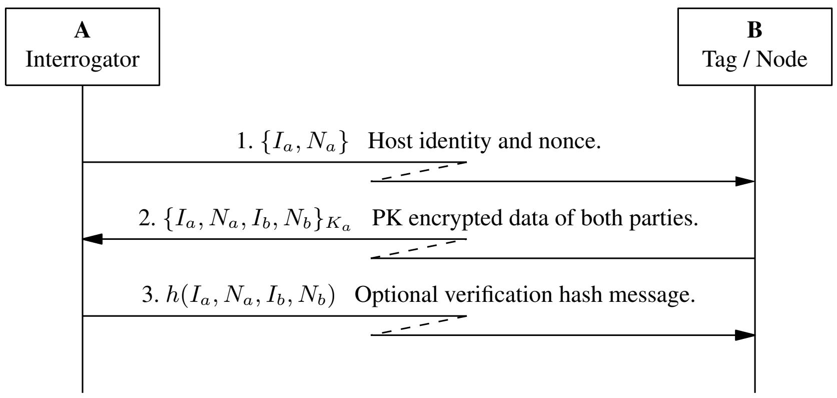
A
Interrogator
B
Tag / Node
1. $\{I_a, N_a\}$ Host identity and nonce.
2. $\{I_a, N_a, I_b, N_b\}_{K_a}$ PK encrypted data of both parties.
3. $h(I_a, N_a, I_b, N_b)$ Optional verification hash message.

Fig. 1. A Simple Authentication Protocol.

Figure 1 illustrates a simple three-message authentication protocol. We use a notation similar to that of Lowe [11]. In Step 1, Alice the Interrogator initializes the protocol by sending her identity I_a (an identifier that can be mapped to Alice's public key K_a) and a random nonce N_a to Bob (RFID Tag, sensor node or other light-weight device). Bob responds in Step 2 by encrypting Alice's information I_a and N_a together with his own secret identifier I_b and nonce N_b with Alice's public key K_a. Alice then decrypts this data using her private key K_a^{-1} and checks the nonce N_a and identity I_a. If successful, Alice may conclude the protocol by sending a hash (or a MAC such as an AES-CBC-MAC) of all identity and nonce data back to Bob (Step 3). This is optional, depending on the application. The secret nonce N_b can then be used to establish a secure communication link between the two parties, if so desired.

We note that this protocol requires Bob to perform public key encryption, but no private key operations. The protocol does not leak the secret identity I_b of the tag (typically a name and a password), even in a man-in-the-middle attack. The protocol also allows Alice to store only a salted hash of the password in I_b, rather than the password itself. Most importantly this limits the security implications if a single Alice interrogator station has been hacked or physically compromised.

Wireless sensor networks. In a military application a large number of sensors may be dispersed to an area of operations to lay passively dormant until an a particular combination of events triggers their activation. In such a scenario, key management with symmetric-only encryption may become exceedingly difficult. A single captured and reverse-engineered sensor unit may reveal all shared keys that it contains, possibly compromising the entire sensor network. Use of public-key cryptography simplifies key management and also reduces the need to protect keying information contained in the node. Each node only needs to store its unique identifier and the public key of the secure receiving station. The adversary can only to impersonate a single physically captured sensor unit.

In this scenario the devices are controlled by a base station that stores their private identifiers. The devices only need to be able to perform the public key operation - to broadcast messages to the base station. A sensor unit can securely authenticate an another node with the aid of the trusted base station.

1.2 Previous Work

The use of Rabin encryption in low-resource platforms has been investigated by Shamir [20], Gaubatz et al. [7, 8] and more recently by Oren and Feldhofer [17]. The approaches considered in these papers differ significantly from PASSERINE; Gaubatz et al. do not consider randomized multiplication but only bit-serial multiplication. Shamir, Oren and Feldhofer use randomized multiplication but not CRT arithmetic nor payload encoding into the random mask. Systems described in [17, 20] require substantial amounts of real randomness, which may be difficult to generate in a resource-strained device. PASSERINE requires only a single random 128-bit key for each message. Naccache et al. [13] use randomized multiplication and CRT arithmetic (which they call Brugia-di Porto-Filipponi number system after [4]) in a low-resource implementation of a related identification protocol which was subsequently broken in [5].

1.3 Section Breakdown

Section 2 describes the basic tricks used by PASSERINE: Naccache-Shamir randomized multiplication, CRT arithmetic, and message encoding in the randomization mask. Our very compact prototype PASSERINE-AES implementation is described in Section 3. This is followed by a discussion on security considerations in Section 4 and our conclusions and ongoing further work in Section 5.

2 The PASSERINE Randomized Rabin Cryptosystem

Rabin's public key cryptosystem [18] is in many ways similar to the RSA cryptosystem. Let n be a product of two large primes p and q. In order to facilitate implementation, these primes are often chosen so that $p \equiv q \equiv 3 \pmod 4$. To encrypt a message x, one simply squares it modulo the public modulus n:

$$z = x^2 \pmod n. \tag{1}$$

The Rabin private key operation requires computation of modular square roots and is of comparable complexity to the RSA private key algorithm. Since there are a total of four possible square roots ($\sqrt{z} \equiv \pm x \mod p$ and $\sqrt{z} \equiv \pm x \mod q$), a special mechanism is required in to mark and find the correct root. We refer to standard cryptography textbooks such as [10] for a discussion about implementation options.

The main distinguishing factor for the Rabin cryptosystem, in addition to being slightly faster than RSA in encryption, is that it is provably as secure as factoring. This equivalence may or may not hold for RSA [1, 3].

2.1 Shamir's Randomized Variant

In Eurocrypt '94 [20] Shamir proposed a randomized variant of the Rabin cryptosystem that avoids arithmetic $\mathrm{mod}\ n$ by using a random masking variable $r > n$. The encryption operation is

$$z = x^2 + r \cdot n. \tag{2}$$

The private key operation is essentially the same as with the standard Rabin scheme.

Randomized multiplication was originally considered by Naccache [12], albeit for a different application. Shamir proved that this randomized variant has equivalent security properties to the standard version. The main drawback from avoiding modular arithmetic is that the ciphertext roughly doubles in size and that a large amount of high quality random bits must be generated for r. We avoid this problem using an encoding technique described in Section 2.3.

2.2 Arithmetic Modulo a Set of Coprime Numbers

A large majority of the implementation footprint of traditional public key encryption schemes such as RSA or ECC tends to be consumed by implementing large finite field multiplication and exponentiation. We avoid this by using arithmetic modulo a set of coprime numbers.

Let $b_1, b_2, \ldots, b_k$ denote a *base*, a set of coprime numbers, and $B = \prod_{i=1}^{k} b_i$ their product. The Chinese Remainder Theorem (CRT) states that any number x, $0 \leq x < B$ can be uniquely expressed as a vector x_i that represents a set of k congruences $x_i = x \mod b_i$ when $i = 1, 2, \ldots, k$. Furthermore, ring arithmetic modulo B can be performed in this domain. To compute the sum, difference or a product of two numbers $\mathrm{mod}\ B$, all one needs to do is to is to add or multiply the individual vector components i, each $\mathrm{mod}\ b_i$. Multiplication modulo B therefore has essentially linear complexity. Looking at Equation 2, one notices that when $z < B$, the entire public key computation can be performed in the CRT domain. This observation was first made in [4, 13].

Encryption Latency. One of the main advantages of a CRT implementation of PASSERINE is that serial transmission of encrypted data may be started immediately after the first word of $x^2 + r \cdot n$ has been computed. This is not the case with RSA or in ECC cryptography. This technique also helps to reduce the memory requirements of a PASSERINE implementation.

2.3 Carrying Payload Data in the Randomization Mask

An important and novel feature of PASSERINE is that r is used to carry payload data that has been encrypted using a random symmetric key, contained in x. This encoding technique allows us to essentially double the transmission bandwidth of the channel when compared to the original proposal by Shamir in [20].

3 Implementing PASSERINE on a Low-Resource Platform

We targeted our implementation for low-end 8/16 - bit microprocessors and microcontrollers. We chose to use a 1025-bit public modulus, which offers a reasonable level of security. For highly sensitive data, a larger modulus should be used. For symmetric encryption, we use AES-128 in counter mode [14, 15]. These parameters are practical choices that were selected for demonstration purposes – alternative choices may be more appropriate for some applications.

The CRT base (Section 2.2) was chosen to consist of 133 primes (6410th to 6542nd prime). These primes fit into sixteen-bit words as their numerical value ranges from 63929 to 65521. We chose to use primes rather than coprime numbers as there seems to be only a negligible encoding penalty from doing so. The encoding capacity is $\prod_{i=1}^{133} b_i \approx 2^{2125.70}$, which is only 2.30 bits short of the maximum channel capacity of $133 \times 16 = 2128$ bits.

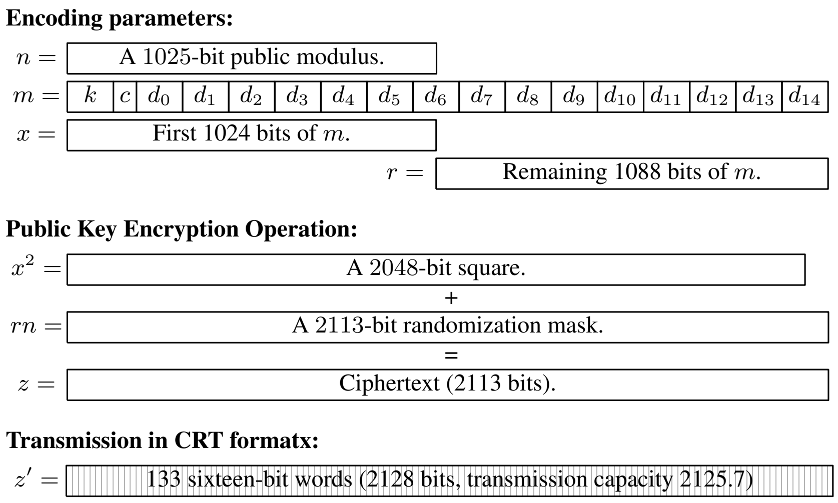
Encoding parameters:
$n =$
A 1025-bit public modulus.
$m =$
k c d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} d_{12} d_{13} d_{14}
$x =$
First 1024 bits of m.
$r =$
Remaining 1088 bits of m.
Public Key Encryption Operation:
$x^2 =$
A 2048-bit square.
+
$rn =$
A 2113-bit randomization mask.
=
$z =$
Ciphertext (2113 bits).
Transmission in CRT formatx:
$z' =$
133 sixteen-bit words (2128 bits, transmission capacity 2125.7)

Fig. 2. Encoding of key and payload in PASSERINE. Encryption is actually performed using CRT representation (modulo small primes in base b), not in standard two's complement representation.

Figure 2 illustrates PASSERINE data encoding. We use AES-128 [14] in counter mode (CTR) [15] for encryption of data blocks d_i

Table 1 describes the structure of the message m in our implementation. The first 1024 bits ($m[0..63]$) of the message are used as x and the latter 1088 bits ($m[64..131]$) as r in Equation 2. The 64-bit authenticator c should ideally be a secure MAC that is computed from all of the payload and key data. Our simple implementation sacrifices message integrity protection for implementation simplicity, and uses the result of a XOR operation between the upper and lower 64-bit halves of k as c. Adding good integrity protection with the EAX [2], CCM [16] or other authenticated mode is straightforward, but will slightly complicate implementation.

It is clear that the secret key k can be used to encrypt more payload data than the 240-byte section contained in z. Once k has been securely transmitted, the rest of the payload data can be sent directly using CTR encryption.

Table 1. Contents of the message m after symmetric encryption.

Symbol	Words	Contents
k	$m[0..7]$	A random 128-bit AES key, unique for each message.
c	$m[8..11]$	A 64-bit authenticator for message integrity and square root selection.
d_0	$m[12..19]$	First AES-encrypted block (payload bytes 0..15).
d_1	$m[20..27]$	Second AES-encrypted data block (payload bytes 16..31).
		...
d_{14}	$m[124..131]$	Fifteenth AES-encrypted data block (payload bytes 224..239).

3.1 An Experimental Small-Footprint Software Implementation

Algorithms are described using ANSI-C rather than pseudocode in this paper. Our small software AES implementation is described (and included) in Appendix A as Listing A.1. Note that only AES ECB encryption functionality is needed to implement counter mode as AES-CTR essentially behaves like a stream cipher (encryption and decryption operations are the same).

Listing 3.1. Modular reduction of a large integer stored as a vector of 16-bit words.

```
// reduce an l-word integer stored in d[] mod b
u16 red16(const u16 d[], u32 l, u16 b)
{
    u32 i, x;

    x = d[0];
    for (i = 1; i < l; i++) {
        x = ((x << 16) ^ ((u32) d[i])) % b;
    }
    return (u16) x;
}
```

Listing 3.2. Prime Base as difference between consecutive primes in decreasing order.

```
// primes 6542,6541..6410 as negative deltas from 0x10000
u8 pbdelta[133] = {
    15, 2,  22, 18, 30, 2,  10, 14, 4,  6,  6,  14, ...
                    (values removed)
    ... 10, 14, 4,  26, 4,  14, 6,  6,  10, 20, 28, 20
};
```

The main workhorse of our public key encryption implementation is the `red16()` function (Listing 3.1), which reduces a large integer to a single word $\bmod b$.

The input array `d[]` is interpreted as a big-endian integer in base 2^w where $w = 16$ is the word size. We note that if b is close to 2^w, it may be advantageous to break down the modular reduction $\bmod b$ to shifts, adds and a multiplication with a small constant $2^w - b$. Hardware implementations can reduce the gate count by choosing the set of coprime numbers in a special way.

The base of consecutive primes is stored in decreasing order. The base is compressed by only storing the difference of consecutive primes. Listing 3.2 gives the first and last values of the `pbdelta[]` table.

Padding and encryption is performed by the `srxenc()` function (Listing A.2). The function `srxenc()` returns the 133-word ciphertext in `ct[]`. Note that this implementation directly casts a byte array as an array of 16-bit words, and hence its behavior differs on little- and big-endian platforms.

The computational complexity of the public key encryption is $O(n^2)$. The individual values of `ct[i]` can be computed in any order or in a fully parallel fashion as the prime moduli are independent of each other. This is not possible with a traditional public key algorithms. It is not necessary to store the entire `ct[]` table in memory; we may start serially transmitting the ciphertext immediately when `ct[0]` is computed – PASSERINE therefore has exceptionally low latency. The total RAM requirement of the implementation can be decreased to around 300 bytes this way.

We also note that it is possible to precompute many of the variables, most importantly a vector containing the public modulus n as CRT residues. It is also possible to precompute the variable containing a random pad with the key check authenticator. Symmetric encryption can then be performed with a simple XOR operation, further reducing the latency in encryption.

3.2 Implementation profile of PASSERINE Encryption

Even compiled and retargetable implementations of PASSERINE are very compact. A breakdown of the code size of our particular implementation is given in Table 3.2. The entire encryption code (mostly contained in this paper) was complied with gcc 4.4.1 for an i386 target with optimization flags set to `-Os`.

Full sample code for both encryption and decryption is freely available from Author's web page [19].

Table 2. Self-sufficient PASSERINE implementation size profile (ROM).

Bytes	**Function**
256	AES S-Box table `sbox[256]`.
338	AES encryption routine `aesenc()`.
53	Modular reduction routine `red16()`.
133	Prime table `pbdelta[133]`.
361	Padding, CTR mode and public key encryption routine `srxenc()`.
1141	**Total implementation size.**

It is possible to significantly decrease the implementation size by hand-programming the implementation in assembler or by tweaking the algorithms (e.g. replace AES with a simple stream cipher).

3.3 PASSERINE Private Key Operation and Decryption

Unlike the lightweight encoding routine, our implementation of the private key operation and decryption uses the OpenSSL library `libcrypto` for fast big integer arithmetic and AES implementation. Decryption is more complicated than encryption (230 code lines) and we will only give the relevant math in this paper.

For ease of exposition we will denote by x_r^{-1} the unique modular inverse that satisfies $0 < x_r^{-1} < r$ and $x \cdot x_r^{-1} \equiv 1 \pmod{r}$ for a given x with $\gcd(x, r) = 1$.

Constructing z from the CRT representation z_i. A straightforward method for converting the ciphertext to conventional two's complement binary representation is given in Equation 3. Here b_i is the base with $k = 133$ coprime numbers, $B = \prod_{i=1}^{k}$, and the CRT ciphertext vector z_i satisfies $0 \leq z_i < b_i$ for each i.

$$z = \left(\sum_{i=1}^{k} z_i \cdot \frac{B}{b_i} \cdot \left(\frac{B}{b_i}\right)_{b_i}^{-1} \right) \bmod B. \tag{3}$$

The de-CRT coefficients $d_i = (B/b_i) \cdot (B/b_i)_{b_i}^{-1}$ in Equation 3 can be precomputed as they do not depend on the private parameters used.

Computing the square root. For decryption, one needs the private factorization pq of n. Rabin decryption is significantly easier to implement when $p \equiv q \equiv 3 \bmod 4$ and we will assume that this is the case. There are four square roots for every quadratic residue $\bmod pq$.

$$x_p = (z^{\frac{p+1}{4}} \bmod p) \cdot q \cdot q_p^{-1}. \tag{4}$$

$$x_q = (z^{\frac{q+1}{4}} \bmod q) \cdot p \cdot p_q^{-1}. \tag{5}$$

The four square roots of z are given by $x = \{x_p + x_q, x_p - x_q, -x_p + x_q, -x_p - x_q\} \pmod{n}$. The correct root can be recognized using the 64-bit authenticator c which is contained in x.

Symmetric decryption. Once the correct square root x is found, the mask r can be derived from

$$r = \frac{z - x^2}{n}. \tag{6}$$

We can then concatenate the two values and obtain the full message $m = x \mid\mid r$, which contains the symmetric decryption key and proceed to decrypt the entire data payload.

4 Security

PASSERINE is based on Shamir's randomized variant of the Rabin cryptosystem [18, 20], which is provably as secure as factoring n. The 1025-bit modulus used by our implementation may become susceptible to a GNFS factorization attack within the next decade [9]. Hence we recommend the use of a larger modulus when a very high level of confidentiality is required.

Secrecy in PASSERINE is also based on the AES algorithm, which has received more than a decade of thorough cryptanalysis. If secret key k values are truly random and not reused, a break of the CTR mode implies that AES itself is broken.

Our prototype PASSERINE implementation does not offer integrity protection. Full message integrity protection can be achieved by switching from CTR to CCM, EAX or similar authenticated mode of operation [2, 16]. These are essentially double modes of operation, and hence there is a performance drawback. The code size would grow by approximately 100 bytes.

5 Conclusions and Further Work

We have described PASSERINE, a practical variant of the Rabin public key cryptosystem that utilizes Naccache-Shamir randomized multiplication, prime base arithmetic to avoid big-integer arithmetic and uses a randomized mask to transmit payload data.

PASSERINE is exceptionally well suited for applications with limited computational capacity, such as embedded systems, RFID tags, smart cards and sensor network nodes. On these platforms PASSERINE tends to offer smaller software and hardware footprint, lower transmission latency, and greater encryption speed when compared to RSA or Elliptic Curve cryptography.

Our self-contained experimental implementation of PASSERINE has very compact implementation size. We have also described an experimental, very compact eight-bit implementation of the AES-128 encryption algorithm, which is of independent research value. We are currently designing an open-source hardware/software implementation of PASSERINE (See Section A.1).

As a future research direction, we note that an interesting feature of an over-defined small prime base representation of large integers is that if some of the individual elements are corrupt or missing, the large integer can still be reconstructed using the Chinese Remainder Theorem. We are currently investigating the use of this mathematical feature in conjunction with error detection and correction codes.

References

1. D. Aggarwal and U. Maurer. "Breaking RSA Generically Is Equivalent to Factoring." Eurocrypt 2009, LNCS 5479, Springer, pp. 36–53, 2009.
2. M. Bellare, P. Rogaway, and D. Wagner. "The EAX Mode of Operation." FSE 2004, LNCS 3017, Springer, pp. 389–407, 2004.
3. D. Boneh and R. Venkatesan. "Breaking RSA may not be equivalent to factoring." Eurocrypt '98, LNCS 1233, Springer, pp. 59–71, 1998.
4. O. Brugia, A. di Porto, and P. Filiponi. "Un metodo per migliorare I'efficienza degli algoritmi di generazione delle chiavi crittografiche basati sull'impiego di grandi numeri primi." Note Recesioni e Notizie, Ministero Poste e Telecommunicazioni, Vol. 33, No. 1-2, pp. 15–22, 1984.
5. J. Coron and D. Naccache. "Cryptanalysis of a Zero-Knowledge Identification Protocol of Eurocrypt '95." CT-RSA 2004, LNCS 2964, Springer, pp. 156–162, 2004.
6. M. Dowty. "Using an AVR as an RFID tag." Blog posting, 21 September 2008. `http://micah.navi.cx/2008/09/using-an-avr-as-an-rfid-tag/`
7. G. Gaubatz, J. Kaps, E. Özturk, and B. Sunar. "State of the Art in Ultra-Low Power Public Key Cryptography for Wireless Sensor Networks." PerCom 2005 Workshops, IEEE, pp. 146–150, 2005.
8. G. Gaubatz, J. Kaps, and B. Sunar. "Public key Cryptography in Sensor Networks – Revisited" Security in Ad-hoc and Sensor Networks – ESAS 2004, LNCS 3313, Springer, pp. 2–18, 2005.
9. T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D.A. Osvik, H. te Riele, A. Timofeev and P. Zimmermann. "Factorization of a 768-bit RSA modulus." `http://eprint.iacr.org/2010/006`. IACR Cryptology ePrint Archive: Report 2010/006, 2010.
10. A. Menezes, P. van Oorschot, and S. Vanstone. "Handbook of Applied Cryptography." CRC Press, 1996.
11. G. Lowe. "An Attack on the Needham-Schroeder Public-Key Authenticaion protocol." Information Processing Letters 56 (1995), Elsevier, pp. 131–131, 1995.
12. D. Naccache. "Method, Sender Apparatus And Receiver Apparatus For Modulo Operation." US patent: US5479511, 1995-12-26. European patent application: EP0611506, 1994-08-24. World publication: WO9309620, 1993.
13. D. Naccache, D. M'raihi, W. Wolfowicz, and A. di Porto. "Are Crypto-Accelerators Really Inevitable?" Eurocrypt '95, LNCS 921, Springer, pp. 404–409, 1995.
14. NIST. "Specification for the Advanced Encryption Standard (AES)" Federal Information Processing Standards Publication. FIPS-197, NIST, 2001.
15. NIST. "Recommendation for Block Cipher Modes of Operation." NIST Special Publication 800 - 38 A, NIST, 2001.
16. NIST. "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality." NIST Special Publication 800-38 C, NIST, 2004.
17. Y. Oren and M. Feldhofer. "A Low-Resource Public-Key Identification Scheme for RFID Tags and Sensor Nodes." WiSec '09, ACM, pp. 59–68, 2009.
18. M. C. Rabin. "Digitalized Signatures and Public-Key Functions as Intractable as Factorization." MIT / LCS / TR-212, Massachusetts Institute of Technology, 1979.
19. M.-J. Saarinen. PASSERINE Demonstration Package. Available from `http://www.netlab.tkk.fi/~mjos/passerine/`
20. A. Shamir. "Memory Efficient Variants of Public-Key Schemes for Smart Card Applications." Eurocrypt '94, LNCS 950, Springer, pp. 445–449, 1995.

A Implementation Tricks and Details

Listing A.1 gives our 8-bit ANSI-C implementation of AES-128 [14] in full. The vector `sbox[]` should contain the 256-byte AES S-Box. Due to its simple algebraic structure, the code required to generate the S-Box can be implemented in much less than 256 bytes, so if you have very little ROM but plenty of RAM, you may want generate the S-Box on the fly. Our implementation utilizes various symmetries and the relatively simple structure of the AES MDS matrix to reduce code size. These optimizations are not as readily applicable for AES decryption. This is especially true for the key schedule, which we compute "on the fly" to reduce the RAM requirement.

A.1 Microcontrollers as RFID Tags

Micah Dowty has demonstrated that an RFID tag can be constructed from an 8-bit Atmel AVR microcontroller (the ATtiny85) with very few external components [6]. Figure 3 shows the scale of these devices. While this "hack" only implements passive EM 4102 - compatible authentication, it clearly demonstrates that experimental RFID protocols can be built using cheap off-the-shelf components. We are currently designing an open-source hardware and software implementation of PASSERINE based on an AVR microcontroller.

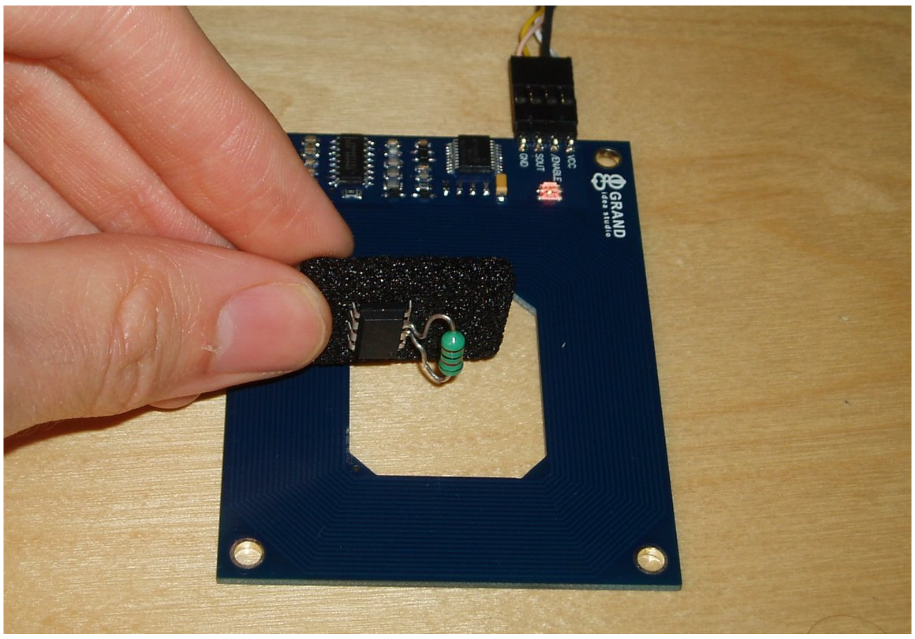

Fig. 3. A very simple RFID tag designed from an ATtiny85 microcontroller communicating with a Parallax RFID reader. © 2008 Micah Dowty. Published with permission.

Listing A.1. A size- and memory-optimized eight-bit AES-128 implementation. Round keys are computed on the fly.

```
// multiply by 0x02 in GF(256)
#define AESLS(x) (x & 0x80 ? (x << 1) ^ 0x1B : x << 1)

// AES-128 encryption of block b with a 128-bit key
void aesenc(u8 *b, const u8 key[16])
{
    u8 i, t[16], k[16], u, c;

    for (i = 0; i < 16; i++)    // copy the key
        k[i] = key[i];

    for (c = 0x01; c != 0x6C; c = AESLS(c)) {

        // AddRoundKey, SubBytes, ShiftRows
        for (i = 0; i < 16; i++) {
            t[(i - (i << 2)) & 15] = sbox[k[i] ^ b[i]];
            b[i] = 0;
        }

        // MixColumns (not on the last round)
        if (c != 0x36) {
            for (i = 0; i < 16; i ++) {
                u = t[i];
                b[i ^ 1] ^= u;
                b[i ^ 2] ^= u;
                b[i ^ 3] ^= u;
                u = AESLS(u);
                b[i] ^= u;
                b[(i & 12) + ((i + 3) & 3)] ^= u;
            }
        }
        // keying
        k[0] ^= sbox[k[13]] ^ c;
        k[1] ^= sbox[k[14]];
        k[2] ^= sbox[k[15]];
        k[3] ^= sbox[k[12]];
        for (i = 4; i < 16; i++)
            k[i] ^= k[i - 4];
    }
    // final AddRoundKey
    for (i = 0; i < 16; i++)
        b[i] = t[i] ^ k[i];
}
```

Listing A.2. PASSERINE Encryption Function. Our implementation does not require “big integer” arithmetic.

```
// encrypt and send a payload of up to 240 bytes

void srxenc(u16 ct[133],      // resulting ciphertext words
    const u8 pt[], int l,     // plaintext data & length
    const u8 key[16],         // AES key input
    const u16 mod[65])        // 1025-bit public modulus
{
    u32 i, j, b, x, y;
    u8 pad[264];

    // store the key k and authenticator c
    for (i = 0; i < 16; i++)
        pad[i] = key[i];
    for (i = 0; i < 8; i++)
        pad[i + 16] = key[i] ^ key[i + 8];

    // AES-CTR
    for (i = 24; i < 264; i += 16) {
        for (j = 0; j < 15; j++)
            pad[i + j] = 0;
        pad[i + 15] = (i >> 4) - 1;  // 0, 1, 2..
        aesenc(&pad[i], key);
    }

    // XOR the payload over the pad
    for (i = 0; i < l; i++)
        pad[i + 24] ^= pt[i];

    // public key encryption
    b = 0x10000;
    for (i = 0; i < 133; i++) {
         b -= pbdelta[i];
         x = (u32) red16((u16 *) pad, 64, b);    // x
         y = (((u32)
            red16((u16 *) &pad[128], 68, b)) *   // r
            red16(mod, 65, b)) % b;              // n
         ct[i] = (x * x + y) % b;                // x^2 + rn
    }
}
```