

Chapter 8

Security

8.1 Symmetric and asymmetric encryption for sensor networks

In early stages of research and development in many fields of computer science security was not an issue from the beginning. This is as well true for the development of operating systems, the specification of e-mail transmission protocols or wireless networks. But once structures have been established it is often difficult to integrate security afterwards.

Inventing secure cryptographic functions and in particular to prove their stability, called cryptanalysis, is not a trivial task. Generally, it is more advisable to read dedicated literature on the topic rather than a single chapter. By adding a chapter anyway, we want to create a certain sensitivity for the problem of security and we encourage the reader to go into the topic in more detail if time permits. Another motivation for writing this chapter was to bridge the gap between research in cryptography and sensor networks, since security issues do not easily migrate from cryptography into sensor networks or vice versa.

So some basic concepts of cryptography will be introduced with particular emphasize on their suitability for sensor networks in the following sections.

A principle important to mention in this context which most cryptographers agree on is that security has to be enforced by the robustness of the algorithms and not by trying to keep it secret.

We differentiate between symmetric and asymmetric encryption approaches. Symmetric encryption uses the same key both for encrypting and decrypting data. Usually, these algorithms are relatively easy to implement, need only limited computation power for encryption while (at least some of them) are known to be hard to break even with massive resources. Their disadvantages it that all participating parties have to agree on a common key prior to exchanging data. In a sensor

network each node might be fed with a secret key in the initialization process. However, a node with a secret key will always be in danger of being stolen by an attacker who might try to rip to the key out of the node's memory. Though some memory technologies can not easily be forced to reveal their content the designer of a sensor network will not want to rely on the missing competence of an attacker.

A solution to the problems of symmetric encryption schemes is *asymmetric encryption* which utilized are pair of keys. The public part of the key is made available to everyone. The private part will always remain in the domain of the owner of the key-pair. If the owner is a human rather than a computer, the private part is usually encrypted once more using a symmetric approach mentioned above. Even if an attacker can get hold of the private part, e.g., by stealing a storage medium or by breaking into a system, she can not use it without the symmetric key. In this particular case, symmetric encryption is a perfect choice since the key owner is the only decrypting and encrypting party.

The public part of the key can be used by everyone to encrypt a message for the recipient. The interesting property of asymmetric (aka. public-key-) encryption is that even the writer of the message can not read his own text anymore after the encryption (apart from the fact that he can of course store a plain text copy). The encryption with the public key can be imagined as a lock which can be closed but not opened anymore. Only the owner of the private part of the key can reveal the clear text again. The major advantage is that parties which have never met can exchange secret messages without the need of a secure channel. Everything can be done publicly.

In cryptography, communication usually takes place between two people called *Bob* and *Alice* (possibly to avoid using A and B all the time). The only problem which remains is a potential passive eavesdropper called *Eve*. Eve might generate her own pair of private and public key and publish it under the false name of Alice. When Bob searches a public-key database he finds a match for Alice but he can never be sure if he is tricked by Eve, at least not if they have never met in person. This problem is less severe in sensor networks as every node is in the same domain and can be equipped with a public key in the initialization phase, e.g., by hard-coding it into the program. But the problem does exist in the real world if people from different places want to communicate. One solution could be a public authority which guarantees that everyone contained in a database has been identified properly at least once in the past. In addition key-owners can check the identity of local fellows themselves and state publicly that they have identified someone, e.g., Alice. If a large number of people have done such an identification, Bob can be relatively sure that the public key belongs to Alice. The cluster of people who guarantee one another's identity is called a *web of trust*. In the best case Bob finds a relative or a close friend who claims to have checked Alice's identity before. More realistic he may find one or more friends who's friends claim to have seen

Alice in person, together with her public key. It is not uncommon that no more than four or less links are necessary to link people from different continents and communities.

The problem remains of how to make the “guarantee” convincing which someone gives for someone else. If it was put on a public database it could be faked easily. The solution to this problem is the reverse functionality of public-key encryption. This means that someone can encrypt a message with his private key as well. In this case the ciphertext which is available to everyone can be decrypted with the public key only. At a first glance this does not seem to make sense because publishing the cleartext would have been easier rather than putting the burden on everyone to do the decryption. But the useful property in this context is that only the owner of the private key is able to generate such a ciphertext. A plain cleartext could however be published by everyone. The encryption with a private key can be considered equivalent to putting a signature under a document which no one else can generate. In the web of trust, the digital signature serves as a guarantee that two persons have authenticated one another. If someone claims to have seen Alice he could write a letter of intent stating that Alice’s identity (along with her public key) is authentic and encrypt it with his private key. No one else could do that as well.

Rather than encrypting a whole letter, a writer will generate a checksum (or hash value) from the letter (or a file in general) and only attach the encrypted checksum to the plaintext. Then, the receiver only has to decrypt the hash value and generate one himself from the letter. If they match, the text has not been manipulated. The advantage of this proceeding is that a letter can also be read without doing any decryption. This is useful for database queries if it is sufficient to perform the check on the hit only and not on every entry.

The web of trust mentioned above may not be of big importance for a single sensor network in which all nodes origin from the same domain. It might possibly be interesting for cooperating networks having different owners. However, signing data with a private key can be important to prevent an intruder from inserting malicious information into a network. Packets with no signature can be identified and rejected. In cryptography an active intruder is often called *Malory*.

8.2 Privacy homomorphisms

Let us consider a tree hierarchy in a network like it is defined in the TAG approach in Chapter 5. We also assume that the application requires all data to be encrypted. This is not a severe problem for the leaf nodes of the tree since they are only concerned with their own data. But on the way to the root data can accumulate substantially. In order not to poison the nodes near the sink with a massive

data stream, different kinds of aggregations can take place in intermediate nodes as shown in Chapter ???. Though a node may be able to encrypt its own data, parent nodes have to decrypt all messages from their children. This increases the computational demand at the inner nodes a lot though they are the most important ones and should save on energy as much as possible. A lost inner node can disconnect a whole branch of a tree at once.

Thus it would be desirable to find a way that allows to process data without having to decrypt it while preserving the content. The aggregating node does not necessarily need to interpret the data, it only has to be able to work with it.

A concept which meets the above requirements is called a *privacy homomorphism* and has been introduced by Rivest, Adelman and Dertouzos as early as 1978 in their work *On data banks and privacy homomorphisms* [?].

A privacy homomorphism is an encryption function which allows operations like e.g., additions or multiplications on the encrypted data. The result will yield an encrypted codeword which is similar to the codeword that would be obtained by applying the operation on the cleartext first and encrypting the result afterwards. In more mathematical terms we can say that the encryption function distributes over the operation. Additions or multiplications are of particular interest in this context.

An instance of such a privacy homomorphism (PH from now on) was suggested by Domingo-Ferrer in *A Provably Secure Additive and Multiplicative Privacy Homomorphism* [?]. It is a symmetric encryption scheme which uses the same key for encryption and decryption. Girao, Westhoff and Schneider suggested to apply it for sensor networks in *CDA: Concealed Data Aggregation for Reverse Multicast Traffic in Wireless Sensor Networks* [9].

The Domingo-Ferrer Privacy Homomorphism

The basic principle of the Domingo-Ferrer encryption is to decompose a number (the cleartext) into a sum consisting for an arbitrary number of d summands. Then, the vector of summands is transmitted over a public channel and the receiver has to add the components in order to obtain the cleartext again. So far, an eavesdropper could do the same and would also obtain the cleartext so the summands have to undergo some kind of scrambling. However, the scrambling or encryption has to be chosen carefully in order not to destroy the distributivity. In the end we want to be able to add or multiply two of those vectors to obtain an encrypted sum.

First, we will go into details on the generation of the key. According to Domingo-Ferrer a large integer g has to be found which holds the following two properties:

- It should consist of a large number of divisors. This can easily be ensured

by repeatedly multiplying prime numbers. The product does not have to consist of unique numbers only. The same prime can also be multiplied several times. In effect, g simply is a product of integers.

- The resulting number should be chosen such that many integers $r < g$ can be found which have a so-called modulo inverse. The inverse element, e.g., for multiplications is more popular. It is true for (mathematical) fields in general that every number n has a multiplicative inverse n_{inv} with $n \times n_{inv} = 1$. The same circumstance exists for the modulo operator. The large g should be chosen such that many integers r can be found for which an r_{inv} exists so that $r \times r_{inv} \bmod g = 1$.

E.g., 2 is modulo invertible with regard to the divisor 9, since $2 \times 5 = 10 \bmod 9 = 1$

In the context of the modulo operator and a chosen g it is not self-evident how many modulo invertible integers r exist. If g was a prime number itself, a modulo inverse can be found for every integer. However, g is the result of a large product.

For the g chosen above we need a g' which divides g with no remainder. Domingo-Ferrer states that $g \times \log_{g'}$ was a hint to the security level. Furthermore, a particular $r < g$ has to be chosen with $r \times r_{inv} \bmod g = 1$. Finally, an integer $d \geq 2$ is needed. As mentioned above a cleartext a will be decomposed into d summands a_k .

$$a = \left(\sum_{k=1}^d a_k \right) \bmod g' \quad (8.1)$$

The sum does not need to yield a exactly as shown in Expression 8.1. It is also allowed to result in $a + n \times g'$ if n is a positive integer. In other words: Adding g' repeatedly is allowed. Note that this means as a consequence that no $a \geq g'$ can be encoded. We will now come to the actual vector $E(a)$ which is to be transmitted:

$$\begin{aligned} E(a) &= \left(a_1 r^1 \bmod g, a_2 r^2 \bmod g, \dots, a_d r^d \bmod g \right) \\ &= (e_1(a), e_2(a), \dots, e_d(a)) \end{aligned} \quad (8.2)$$

The components of the cyphertext $E(a)$ are denoted with $e_k(a)$. As shown in Equation 8.2, encryption is simply done by multiplying with r in the first component, r^2 in the second and so on.

They are decrypted by multiplying r_{inv} in the first component, r_{inv}^2 in the second (and so on) and applying modulo g unless the result is smaller than g .

$$\begin{aligned} D(E(a)) &= \left(e_1(a)r_{inv}^1 \bmod g, e_2(a)r_{inv}^2 \bmod g, \dots, e_d(a)r_{inv}^d \bmod g \right) \quad (8.3) \\ &= (a_1, a_2, \dots, a_d) \end{aligned}$$

Of course, in order to obtain a again the components a_k have to be added:

$$a = \left(\sum_{k=1}^d a_k \right) \bmod g' \quad (8.4)$$

So in essence, scrambling the components is done by multiplying with r^k and multiplying with r_{inv}^k inverts the proceeding if $\bmod g$ is applied after each operation. Let us quickly try an example and analyze why this works in order to de-mystify the approach a little bit. We want to scramble the value $a = 7$, with $r = 2$ and its modulo inverse 5 if $g = 9$.

$$\begin{aligned} 7 \times 2 &= 14 \bmod 9 = 5 \text{ (encrypted value)} \quad (8.5) \\ 5 \times 5 &= 25 \bmod 9 = 7 \text{ (decrypted value)} \end{aligned}$$

In short we could summarize the “scrambling trick” as follows: An integer r and its corresponding r_{inv} is chosen such that

$$r \times r_{inv} \bmod g = 1 \quad (8.6)$$

If we multiply a cleartext word a with 1 we yield

$$a \times 1 = a \quad (8.7)$$

We can also replace 1 by $(r \times r_{inv} \bmod g)$ and get:

$$a \times (r \times r_{inv} \bmod g) = a \quad (8.8)$$

If the modulo is applied to a product (remember that it is the remainder of a division) it is also possible to apply it to every intermediate product. So we can write:

$$[((a \times r) \bmod g] \times r_{inv} \bmod g = a \quad (8.9)$$

Actually this is already the whole magic which takes place in the Domingo-Ferrer encryption. The first operation from Expression 8.9 within the squared brackets takes place at the sender side and the second multiplication and modulo takes place at the receiver side.

8.2.1 Additions in the encrypted domain

We will now give an example for an entire encryption cycle in which two senders encrypt a value and the receiver adds them without prior decryption. The following values have been chosen:

$$\begin{aligned}
 g &= 416, g' = 208 & (8.10) \\
 d &= 4 \\
 r &= 7, r_{inv} = 119 \\
 a &= 12, b = 14 \text{ (data as cleartext)}
 \end{aligned}$$

Sender A and B choose to decompose the sum as follows:

$$\begin{aligned}
 12 &= 2 + 3 + 5 + 2 \\
 14 &= 1 + 3 + 6 + 4
 \end{aligned}$$

Both stations perform the following encoding:

$$\begin{aligned}
 E(12) &= ((2 \times 7^1) \bmod 416, (3 \times 7^2) \bmod 416, (5 \times 7^3) \bmod 416, (2 \times 7^4) \bmod 416) \\
 &= (14, 147, 51, 226) \\
 E(14) &= ((1 \times 7^1) \bmod 416, (3 \times 7^2) \bmod 416, (6 \times 7^3) \bmod 416, (4 \times 7^4) \bmod 416) \\
 &= (7, 147, 394, 36)
 \end{aligned}$$

The parent of node A and B will receive both vectors and add them. In general this is everything the sink has to do because the sum is encrypted already and needs no further processing. However, we are curious to see whether the privacy homomorphism actually works and decode the result immediately.

$$\begin{aligned}
 E(12) + E(14) &= (14, 147, 51, 226) + (7, 147, 394, 36) \\
 &= (21, 294, 445 \bmod 416, 262) = (21, 294, 29, 262) \\
 D(E(12) + E(14)) &= 21 \times 119^1 \bmod 416 + 294 \times 119^2 \bmod 416 + \\
 &\quad 29 \times 119^3 \bmod 416 + 262 \times 119^4 \bmod 416 \\
 &= 3 + 6 + 11 + 6 = 26
 \end{aligned}$$

Distributivity for the addition

The example above showed that the encryption distributes over the addition or in more general terms: It makes no difference whether the addition is performed before or after the encryption. This is true for the following reason:

$$\begin{aligned}
 E(a+b) &= \\
 (a+b) \times r \bmod g &= \\
 (a \times r + b \times r) \bmod g &= \\
 a \times r \bmod g + b \times r \bmod g &= \\
 E(a) + E(b) &
 \end{aligned}$$

8.2.2 Multiplications in the encrypted domain

As we have seen above, the addition is easily done by adding the components of two vector-valued cipher words. This corresponds to the addition of digits in the decimal system with the only difference that no carry over takes place. With this analogy in mind we will try whether the multiplication works similarly.

In Figure 8.1 the components $E(12)$ and $E(14)$ are ordered in reverse order (from the largest to the smallest component) and the cross product is computed like it can be done for decimal values. The small numbers in the figure denote the component number of either the arguments and the result. In the end, the sum of each component is calculated modulo 416 and the bottommost line includes the decrypted summands a_k .

4	3	2	1	x	4	3	2	1	
226	51	147	14		36	394	147	7	
8	7	6	5	4	3	2	1		
226x36	51x36	147x36	14x36						
	226x394	51x394	147x394	14x394					
		226x147	51x147	147x147	14x147				
			226x7	51x7	147x7	14x7			
232	192	368	109	26	175	98			sum mod 416
8	32	48	43	26	9	2			decryp. value

Figure 8.1: Multiplication in the context of the Domingo-Ferrer *privacy homomorphism* in the encrypted domain.

Actually the multiplication worked since

$$8 + 32 + 48 + 43 + 26 + 9 + 2 = 168 = 12 \times 14$$

Distributivity for the multiplication

The example above is a good hint that the encryption distributes over the multiplication but it remains to be shown for the general case. More formally we can ask whether $E(a \times b) = E(a) \times E(b)$?

For the encryption we have to decompose the numbers a and b into d summands:

$$\begin{aligned} a &= \sum_{i=1}^d a_i ; b = \sum_{j=1}^d b_j \\ a \times b &= (a_1 + a_2 + \dots + a_d) \times (b_1 + b_2 + \dots + b_d) \\ &= \sum_{i=1}^d a_i \sum_{j=1}^d b_j = \sum_{i=1}^d \sum_{j=1}^d a_i b_j \end{aligned}$$

Obviously there is no difference between encrypting the product $a \times b$ or the sum:

$$\begin{aligned} E(a \times b) &= E\left(\sum_{i=1}^d \sum_{j=1}^d a_i b_j\right) = \sum_{i=1}^d \sum_{j=1}^d a_i b_j r^{i+j} \bmod g \\ &= \sum_{i=1}^d \sum_{j=1}^d a_i r^i b_j r^j \bmod g = \sum_{i=1}^d \sum_{j=1}^d (a_i r^i \bmod g) (b_j r^j \bmod g) \\ &= \sum_{i=1}^d (a_i r^i \bmod g) \sum_{j=1}^d (b_j r^j \bmod g) \\ &= \sum_{i=1}^d (a_i r^i \bmod g) E(b) = E(a) \times E(b) \end{aligned}$$

Again, the distributive nature of the multiplication and the modulo operator allows to multiply prior or after the encryption.

8.2.3 Critical review

Several studies have been done on the security of the Domingo-Ferrer PH. Cheon and Nam propose a plaintext attack in [5]. If the intruder can guess several plaintexts the key can be broken in polynomial time. Knowing plaintexts means in this context that some ciphered words can be assigned to their decrypted values. The ease of guessing depends on the data being sent by an application. Note that in the context of sensor networks plain text may be easy to gather. If it is known that a node measures the local temperature doing the same measurement yields plain text already. It could be used to break the symmetric key of the entire network if only a single key is used.

The fact that the approach is not cryptographically secure does not necessarily mean that it is not useful. It is enough to increase the effort for the intruder to a degree which makes the attack uninteresting. This is in particular true if data is measured which is available publicly anyway.

A drawback of the approach is the fact the measurements are ballooned in the first place. The number of summands determines the factor by which the size of a message is increased (remember that a whole vector of d components was created from a single scalar value). Furthermore, an upper bound for the range of numbers to be encrypted is g' since the sum of the decomposed summands are taken mod g' . But the actual components of the encrypted values can be as large as $g - 1$ which is a waste of bits because g' divides g without a remainder. In return, the encryption does allow to aggregate data which is a significant improvement as compared to sending encrypted information which must not be aggregated.

Though the multiplication can be performed as shown in the example above, there is no multiplicative inverse which means that we can not divide. So when multiplying values in the inner nodes we have to take care not to exceed the limit g' . After reaching g' a so-called wrap around would happen which decreases a value by g' . The only solution to this problem is to decrypt a value, perform the necessary division and encrypt it again. The problem of the wrap around exists for additions though additions are less prone to exceed their limits.

An advantage of the non-deterministic nature of the DF encryption is the fact that the same number can map to many different cyphered words. Remember that the decomposition of a number into a sum was not restricted. Any summands will represent the same number as long as their sum matches the value to be encrypted. So an intruder can not even tell, whether equal values are transmitted.

8.3 Public key encryption

The kind of public key encryption mentioned in the introduction would be the ultimate solution for sensor networks with regard to security. With the ability to encrypt data based on the public key but the inability to decrypt the message, stealing a sensor node would be no more option for an attacker. Even the nodes themselves would not be able to decrypt their own messages or those of their neighbors. The private key would solely reside in a distant control center.

8.3.1 The RSA scheme

We will now introduce the basic principle of the well-known RSA encryption which was suggested by Rivest, Shamir and Adelman in 1977 [22]. Like in the Domingo-Ferrer scheme the modulo inverse plays an important role again but this time it is not multiplied but the cleartext is raised to the r -th power if r denotes the key.

The encryption-decryption scheme works as follows:

m = message
 c = cyphertext
 r = public key (3 in the example)
 r' = private key (7 in the example)
 n = common part of public and private key (33)

$$\begin{aligned}
 c &= m^r \bmod n \\
 c^{r'} \bmod n &= m
 \end{aligned}
 \tag{8.11}$$

Figure 8.2 shows an example in which a couple of letters are encrypted and decrypted. The letters are represented by their order in the alphabet.

(C)	$3^3=$	27	$\bmod 33 =$	$27^7 =$	10460353203	$\bmod 33 =$	3	(C)
(R)	$18^3=$	5832	$\bmod 33 =$	$24^7 =$	4586471424	$\bmod 33 =$	18	(R)
(Y)	$25^3=$	15625	$\bmod 33 =$	$16^7 =$	268435456	$\bmod 33 =$	25	(Y)
(P)	$16^3=$	4096	$\bmod 33 =$	$4^7 =$	16384	$\bmod 33 =$	16	(P)
(T)	$20^3=$	8000	$\bmod 33 =$	$14^7 =$	105413504	$\bmod 33 =$	20	(T)
(O)	$15^3=$	3375	$\bmod 33 =$	$9^7 =$	4782969	$\bmod 33 =$	15	(O)

Figure 8.2: Example for the public key encryption proposed by Rivest, Shamir and Adelman.

Let us now see how to derive the public- and the private key. First, we have to obtain two prime numbers p and q which form the product $n = p \times q$. The number n will be part of the public and of the private key. For the above example we used $p = 3$ and $q = 11$. The strength of RSA lies in the fact that the factorization of n is not trivial if the prime numbers are sufficiently large. As of today, 1024-bit or more are considered to be a safe choice for the length of the product n . The reason for the security is that the factorization of a number involves a more or less sophisticated trial and error approach. Despite a couple of centuries of consideration, no algorithm has been developed which can factor a prime in at least polynomial time.

The n will be one part of both, the public and the private key. The other part will be r and r' , respectively. We will now have a look at how to derive these values.

First, the integer $z = (p - 1)(q - 1)$ has to be calculated. Then we need another integer r which has no common factor with z (rather than the “trivial” factor 1). The pair (n, r) forms the key already. It can either be used as a public or as a private key. We choose it as the public part of the key pair from now on. The complementary path (the private key in our case) is formed by (n, r') . r' has to be chosen such that $r \times r' \bmod z = 1$. The initial prime numbers from which e and d were derived are not involved in the encryption or decryption process. They should be deleted right after the key generation because their mere existence is a security risk.

The way RSA was used in the example above is not better than any substitution scheme which replaces one letter in the alphabet by another. Even more severe, the cleartext 0 always maps to 0 and 1 stays 1, no matter how the key was chosen. This is the reason why in practice a whole block of information is concatenated and why certain padding schemes are applied in order to make 0 and 1 unlikely.

Last not least, the example shown above hides the amount of calculation which has to be performed in real applications. Actually, e and d are numbers of the degree of magnitude of 500 bits and more. The message could e.g., be 64 bytes long. This means that a huge number has to be raised to the power of another huge number. Since the size of the numbers are far beyond the size of a processor’s register the calculation has to be decomposed and emulated by software.

Note that the result of $(2^{64 \times 8}) \uparrow (2^{500})$ could not even be stored in a sensor node easily. Only the fact that the results of the en- and decryption calculations are taken modulo n makes the computation feasible. In every intermediate step of

the power calculation, *modulo* n can be performed. If the intermediate result was smaller than n anyway, nothing happens. If it was larger, a multiple of n would be subtracted. This is exactly the same that happens to the result in the end. The remainder can however not be influenced by intermediate modulo operations.

In asynchronous applications like sending e-mail, the computational overhead of public-key encryption is not important. In synchronous and time-critical applications, like e.g., browsing the web, a client and a server will use RSA to establish a *Secure Socket Layer (SSL)* session. Then, a random key is negotiated between both communicating parties. This session key is then used in conjunction with a symmetric encryption algorithm which is much faster. Using RSA to initialize a secure session is a good choice if both, the client and the server are in a secure domain and only the transmission channel is insecure.

An alternative to RSA could be *elliptic curve cryptography (ECC)* in future applications. It is common ground in the cryptographic community that ECC can reduce the length of keys significantly while maintaining the same level of security. A 1024-bit key of RSA is considered to be equivalent to a 160-bit ECC key [28].

A note on key generation

The security of RSA is based on the problem to factorize n into the two primes p and q from which it was constructed. Of course this implies that the primes can be generated by the encrypting party as well. Though it is easier to find an arbitrary prime q as compared to factorize n , it is still no trivial task, in particular given that the attacker may possess computational capabilities which could be several degrees of magnitude larger than the encrypting party.

Fortunately, the density of primes is sufficiently large, even for large numbers (though it decreases). The chance to find a prime q only decreases proportional to $1/\log(q)$ so the density of the set of primes is no reason to worry. But it is still an open issue to prove that q is a real prime. In practical applications, a number of tests are performed on a candidate. Though these do not eventually prove that no factorization is possible, they make it very unlikely.

Another critical issue to consider is how to generate the random number. An (almost) entirely deterministic machine like a computer is not able to produce much entropy or randomness. In fact, a swapped bit in the memory usually is a catastrophe for someone. So the sensor node has to gather some entropy from the external world. At a first glance the sensors it is equipped with may be a good source. However, they are not if they measure the same phenomena an attacker can measure.

8.3.2 Feasibility for sensor networks

Many authors argue that public key encryption is too demanding for application in sensor networks.

Wander et. al. have implemented RSA and ECC for the *Mica2dot* sensor nodes¹ which use the low-power 8-bit CPU ATmega128L²[28]. Contrary to widely held beliefs their thorough analysis of time- and energy-consumption surprisingly shows that “public-key cryptography is very viable on small wireless devices”. One reason is, that sending information using a slow transceiver takes a lot of time. The authors state that sending a single bit corresponds to performing 2090 CPU cycles with regard to energy consumption on the *Mica2dot* nodes.

Like it is true for SSL connections on the Internet, the authors have implemented a slightly simplified session initiation protocol for their nodes. After the initialization, a symmetric cypher is used. A second reason for the feasibility of their implementation is that they transmit a larger amount of data over a single session. Of course this reduces the ratio between the payload and the effort for the session initialization. When sending 64kB, the amount of energy consumption for the initialization of the session drops down to about 2% for ECC. ECC uses about five times less energy in their evaluation. One of its advantages over RSA is that no large prime numbers are needed for the key generation process.

The drawback of initializing sessions is that every node has to contain a unique public key and the private counterpart. An attacker could take over a whole node and use both keys for intrusion. Whether this is beneficial for the attacker depends on the role of the captured node. If it had only few children, the amount of data that could be gathered is small. If the node resided near the sink, the amount of its children could be substantial. Theoretically, the ultimate solution to this problem would be to encrypt all data with a public key only and leave the private key for decryption in a safe control center. On the other hand this would prevent any kind of aggregation.

Interestingly, the authors mention the MSP430 processor which we use in the context of this book in the end of their evaluation. They state that the ratio between encryption and transmission might even be better for this processor as it can handle 16-bit values at once. An additional advantage is that it runs at twice the clock rate of the Amtel processor whereas sending bits is as slow on the ESB platform as on the *Mica2dots*. So the proportion of the encryption should shrink on the ESB platform as compared to the *mica*-nodes.

¹manufactured by Crossbow Technologies, see www.xbow.com

²manufactured by Atmel