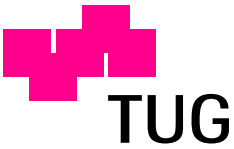
Implementation of a Standards-Based Security Architecture for Wireless Sensor Networks

Michael Wurm
mwurm@sime.com

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
A–8010 Graz, Austria



Diploma Thesis

Supervisor: Dipl.-Ing. Johann Großschädl
Assessor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Karl Christian Posch

January 2006

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original, except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Diplomarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

Michael Wurm

Acknowledgments

I would like to thank my supervisor Johann Großschädl for his help and support. I am especially indebted to him as he engineered my internship with Sun Microsystems, where I performed my practical work. I thank Stefan Tillich, my advisor Karl Christian Posch, and the other members of the Institute for Applied Information Processing and Communications (IAIK) of Graz University of Technology.

I also wish to thank the members of Sun Microsystems Laboratories in Menlo Park, California, especially Vipul Gupta, Sheueling Chang-Shantz, Nils Gura, and Hans Eberle, for their help and feedback during the practical part of this work.

Last, but not least, I thank my friends and my parents, who provided me with invaluable moral support, and at times, took care of the necessary distractions.

Abstract

As the number of potential applications for wireless sensor devices grows, so does the need to simplify and secure interaction with these devices. Embedding a secure web server (capable of HTTP over SSL, *aka* HTTPS) inside these devices enables them to be monitored and controlled securely via a user-friendly browser-based interface. This idea led to the development of a small secure web server for wireless sensors, nicknamed "Sizzle", by Sun Microsystems Laboratories.

This thesis presents several enhancements of Sizzle's architecture, made with the goal of improving its performance, energy efficiency, and reliability. These enhancements include the implementation of an energy-conserving communication protocol, capable of low-power listening for incoming service requests, and the porting of Sizzle to a new platform with a faster processor and faster networking. SSL key exchanges based on RSA and Elliptic Curve Cryptography are compared on different wireless sensor platforms, and an empirical analysis of the energy consumption of SSL on the Telos platform is presented.

Keywords: Wireless Sensor Networks, Elliptic Curve Cryptography, SSL, Secure Web Server

Kurzfassung

Mit der Zahl der möglichen Anwendungen für drahtlose Sensoren steigt auch das Bedürfnis, die Interaktion mit diesen Geräten sicherer und einfacher zu gestalten. Durch Einbetten eines sicheren Webservers (mit Unterstützung für HTTP über SSL, genannt HTTPS) in diese Geräte wird es möglich, einzelne Sensoren über eine benutzerfreundliche, Browser-basierte Oberfläche zu überwachen und zu steuern. Basierend auf dieser Idee wurde in den Sun Microsystems Labaratories ein kleiner, sicherer Webserver für drahtlose Sensoren namens "Sizzle" entwickelt.

Diese Diplomarbeit präsentiert einige Verbesserungen an der Architektur von Sizzle, mit dem Ziel die Performance, Energieeffizienz und Zuverlässigkeit zu steigern. Diese Verbesserungen beinhalten die Implementierung eines Kommunikationsprotokolls, das in der Lage ist, energiesparend auf eingehende Anforderungen zu warten, und das Portieren von Sizzle auf eine andere Plattform mit schnellerem Prozessor und schnellerem Netzwerk. Der SSL Schlüsselaustausch basierend auf RSA und Elliptische-Kurven-Kryptographie wird auf verschiedenen Plattformen verglichen, und eine empirische Analyse des Energieverbrauchs von SSL auf der Telos Plattform wird präsentiert.

Stichwörter: Drahtlose Sensornetzwerke, Elliptische-Kurven-Kryptographie, SSL, Sicherer Webserver

Contents

Introduction

In recent years, the Internet has grown far beyond servers and desktop PCs to include handheld computers and cell phones. In the near future an even smaller class of devices will become available in large numbers and will further accelerate the expansion of the Internet: tiny, battery-powered wireless sensor devices, capable of monitoring temperature, vibration, light intensity, moisture, vital signs, and more. Potential applications of these wireless sensors include habitat monitoring, medical emergency response, battlefield monitoring, process control, and home automation.

Interaction with wireless sensors is made difficult by the fact that they usually lack a display or input mechanism and users are often unskilled in administrating computers. Consider the task of configuring thresholds for alarm notifications in a heart monitor or a temperature sensor or remotely accessing their current reading. By integrating a web server into a wireless sensor device and by connecting it to the Internet, users are able to access it using a standard web browser. This approach not only has the advantages of a familiar user interface and platform independence, but also allows the integration of the widely trusted security protocol SSL (Secure Sockets Layer), which is supported by virtually every browser.

Sizzle[1] [19] is a tiny web server for embedded systems which supports both HTTP and HTTPS. The latter is widely used on the Internet to protect security-critical transactions in applications like e-commerce and online banking. An emerging alternative to the most common public-key cryptosystem, RSA, is Elliptic Curve Cryptography (ECC). ECC has higher performance and requires smaller keys than RSA while providing the same level of security. These advantages are especially important in resource-constrained environments. Sizzle supports both cryptosystems, and can take advantage of the performance benefits of ECC when communicating with an ECC-enabled version of the Mozilla browser. The support for RSA ensures interoperability with browsers like Internet Explorer and Safari, which do not yet support ECC.

Sizzle runs efficiently within the tight computing, network, and memory constraints of wireless sensor devices. Besides these constraints the limitation of available energy is also a major concern. Where electricity is not available, batteries provide a nonrecurring energy reservoir, and maximizing the battery lifetime is crucial to keeping maintenance costs low. The fact that security is a key requirement in many applications raises the question about the impact of security on energy consumption and performance.

This document describes the results of my efforts to improve and to analyze the performance, reliability, and energy efficiency of Sizzle. These efforts involved

- Porting Sizzle to a new platform with a faster wireless transceiver and a more powerful CPU, as well as reimplementation of most cryptographic functionality.

[1] The name derives from "Slim SSL" (SSSL).

- Architectural enhancements to reduce the energy consumption of Sizzle while waiting for incoming service requests.
- Analysis and comparison of the performance of Sizzle on various platforms.
- Analysis of the energy consumption of an SSL handshake and bulk data transfer, as well as the underlying cryptographic algorithms.
- Investigation of the overhead of security by comparing HTTP with HTTPS data transfers (using RSA and ECC public-key mechanisms), and estimation of the battery lifetime in different scenarios.

This document consists of seven chapters where the first four chapters give the necessary theoretical background in cryptography and introduce wireless sensor networks. The last three chapters detail the architecture and implementation of Sizzle, with focus on my work, and present an evaluation the resulting system.

Chapter 1 introduces basic concepts of cryptography and describes the cryptographic primitives required to implement a security protocol.

Chapter 2 elaborates on elliptic curve cryptography and discusses its advantages over other cryptosystems such as RSA. A brief mathematical background and a selection of the most important algorithms are given.

Chapter 3 describes the SSL protocol, being one of the most widely used security protocols in today's Internet. The focus lies on the handshake when using elliptic curve cryptography for key exchange and authentication. A short security analysis is given and existing implementations are compared.

Chapter 4 introduces wireless sensor networks, their applications, and security-related problems. Special challenges in software development for wireless sensors are discussed, such as tight resource constraints and the limitation of available energy. A popular family of hardware platforms and an operating system tailored for wireless sensors are described in detail.

Chapter 5 presents the architecture of Sizzle, which allows you to connect wireless sensors to the Internet and to access them with a standard web browser. Potential applications are discussed and a number of requirements are compiled. A simple communication protocol for wireless networks is presented which was designed with the goal of minimizing the energy consumption.

Chapter 6 focuses on the software stack of the secure web server on the wireless sensors, in particular the implementation of elliptic curve cryptography. The choice of algorithms is motivated and the structure of the implementation is detailed.

Chapter 7 evaluates the implementation of this security architecture in terms of resource utilization, performance, and energy efficiency. The performance of Sizzle on different wireless sensor platforms is compared and battery lifetime estimations are given for different scenarios.

Chapter 1

Cryptography

Cryptography (from Greek kryptós, "hidden", and gráphein, "to write") is the science of keeping information secret. That is how it started. Today cryptography covers not only secrecy, but also authentication, digital signatures and more. While the cryptographers' job is to secure information their counterplayers, the cryptanalysts, try to break cryptographic algorithms, so called *ciphers*, in order to recover the protected information. They exploit weaknesses in the ciphers or weaknesses in the use of these ciphers.

A strong cipher has the property that its security does not rely on the secrecy of the algorithm, but only on the secrecy of the key. The best ciphers are public ones, studied by many people over a long time, which are still found secure. On the other side, proprietary schemes often have security problems which are discovered as soon as details about the algorithms become public.

The security level of a cipher is commonly measured in bits. The number of bits corresponds to the length of the key when there is no faster way to break the cipher other than to try every possible key. Now, the minimum security level that is regarded secure is 80 bits, which means that 2^{80} computational steps are required to find the key. However the security level is not always equivalent to the key length, since there are ciphers which are easier to break than by exhaustive search.

Cryptography has to deal with the following elementary requirements [35]:

- **Confidentiality**. Only entitled parties can read the message.
- **Authentication**. The origin of a message can be identified.
- **Integrity**. The receiver is able to verify that the message has not been modified.
- **Nonrepudiation**. The sender is not able to falsely deny that he sent the message.

This section gives an overview over state-of-the-art methods in cryptography and presents cryptographic primitives which can be combined to create systems meeting the requirements mentioned above.

1.1 Symmetric Cryptography

Symmetric cryptography uses the same key for encryption and decryption. The sender and the receiver have to agree on a key before secure communication can take place. The problem on how to keep the key hidden from potential adversaries can be solved using public-key cryptography.

The basic operation of a symmetric cipher is depicted in Figure 1.1. Encryption can

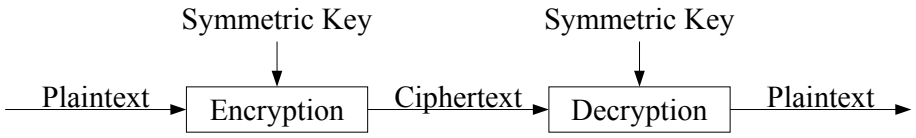


Figure 1.1: Model of a Symmetric Cipher [36]

be seen as a function which takes some plaintext and the key as argument and produces ciphertext. Decryption reverses this process and produces plaintext from a given ciphertext and the key.

Symmetric ciphers are divided into two classes: *Stream ciphers* and *block ciphers*. Stream ciphers operate on single data items at a time, while block ciphers operate on multiple data items at a time.

1.1.1 Stream Ciphers

Stream ciphers contain a keystream generator which is initialized using the secret key. The keystream is generated in a way, such that it appears to be random to people who do not have knowledge about the key. The function used for encryption is the same function used for decryption. For encryption, it combines the plaintext with the keystream to produce the ciphertext and for encryption it combines the ciphertext with the keystream to produce the original plaintext.

Many stream ciphers operate on bits and rely on special feedback shift registers implemented in hardware. Their hardware implementations are usually faster and smaller than the implementations of block ciphers.

The RC4 cipher [33], however, is a very simple stream cipher which operates on bytes and is suitable for efficient implementation in software. RC4 is a proprietary algorithm from RSA Security, Inc. which has been reverse-engineered and published in a newsgroup. To avoid copyright and patent issues, RC4 is often referred to as *ARCFOUR*.

1.1.2 Block Ciphers

While stream ciphers maintain an internal state, block ciphers are stateless and involve two different functions for encryption and decryption.

$$\begin{aligned} ciphertext &= encrypt_{key}(plaintext) \\ plaintext &= decrypt_{key}(ciphertext) \end{aligned}$$

When a block cipher is used without precaution, it is not possible to detect insertion and deletion of blocks. Furthermore, the fact that identical blocks of plaintext always result in identical blocks of ciphertext can be exploited.

To overcome these problems various modes of operation have been defined, which introduce state to the process of encryption and decryption. The most common mode is the Cipher Block Chaining (CBC) mode, depicted in Figure 1.2. Here, the ciphertext of the previous block is combined with the plaintext of the current block before encryption. In addition to the key, the CBC mode requires an initial value IV, whose secrecy is not

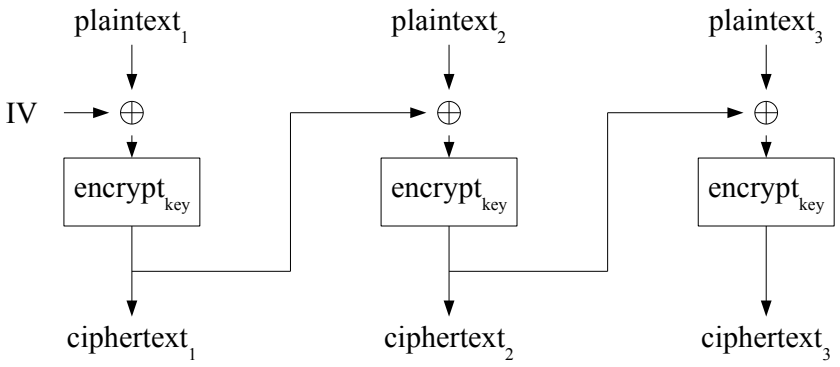


Figure 1.2: Encryption in Cipher Block Chaining Mode [36]

critical. Using different initial values ensures that the encryption of the same plaintext block always produces a different ciphertext block.

Commonly used block ciphers include 3DES and AES. Descriptions of these algorithms can be found in [36], and [11] gives a very good insight in the design process of AES. Most symmetric ciphers share a basic structure: The encryption and decryption process is divided in equal rounds which modify an internal state. A round consists of operations which involve the shuffling of bits and bytes, substitution of values, key-dependent modifications of the state. Generally speaking, adding more rounds yields higher security at the cost of losing performance.

1.2 Hashing Functions

A hashing function maps input data of arbitrary size, the *preimage*, to a fixed size output, the *hash value*. Hash values can be seen as fingerprints of their preimages and can be used to detect changes of the original data.

Cryptographically secure hash functions are one-way functions which have to be

- **Preimage Resistant**. It is hard to find a message with a given hash value.
- **Collision Resistant**. It is hard to find two messages with the same hash value.
- **Second Preimage Resistant**. Given one message, it is hard to find a second one with the same hash value.

The strongest requirement is that of collision resistance, and hash functions which are collision resistant are automatically preimage resistant and second preimage resistant. However, the birthday paradox shows that it is possible to find collision by just comparing the hash values of $O(2^{n/2})$ messages. As a consequence, a function which produces hash values with 160 bits has a security level of at most 80 bits.

Commonly used hashing functions are MD5 [32] and SHA1 [13]. They have much in common with block ciphers, and in fact, block ciphers can be used as hashing functions. MD5 and SHA1 also employ the concept of rounds which perform data-dependent modifications of an internal state.

1.2.1 Message Authentication Codes

A message authentication code is a piece of information which can be used to verify the integrity and the origin of a message. Hashing functions by themselves only allow to verify integrity but can be used to generate message authentication codes.

The idea is to concatenate the secret symmetric key k and the message m and to compute the hash value of this concatenation $h(k \parallel m)$. This message authentication code is sent along with the message and the recipient can check the integrity by performing the same computation. Since k must be known to generate the message authentication code and is only known to the two communicating parties, the receiver can also be sure of the origin of the message.

A similar method used in many protocols, including SSL, is HMAC which uses two nested hashing functions for extra security:

$$\text{HMAC} = h(k \parallel p_1 \parallel h(k \parallel p_2 \parallel m))$$

where p_1 and p_2 are strings used to pad the input of the hash function to a multiple of the block size.

1.3 Public-Key Cryptography

The major problem with symmetric cryptography is that of key distribution. In a larger environment, where potentially everybody wants to communicate with everybody else, each pair of communicating parties needs a different symmetric key. For example, when there are ten parties, the number of required keys is 45, and this number grows with the square of the number of parties.

Public-key cryptography solves the key distribution problem by enabling secure communication even between parties who have no prior knowledge about each other. Each party has two keys: A *private key* and a *public key*. While the private key must be kept secret, the public key can be made available to everybody, for example in a directory.

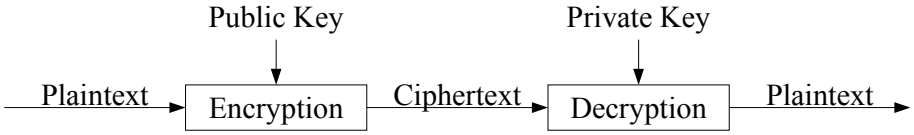


Figure 1.3: Model of an Asymmetric Cipher [36]

Figure 1.3 shows the basic principle of public-key cryptography. Given, for example, that Alice wants to send a message to Bob. She first looks up Bob's public key, encrypts her message with this key and sends it to Bob. Only Bob can decrypt this message using his own private key. But this scheme also poses a problem: The management of public keys. How can Alice be sure that she has Bob's public key and not the key of an adversary? A solution will be introduced in Section 1.4 which deals with public-key infrastructure and certificates.

Because public-key cryptography is rather computationally expensive it is common to use it only to exchange a symmetric key and to use symmetric ciphers to encrypt the rest of the communication. Other than for key exchange, public-key cryptography can be used to digitally sign documents (see Section 1.3.4).

The remainder of this section deals with the theoretical background of public-key systems and introduces some common systems.

1.3.1 One-Way Functions

A one-way function is a mathematical function with the property that its inverse is hard to compute. A *trapdoor one-way function* is a special kind of one-way function which has a trapdoor, and the knowledge of the trapdoor makes the computation of the inverse efficient. Functions of this kind form the basis of public-key cryptography.

On the one side, it is easy to compute $z = f(x)$, which is equivalent to the encryption of a message using the public key. On the other side, it is hard to compute the inverse $x = f^{-1}(z)$ to decrypt the message without knowledge of the trapdoor, the private key.

To measure the runtime of algorithms we define:

$$L_N(\alpha, \beta) = \exp((\beta + O(1))(\log N)^{\alpha}(\log \log N)^{1-\alpha})$$

Algorithms with a runtime of $O(L_N(0, \beta))$ run in polynomial time, and algorithms with a runtime of $O(L_N(1, \beta))$ run in exponential time. Runtimes where $0 < \alpha < 1$ are called sub-exponential.

Finding the inverse of a one-way function is a mathematical problem and public-key systems rely on the hardness of this problem.

Integer Factorization Problem

The integer factorization problem is defined as follows: Given N, find p and q such that $N = p \cdot q$. The fastest known algorithm to solve this problem is the number field sieve which has the sub-exponential runtime of the order $O(L_N(1/3, 1.923))$.

A related problem forms the basis of RSA: Given is e such that e and $(p-1)(q-1)$ are relatively prime. Also given is c, find m to satisfy

$$m^e = c \pmod{N}.$$

This problem can be reduced to the integer factorization problem. However it is possible that the RSA problem is in fact easier than factoring, although this remains an open question.

Discrete Logarithm Problem

Given an Abelian group G and two numbers $g, h \in G$, such that $h = g^x$, find x. The hardness of this problem heavily depends on the used group. When G is the multiplicative group of integers modulo a prime, the fastest known solution is the number field sieve with sub-exponential runtime. On the other side, for elliptic curve groups no sub-exponential algorithm is known.

1.3.2 Public-Key Encryption

Although it is possible to encrypt information solely by public-key cryptography, this is not very practical. Usually only a symmetric key is encrypted and transmitted to the other party.

RSA

RSA, named after its inventors Rivest, Shamir and Adleman, was one of the first public-key schemes invented in 1977. Surprisingly it is still one of the most widespread schemes today. The mathematical problem behind RSA is believed to be as hard as the integer factorization problem.

Key Generation Choose two large prime number p and q, and compute

$$N = p \cdot q.$$

Also choose an encryption exponent e, such that e and $(p-1)(q-1)$ is relatively prime. Common values for e are $3, 17$ and 65537. Perform a modulo inversion of e to compute the decryption exponent d:

$$d = e^{-1} \pmod{(p-1)(q-1)}.$$

(N, e) is the public key and (d, p, q) is the private key. Although p and q are not absolutely necessary anymore, they can be used to improve the performance of decryption.

Encryption and Decryption Transform the message so that it is represented by a number m, which must be less than N. Encrypt it as follows:

$$c = m^e \pmod{N}.$$

To decrypt the message, the secret decryption exponent d is necessary:

$$m = c^d \pmod{N}.$$

This technique works, since

$$c^d = (m^e)^d = m^{k(p-1)(q-1)+1} = m \cdot m^{k(p-1)(q-1)} = m \cdot 1 = m \pmod{N}.$$

A more detailed proof can be found in [36].

1.3.3 Key Exchange

In contrast to public-key encryption, key exchange schemes cannot be used to secure messages. Key exchange schemes only enable two parties to arrive at a shared secret value.

Diffie-Hellman

The Diffie-Hellman key exchange is based the discrete logarithm problem in a finite Abelian group G. The original design chooses G to be $\mathbb{F}_p^*$, but a more efficient version can be produced taking an elliptic curve group. In addition to the prime p a so-called generator g is required which can be shared by multiple users.
The key exchange works as follows:

Alice chooses a random number $a \in G$ and computes g^a.	Bob chooses a random number $b \in G$ and computes g^b.
Both parties exchange the values g^a and g^b	
Alice computes the secret key $k = (g^b)^a = g^{ba}$	Bob computes the secret key $k = (g^a)^b = g^{ba}$

One problem with this scheme is, that it is not possible to be sure that the other party really is who it pretends to be. In a protocol using the Diffie-Hellman scheme some kind of authentication mechanism is required.

1.3.4 Digital Signatures

A digital signature is a piece of information which allows to verify the authenticity of a message. In contrast to message authentication codes which work with secret symmetric keys, digital signatures allow everybody who has the public key of the sender to verify the authenticity of a message.

There are two general types of signatures: Signatures with appendix, where the signature is appended to the message, and signatures with message recovery, which allow to recover the message from the signature. Examples of these schemes are the digital signature algorithm and RSA, respectively.

RSA Signatures

The RSA algorithm presented above can be used to generate signatures. To sign a message apply the RSA decryption operation, where d denotes the private exponent:

$$s = m^d \pmod{N}$$

To recover the message, the receiver performs the RSA encryption operation:

$$m = s^e \pmod{N}$$

This algorithm by itself, however, does not allow to check the integrity of the message. To overcome this problem some redundancy must be added to the message before signing it, for example by adding a certain padding. After message recovery one can check whether the padding is correct.

The problem with this message recovery scheme is, that message sizes are restricted to the sizes of the RSA domain parameters. In practice, to sign messages of arbitrary size, one uses signatures with appendix by first computing the hash value of the message, signing the hash and transmitting the signature-message pair.

Digital Signature Algorithm

The disadvantage of RSA is that signature generation is a very costly process. Also, RSA signatures are very large. DSA, the digital signature algorithm, like the Diffie-Hellman key exchange, was originally proposed for the finite Abelian group $\mathbb{F}_p^*$. However the elliptic curve version of DSA, ECDSA, which is increasingly used produces smaller signatures and is faster than RSA.

There exist common domain parameters (p, q, g) which can be shared be multiple users. The requirements for these parameters can be found in [36]. A private key x is generated by choosing a random number, such that $0 < x < q$, and the associated public key is $y = g^x \pmod{p}$.

To sign a document, compute its hash value $h = H(m)$ and generate a random ephemeral key k, with $0 < k < q$. The signature is the pair (r, s), computed as follows:

$$\begin{aligned} r &= (g^k \pmod{p}) \pmod{q} \\ s &= (h + xr)/k \pmod{q} \end{aligned}$$

To verify this signature, compute the hash value of the message $h = H(m)$ and perform the following steps, where y is the public key of the signer:

$$\begin{aligned} a &= h/s \pmod{q} \\ b &= r/s \pmod{q} \\ v &= (g^a y^b \pmod{p}) \bmod q \end{aligned}$$

Accept the signature if and only if $v = r$.

1.4 Public-Key Infrastructure

As stated above, the main problem with public-key cryptography is how to be sure that a public key really belong to the communication partner.

Consider this simple example of a man-in-the-middle attack: Alice wants to obtain Bob's public key and asks Bob to send her the key. Eve, however, intercepts this transfer, keeps Bob's public key and sends her public key to Alice. Alice believes to have Bob's key, but she encrypts her message with Eve's key. When Eve intercepts the message, she decrypts it, reads it, encrypts it again using Bob's real public key, and sends the encrypted message to Bob. Neither Alice nor Bob would notice the attack.

The solution is that a public key is bound to the identity of its owner in a certificate, and this certificate is signed by a trusted third party. The trusted party is called Certificate Authority, or CA, and all users need to have a trusted copy of the CA's public key. In addition the CA must be trusted to do its job right. The basic structure of a certificate is shown in Figure 1.4.

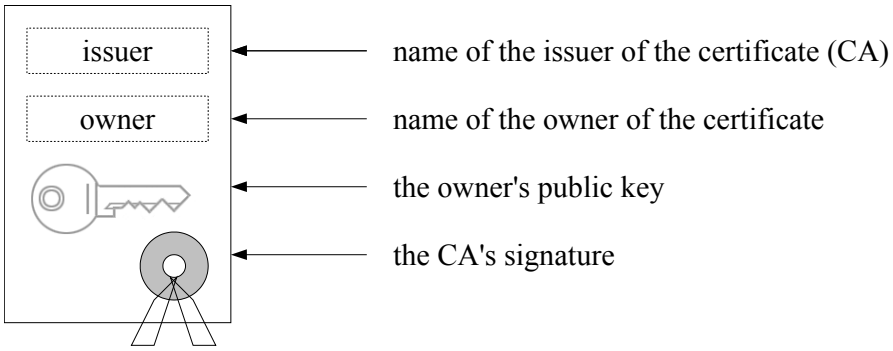


Figure 1.4: Structure of a Certificate

In practice this scheme is not so simple, since one single CA cannot possibly sign every certificate. A certification hierarchy is formed, where one or more top-level CAs issue certificates for a number of sub-level CAs, and sub-level CAs issue certificates for further CAs or individual subjects (see Figure 1.5).

1.4.1 X.509 Certificates

X.509 is a very common format for certificates, defined by the International Telecommunication Union. An X.509 certificate contains the following elements [23]:

- The version of the X.509 standard in use.

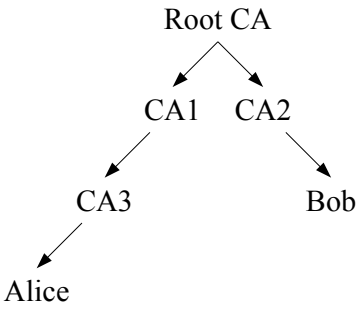


Figure 1.5: Example of a Certification Hierarchy [36]

- The serial number of the certificate.
- The identifier of the signature algorithm.
- The name of the issuing CA.
- The beginning and the end date of the validity period.
- The subject's name, in the form of an email address or domain name.
- The subject's public key, including the identifier of the algorithm the key can be used for and the associated domain parameters.
- Optional extensions.
- The CA's signature of the information listed above.

1.4.2 Revocation of Certificates

Although certificates contain an expiration date, in some cases it is necessary to invalidate them earlier. For example when the owner's private key is compromised or when an employee leaves the company. The X.509 standard provides for so-called certificate revocation lists as a means for the CA to inform all users about invalidated certificates. Such a list contains the serial numbers of revoked certificates whose validity period has not yet expired, signed by the CA. The distribution of certificate revocation lists poses the biggest problem, since usually users have a long list of trusted CAs and it is difficult the maintain up-to-date lists of all of them.

Chapter 2

Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is a public-key cryptosystem which operates on a set of points on an elliptic curve. Although the mathematics of elliptic curves has been studied for centuries, its use for cryptography was first proposed by Neal Koblitz and Victor Miller in 1985.

Elliptic curves are plane curves over a field K, defined by the Weierstrass equation $E : y^2 = x^3 + ax + b$, with $a, b \in K$. A point $P = (x_P, y_P)$ is on the curve when it satisfies this equation. The introduction of a "+" operation turns the elliptic curve into an Abelian group, and allows for point additions and doublings.

The main cryptographic operation is the *point multiplication*, which computes $Q = kP$, where Q and P are points on the curve, and k is a scalar. This multiplication can be done as a series of point additions and doublings. For instance, $Q = 7P$ can be expressed as $Q = (2((2P) + P)) + P$.

Most of the information in this chapter is taken from Hankerson, Menezes and Vanstone's "Guide to Elliptic Curve Cryptography" [22], and further references to this book are omitted. An exhaustive mathematical background is out of the scope of this text and can be found in this book. In this chapter the use of elliptic curve cryptography is motivated by a comparison with other public-key cryptosystems. Elliptic curve cryptography is presented in a bottom-up approach, starting with the underlying finite fields. Then, algorithms to perform elliptic curve arithmetic are presented and finally cryptographic schemes based on ECC are introduced. Figure 2.1 gives an overview of the layered implementation of an ECC-based algorithm.

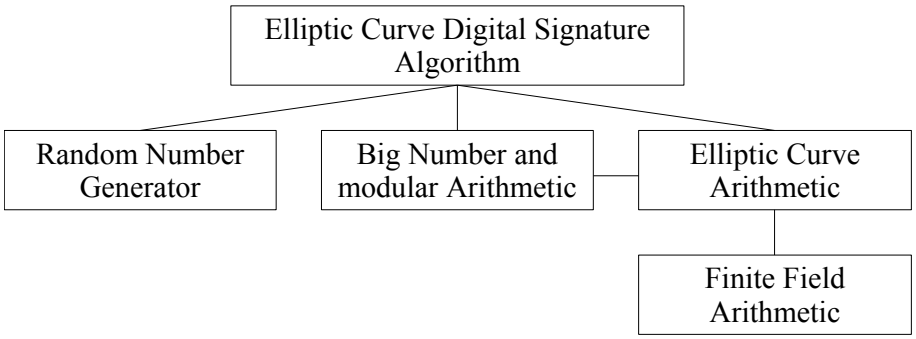


Figure 2.1: Hierarchy of Modules Required to Support ECDSA [22]

2.1 Comparison of ECC With Other Cryptosystems

The security of a public-key cryptosystem is determined by the hardness of the underlying mathematical problem. This hardness has direct impact on the performance, as it dictates the size of domain parameters and keys. Table 2.1 compares common cryptosystems with respect to their underlying problem.

Table 2.1: Comparison of Cryptosystems Based on the Mathematical Problem They Are Built Upon

Public-key system	Underlying mathematical problem	Best known algorithm	Runtime
RSA	Integer factorization problem	Number field sieve	sub-exponential
DH, ElGamal	Discrete logarithm problem	Number field sieve	sub-exponential
ECC (ECDH, ECDSA)	Elliptic curve discrete logarithm problem	Pollard-rho algorithm	fully exponential

Elliptic curve cryptography relies on the hardness of the *elliptic curve discrete logarithm problem* (ECDLP). That is, given the two points P and Q, to find the integer k which satisfies the equation $Q = kP$.

In today's Internet security protocols, RSA is dominant. However, ECC can offer equivalent levels of security with smaller keys and less computational expense. The reason is that there exist algorithms with sub-exponential runtime to solve the integer factorization problem, the foundation of RSA. On the other hand, the fastest known algorithm to solve the the elliptic curve discrete logarithm problem has fully exponential runtime. As a consequence, smaller domain parameters and smaller keys are sufficient to provide the same level of security. This is especially important for resource-constrained devices like PDAs, smart cards, and wireless sensors.

Furthermore, the advantages of ECC increase as security needs grow [41]. Table 2.2 compares the required key sizes of RSA and ECC for different levels of security. For

Table 2.2: Comparison of Key Sizes for Equivalent Levels of Security

Security (bits)	Symmetric encryption algorithm	Minimum size (bits) of the public key	
		RSA	ECC
80		1024	160
112	3DES	2048	224
128	AES-128	3072	256
192	AES-192	7680	384
256	AES-256	15360	512

example, when moving from 80 bits to 112 bits, the RSA key size grows by a factor of 2,

while the ECC key size only grows by a factor of 1.4. The runtime of both algorithms is of the order $O(n^3)$, where n denotes the size of the key. Thus there is an 8-fold increase of runtime when using RSA, but only a 2.7-fold increase when using ECC.

Gura *et al.* compared implementations of RSA and ECC on an 8-bit microcontroller, and found that a point multiplication over a 160-bit prime field is 13 times faster than a 1024-bit RSA decryption. At the same time, a point multiplication over a 224-bit field is almost 38 times faster then a 2048-bit RSA decryption [21].

2.2 Finite Fields

In the process of implementing elliptic curve cryptography, a lot of design decisions must be made, based on the intended applications and the target system. Numerous different algorithms exist for each operation in elliptic curve arithmetic and in finite field arithmetic. A significant influence on the speed of the implementation has the choice of the finite field underlying the elliptic curve.

A field is a set $\mathbb{F}$, together with two operations, addition ("+") and multiplication ("$\cdot$"), that have to satisfy certain arithmetic properties. If the set $\mathbb{F}$ is finite, the field is said to be finite. Finite fields are also referred to as Galois fields, after their inventor, the mathematician Evariste Galois, and the abbreviation "GF" is common.

In elliptic curve cryptography, the following fields are used:

Prime Fields ($\mathrm{GF}(p)$) are all integers in the range $[0, p-1]$, where p is a prime. Operations in prime fields are performed modulo p. Prime field arithmetic can be efficiently implemented in software on processors with hardware support for regular integer multiplication.

Binary Fields ($\mathrm{GF}(2^m)$) consist of binary polynomials of degree less than m. The coefficients of these polynomials can either be 0 or 1, such that each coefficient can be represented by a bit. Operations in binary fields are performed modulo an irreducible polynomial $p(x)$. For an efficient implementation, binary field arithmetic requires special hardware which is usually not present in general-purpose microprocessors.

Optimal Extension Fields ($\mathrm{GF}(p^m)$) combine properties of prime fields and binary fields. The elements are polynomials of degree less than m, having prime-field coefficients. The size of the coefficients can be chosen to fit the word size of a general-purpose processor to allow for an efficient implementation.

Curves which are recommended by standards organizations are defined over prime fields or binary fields. Because arithmetic in prime fields is more efficient than arithmetic in binary fields when no special hardware is present, this section deals only with prime fields.

The elements of prime fields which are used in cryptography have a size of 160 bits or more and do not fit into a single processor word. In this section, the wordsize of the processor is referred to as W. When $m = \log_2 p$ is the bitlength of p, a number a can be represented by an array $A = (A[t-1], \ldots, A[2], A[1], A[0])$. The array consists of $t = \lceil m/W \rceil$ W-bit words and has the following semantic:

$$a = 2^{(t-1)W} A[t-1] + \cdots + 2^{2W} A[2] + 2^W A[1] + A[0]$$

2.2.1 Addition and Subtraction

Addition and subtraction in prime fields is similar to multiprecision operations for integers. Care must be taken that the result is in the range $[0, p-1]$, by adding or subtracting the prime, if necessary. The following two algorithms perform addition and subtraction and GF(p).

Algorithm 2.1: Addition in GF(p)

Input: Operands a, b, modulus p
Output: $c = a + b \pmod{p}$

1 $(\varepsilon, C[0]) \leftarrow A[0] + B[0]$
2 **for** i *from* 1 *to* $t-1$ **do**
3 $\quad (\varepsilon, C[i]) \leftarrow A[i] + B[i] + \varepsilon$
4 **if** $\varepsilon = 1$ *or* $c \geq p$ **then** $c \leftarrow c - p$
5 **return** c

Algorithm 2.2: Subtraction in GF(p)

Input: Operands a, b, modulus p
Output: $c = a - b \pmod{p}$

1 $(\varepsilon, C[0]) \leftarrow A[0] - B[0]$
2 **for** i *from* 1 *to* $t-1$ **do**
3 $\quad (\varepsilon, C[i]) \leftarrow A[i] - B[i] - \varepsilon$
4 **if** $\varepsilon = 1$ **then** $c \leftarrow c + p$
5 **return** c

ε is a single-bit variable which stores the carry of the addition or the borrow of the subtraction. Architectures of general-purpose processors store the carry or borrow in a hardware register and the operations in lines 2 and 4 require only a single instruction.

2.2.2 Multiplication and Reduction

Prime field multiplication of two numbers a and b can be accomplished by first performing a regular multiprecision multiplication and then reducing the result modulo p.

Multiplication

The two fundamental methods to do the multiplication are the operand scanning and the product scanning method, both requiring t^2 single-precision multiplications. The relative performance of these methods strongly depends on the architecture of the multiplier implemented in the processor.

In the following algorithm, U refers to an accumulator of the size $2W$, which is large

enough to hold the result of a $W \times W$-bit multiplication.

Algorithm 2.3: Multiprecision Multiplication in Operand Scanning Form

Input: Operands a, b
Output: $c = a \cdot b$
1 **for** i *from* 0 *to* $t-1$ **do** $C[i] \leftarrow 0$
2 **for** i *from* 0 *to* $t-1$ **do**
3 $\quad U \leftarrow 0$
4 $\quad$ **for** j *from* 0 *to* $t-1$ **do**
5 $\quad\quad U \leftarrow U + C[i+j] + A[i] \cdot B[j]$
6 $\quad\quad C[i+j] \leftarrow U \bmod 2^W$
7 $\quad\quad U \leftarrow U/2^W$
8 $\quad C[i+j] \leftarrow U \bmod 2^W$
9 **return** c

For the algorithm in product scanning form, the accumulator must be bigger than $2W$ bits, such that the results of multiple multiplications can be summed up.

Algorithm 2.4: Multiprecision Multiplication in Product Scanning Form

Input: Operands a, b
Output: $c = a \cdot b$
1 $U \leftarrow 0$
2 **for** k *from* 0 *to* $2t-2$ **do**
3 $\quad$ **foreach** (i, j) *where* $i + j = k$ *and* $0 \leq i, j < t$ **do**
4 $\quad\quad U \leftarrow U + A[i] \cdot B[j]$
5 $\quad C[k] \leftarrow U \bmod 2^W$
6 $\quad U \leftarrow U/2^W$
7 $C[2t-1] \leftarrow U \bmod 2^W$
8 **return** c

The multiplication algorithms can also be used for squaring, but a slight variation of Algorithm 2.4 can yield higher performance. In line 4, every single element of A is multiplied by every single element of B, for instance $A[i] \cdot B[j]$ and $A[j] \cdot B[i]$. When squaring, however, A and B are equal and pairs of multiplications can be combined when $i \neq j$. In this example only $2 \cdot A[i] \cdot A[j]$ needs to be calculated. With this improvement it is possible to avoid almost half of the multiplications; the exact number of multiplications is $\frac{t^2+t}{2}$.

A different algorithm, the Karatsuba-Ofman multiplication, is a divide-and-conquer approach where the multiplication of large integers is reduced to three multiplication of smaller integers, two additions, and two subtractions. This reduction can be done recursively until the operands match the wordsize of the processor or until one of the algorithms described above has higher performance. The Karatsuba-Ofman method reduces the number of single-precision multiplications from t^2 to $t^{\log_2 3}$. The disadvantage is that the implementation is more complex and larger, and more memory is required to store the intermediate results.

Reduction

The computation of $r = z \bmod p$ is called reduction. A multiplication or squaring results in a number z twice as large as the prime, and z must be reduced such that $z < p$ and can be used as a operand and subsequent prime field operations. There are various methods to reduce an integer, ranging from very generic but slow to very specialized and fast.

The generic reduction operation divides an integer z by a prime p, and returns the remainder r as result. Division, however, is a very expensive operation and typically is 10 to 100 times slower than a multiprecision multiplication. Two other algorithms which are faster than division are the Barrett reduction and the Montgomery reduction, which are explained in [22]. Both methods are generic in that they do not exploit the form of the prime p.

Prime fields which are recommended by standards usually have moduli which allow for a more efficient reduction. Examples of such moduli used in elliptic curves, recommended by SEC [7], are:

$$\begin{aligned} p_{160} &= 2^{160} - 2^{31} - 1 \\ p_{192} &= 2^{192} - 2^{64} - 1 \end{aligned}$$

The following is a fast algorithm to reduce modulo $p = 2^n - c$, where c is an l-bit positive integer for some $l < n$.

Algorithm 2.5: Fast Reduction Modulo $2^n - c$

Input: Number to reduce z
Output: $r = z \bmod (2^n - c)$

1 **while** $z/2^n > 0$ **do**
2 $\quad z \leftarrow (z \bmod 2^n) - c \cdot (z/2^n)$
3 **if** $z \geq p$ **then** $r \leftarrow z - p$ **else** $r \leftarrow z$
4 **return** r

The algorithm is based on the fact that $z \equiv z - mp \pmod{p}$, for arbitrary integers m, and chooses m to eliminate as many digits of z as possible. When $l \leq n/2$ and the length of z is at most $2n$ bits then step 2 is executed at most twice.

2.2.3 Inversion

The multiplicative inverse of a prime field element a is denoted by a^{-1} and has the property that $a^{-1} \cdot a \equiv 1 \pmod{p}$. When a and p have no common divisors, the inverse can be calculated using the equality $a^{-1} \equiv a^{p-2} \pmod{p}$. This method is quite inefficient as it requires to perform an exponentiation, which costs $\log_2(p-2)$ squarings and, on average, $0.5 \cdot \log_2(p-2)$ multiplications.

A more efficient algorithm can be derived from the Euclidean algorithm to compute the greatest common divisor $d = \gcd(a, b)$. Each step of the Euclidean algorithm reduces the problem of finding $\gcd(a, b)$ to the problem of finding $\gcd(b - n \cdot a, a)$, where n is an integer value. After some iterations one finally arrives at $\gcd(0, d) = d$. The Euclidean algorithm to compute $\gcd(a, p) = 1$ can be extended by two additional variables x and y, which are updated in each iteration to maintain the following invariant: $a \cdot x + p \cdot y = d \pmod{p}$. After the final step one arrives at $a \cdot a^{-1} + p \cdot 0 = 1 \pmod{p}$. Algorithm 2.6 is further modified from the original Euclidean algorithm to contain only divisions by 2,

which can be done efficiently with shift operations.

Algorithm 2.6: Binary Algorithm for Inversion in GF(p)

Input: Prime p, number to invert $a \in [1, p-1]$
Output: $a^{-1} \pmod p$

```
1  u ← a,  v ← p
2  x1 ← 1,  x2 ← 0
3  while u ≠ 1 and v ≠ 1 do
4      while u is even do
5          u ← u/2
6          if x1 is even then x1 ← x1/2 else x1 ← (x1 + p)/2
7      while v is even do
8          v ← v/2
9          if x2 is even then x2 ← x2/2 else x2 ← (x2 + p)/2
10     if u ≥ v then u ← u − v,  x1 ← x1 − x2  (mod p)
11     else v ← v − u,  x2 ← x2 − x1  (mod p)
12 if u = 1 then return x1 else return x2
```

2.3 Elliptic Curves

This section gives the definition of elliptic curves and deals with arithmetic in curves over prime fields. Finally, algorithms for the elliptic curve point multiplication are presented and compared.

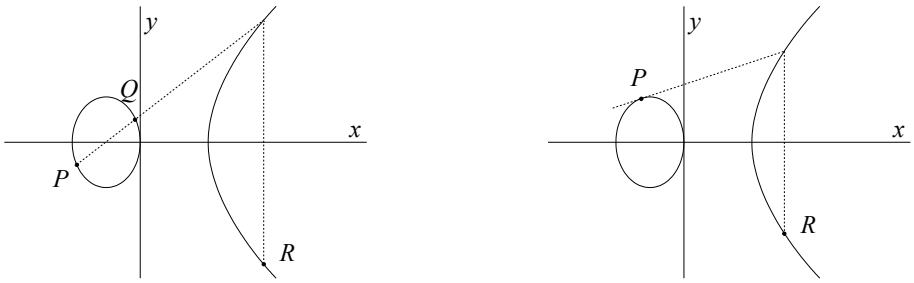
2.3.1 Elliptic Curve Groups

Elliptic curves over prime fields are defined by the Weierstrass equation:

$$\mathrm{EC}(\mathrm{GF}(p)) : y^2 = x^3 + ax + b. \qquad a, b \in \mathrm{GF}(p)$$

A point $P = (x_P, y_P)$ is on the curve when it satisfies this equation. A rule for adding points can be defined to form an Abelian group, with ∞ serving as the identity element. The negative of a point $P = (x_P, y_P)$ is $-P = (x_P, -y_P)$.

The rule for adding is a so-called *chord-and-tangent* rule (see Figure 2.2). Two points



(a) Addition: P+Q=R. (b) Doubling: P+P=R.

Figure 2.2: Addition and Doubling of Points on Elliptic Curves [22]

P and Q lie on a line, which intersects the curve in a third point, R'. The result of the addition R is the reflection of R' about the x-axis.

In the case $P = Q$, the tangent of the curve at this point is taken. The tangent, again, intersects the curve in another point whose reflection is the result R.

2.3.2 Elliptic Curve Domain Parameters

An elliptic curve is specified by the type and structure of the underlying finite field, and by the curve parameters a and b. The complete set of domain parameters is listed in Table 2.3.

Table 2.3: Elliptic Curve Domain Parameters

Parameter	Name	Description
FT	Field type	"prime field" or "binary field"
q	Field parameter	Prime field: Prime p. Binary field: Irreducible polynomial $p(x)$
S	Seed	Random seed used for generating parameters
a, b	Curve parameters	Define the curve equation
$P = (x_P, y_P)$	Base point	Base point in affine coordinates
n	Order of base point	$n = \|P\|$
h	Cofactor	$h = \|EC(GF(q))\|/n$

The seed S is the random seed which has been used to generate the curve parameters. With this value it is possible to prove that the parameters are randomly generated and not deliberately chosen to produce a curve with certain weaknesses.

The base point P is generator of a cyclic group which is a subgroup of EC(GF(q)). The number of elements of the cyclic group is called order of P, and is denoted by n. The group consists of all points which can be generated by multiplying kP, with $k \in [0, n-1]$, and has the property that $nP = \infty$. To provide sufficient security, n should be at least 2^{160}.

The cofactor h is the ratio between the order of the curve (the number of points on the curve) and the order of the subgroup generated by the base point. Typical values for the cofactor are 2, 3, and 4.

In order to be secure, the domain parameters must be chosen such that the curve is not vulnerable to known attacks. Verification of domain parameters is a costly process and therefore not done online. Various standards, like SEC [7], recommend domain parameters for elliptic curves which are proven to be secure. Additionally, these parameters have the advantage that they are optimized for high-performance implementations.

2.3.3 Point Representation

The implementation of the chord-and-tangent rule for adding and doubling points on elliptic curves requires multiple arithmetic operations in the underlying field. The representation of points influences the kind and number of finite-field operations, and thus influences the performance.

Affine coordinates are most intuitive. Points are represented by their x and y coordinate: $P = (x, y)$. The resulting algorithms for adding and doubling points, however,

contain a finite-field inversion. The inversion is a very inefficient operation and should be avoided.

Projective coordinates use three instead of two coordinates: $P = (x, y, z)$. A point $P = (x, y, z)$ in projective coordinates corresponds to the point $P = (x/z^c, y/z^d)$ in affine coordinates, where c and d are small constants (1, 2, or 3). *Jacobian coordinates* are projective coordinates with $c = 2$ and $d = 3$, and allow for an efficient implementation of point addition and doubling.

Table 2.4 compares the different point representations based on the number and kind of finite-field operations required for point addition and doubling. The comparison shows that the point addition and is most efficient when Jacobian and affine coordinates are mixed.

Table 2.4: Operation Count for Point Addition and Doubling on $y^2 = x^3 - 3x + b$. A=affine, J=Jacobian; I=inversion, M=multiplication, S=squaring.

Coordinates	$2 \cdot P_1$	$P_1 + P_2$	$P_1 + P_2$ mixed
Affine	1I, 2M, 2S	1I, 2M, 1S	-
Jacobian	4M, 4S	12M, 4S	8M, 3S

2.3.4 Point Multiplication

Point multiplication, the computation of $Q = kP$, is the foundation of all ECC-based cryptographic schemes. The multiplication can be performed by a sequence of point doublings and optional additions, as outlined in Algorithm 2.7

Algorithm 2.7: Left-To-Right Binary Point Multiplication

Input: Binary representation of $k = (k_{t-1}, \ldots, k_1, k_0)$, point P
Output: $Q = kP$

```
1 Q ← ∞
2 for i from t − 1 down to 0 do
3 |   Q ← 2 · Q
4 |   if k_i = 1 then Q ← Q + P
5 return Q
```

At the occurrence of the first $k_i = 1$, Q is initialized with P. After that, Q is doubled i times and finally contains the partial product $2^i \cdot (k_i \cdot P)$. By the conditional addition of P whenever $k_i = 1$, all partial products are accumulated in Q, such that in the end $Q = \sum_{i=0}^{t-1} 2^i \cdot (k_i \cdot P) = k \cdot P$.

This algorithm requires t point doublings and on average $t/2$ additions. It is possible to cut down on the number of additions by first transforming k into its non-adjacent form (NAF). The digits of a non-adjacent form can be 0, 1, or -1, and no two consecutive digits are nonzero. Actually, on average, only every third digit is nonzero, such that the expected number of additions is $t/3$.

Since a NAF can contain the digit -1, it becomes necessary to perform point subtractions. Subtractions are done by adding $-P$, the computation of which is very efficient in curves over prime fields. Algorithm 2.8 is a slight variation of the binary point multipli-

cation; here k is given in its non-adjacent form.

Algorithm 2.8: Left-To-Right Binary NAF Method for Point Multiplication

Input: NAF of $k = (k_{t-1}, \ldots, k_1, k_0)$, point P
Output: $Q = kP$

```
1 Q ← ∞
2 for i from t − 1 down to 0 do
3 |   Q ← 2 · Q
4 |   if k_i = 1 then Q ← Q + P
5 |   if k_i = −1 then Q ← Q − P
6 return Q
```

Algorithm 2.9 computes the NAF of an integer value. The result is a pair of numbers, (d, s), where the bit d_i contains the absolute value of the i-th digit of NAF(k), and s_i contains the sign ($0 \equiv$ " $+$ " and $1 \equiv$ " $-$ ").

Algorithm 2.9: Computing the NAF of a Positive Integer

Input: Integer k
Output: $(d, s) = \text{NAF}(k)$

```
1 d ← (3 · k) ⊕ k ;      /* ⊕ denotes the bit-wise exclusive-or operation */
2 s ← k
3 return (d, s)
```

The NAF described so far is a width-2 NAF (NAF_2), meaning that two consecutive digits are zero. This principle can be generalized to greater widths to further reduce the number of additions. When using NAFs with a width greater than two it is necessary to precompute multiples of P which is a one-time cost when the same point is the operand of multiple point multiplications. The disadvantage is that additional memory is required to store the precomputed points. To precompute points for a width-w NAF (NAF_w), one point doubling and $A = 2^{w-2} - 1$ point additions are required, and additional memory to store $2^{w-2} - 1$ points is needed. The number of point additions in the multiplication can be reduced to $t/w + 1$, where t denotes the bit-length of the scalar.

2.4 Cryptographic Algorithms Based on ECC

The two main applications of public-key cryptography are key exchange and digital signatures. Public-key encryption is possible, but very inefficient, and key exchange in combination with symmetric encryption is used instead.

In the case of elliptic curve cryptography, the Elliptic Curve Diffie-Hellman algorithm (ECDH) or a variation thereof is commonly used for key exchange, and the Elliptic Curve Digital Signature Algorithm (ECDSA) is used to generate and verify digital signatures.

2.4.1 Elliptic Curve Keys

An elliptic curve key pair consists of a scalar private key d, and a public key Q which is a point on the curve.

Algorithm 2.10: Key Pair Generation

Input: Base point P, order $n = |P|$
Output: Key pair (Q, d)
1 randomly choose $d \in [1, n-1]$
2 $Q \leftarrow dP$
3 **return** (Q, d)

For the generation of the private key a cryptographically secure random number generator must be used. The computation of the private key, when the public key is given, is exactly the elliptic curve discrete logarithm problem and is thus intractable.

2.4.2 Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman (ECDH) is based on the problem of finding C, when the points A, B and P are given, and the following equations hold: $A = aP$, $B = bP$ and $C = abP$. This problem is not harder than ECDLP, because the knowledge of the discrete logarithms a and b leads to the knowledge of C.

In a basic Diffie-Hellman key exchange, both parties exchange their public keys A and B. Now, each party can use the own private key to arrive at the same secret value C.
Party A computes $C = aB = abP$.
Party B computes $C = bA = abP$.

In practice ephemeral key pairs [1] are used and the exchange of public keys is authenticated. ECMQV is an example for such a scheme which is included in many international standards.

2.4.3 Elliptic Curve Digital Signature Algorithm

The elliptic curve digital signature algorithm (ECDSA) produces digital signatures with appendix. This means that the message remains unaltered and a signature is appended. To avoid restrictions on the message size, the fixed-length SHA1 hash of the message is signed.

The following two algorithms specify how signatures are created and verified. For a proof of correctness and remarks on security refer to [22].

[1] An ephemeral key is a key which is only used once.

Algorithm 2.11: ECDSA Signature Generation

Input: Base point P, order $n = |P|$, private key d, message m
Output: Signature (r, s)

randomly choose $k \in [1, n-1]$
$R = (x_R, y_R) = kP$
$r = x_R \bmod n$
if $r = 0$ **then** goto 1
$e = \text{Hash}(m)$
$s = k^{-1}(e + dr) \pmod{n}$
if $s = 0$ **then** goto 1
return (r, s)

Algorithm 2.12: ECDSA Signature Verification

Input: Base point P, order $n = |P|$, public key Q, message m, signature (r, s)
Output: Accept/Reject

if $r \notin [1, n-1]$ *or* $s \notin [1, n-1]$ **then return** *"Reject"*
$e = \text{Hash}(m)$
$w = s^{-1} \pmod{n}$
$u_1 = ew \pmod{n}$
$u_2 = rw \pmod{n}$
$X = (x_X, y_X) = u_1 P + u_2 Q$
if $X = \infty$ **then return** *"Reject"*
$v = x_X \pmod{n}$
if $v = r$ **then return** *"Accept"* **else return** *"Reject"*

Chapter 3

SSL

SSL (Secure Sockets Layer) [15] is a security protocol, designed with the goal to provide privacy and reliability between two communicating parties. It is widely used in applications like e-commerce and Internet banking. SSL offers encryption, authentication and integrity protection on top of a streaming data-transport mechanism like TCP.

The SSL protocol has been developed by Netscape and never became an Internet standard. Standardization of SSL is now being done under the name TLS [1] (Transport Layer Security) by the Internet Engineering Task Force. There are only minor changes between SSL 3.0 and TLS 1.0 and a single implementation can easily support both protocols. The following information is true for both versions, and the name SSL is used.

SSL is very flexible, as it allows communicating parties to negotiate a set of cryptographic algorithms, called *cipher suite*, which determines the method of key exchange, signature verification, encryption and message authentication. Cipher suites which take advantage of ECC are originally not included in the specification. However, there exists a proposal of how to incorporate ECC into the SSL protocol [18].
Examples for cipher suites are:

- **SSL_RSA_WITH_RC4_128_MD5.** This cipher suite specifies that the certificate is signed with the RSA algorithm, RSA is used for key exchange, RC4 with a 128-bit key is used for symmetric encryption, and the MD5 hashing algorithm is used for message authentication.
- **TLS_ECDH_ECDSA_WITH_RC4_128_SHA.** Here, the certificate is signed with ECDSA, the algorithms for key exchange is ECDH, RC4 is used for symmetric encryption, and the SHA1 hashing algorithm is used for message authentication.

The structure of the SSL protocol is depicted in Figure 3.1. The main components are the record layer protocol and the handshake protocol. In the handshake, both parties negotiate a cipher suite, authenticate each other and establish a *shared secret* using public-key cryptography. The record layer protocol is responsible for fragmentation and defragmentation of SSL records and for symmetric cryptography once the handshake is completed. Further there is an alert protocol to transmit notifications about errors and about the closure of the connection. The change cipher spec protocol is used to change the active cipher suite.

The remainder of this chapter elaborates on the record layer protocol and on the handshake protocol, and gives an analysis of the security provided by SSL.

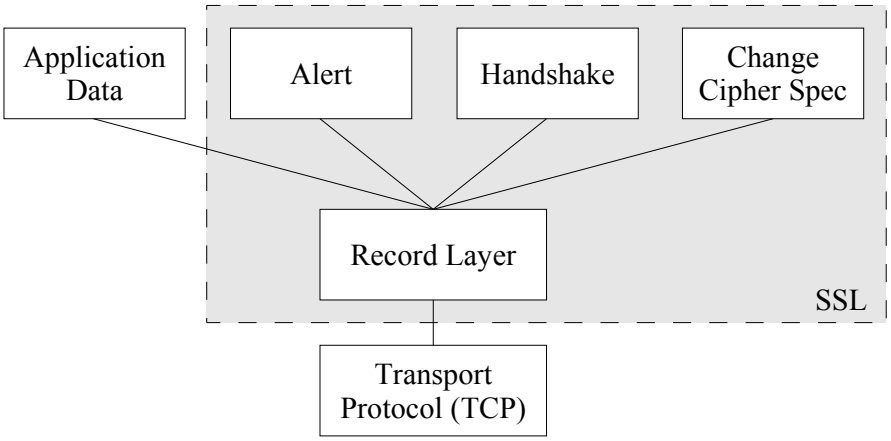


Figure 3.1: Structure of the SSL Protocol

3.1 Record Layer Protocol

The record layer takes data in blocks of arbitrary sizes from the application or from one of the other parts of the SSL protocol. The record layer is then responsible for fragmenting the data to fit into SSL records. An SSL record consists of a header and the data payload, which includes a message authentication code. For communication in the other direction, the record layer defragments incoming SSL records and passes the data blocks to the appropriate higher layer.

After the exchange of change cipher spec messages, the negotiated cipher suite is active and the record layer additionally performs cryptographic operations on the data, namely it computes a keyed message authentication code and encrypts the data. The message authentication code is generated from the hash of the plaintext, the shared secret, the data length, a sequence number, and some padding bytes. The record layer appends the result to the data and encrypts the whole record using the active symmetric cipher.

3.2 Handshake Protocol

The handshake protocol sits on top of the record layer and allows communicating parties to negotiate session parameters and to exchange symmetric keys. This section explains the handshake with focus on elliptic curve cryptography, outlines the differences of an RSA handshake, and introduces the abbreviated handshake, which is based on the reuse of a previous session.

3.2.1 ECC Handshake

The handshake layer is a state machine and the sending and receiving of a handshake message causes a transition from one state to another. The reception of an unexpected or corrupted handshake message causes the handshake to fail. Figure 3.2 depicts the message flow of a minimal handshake when an ECC cipher suite is used. This figure does not contain messages required to support client authentication, since this feature is not commonly used.

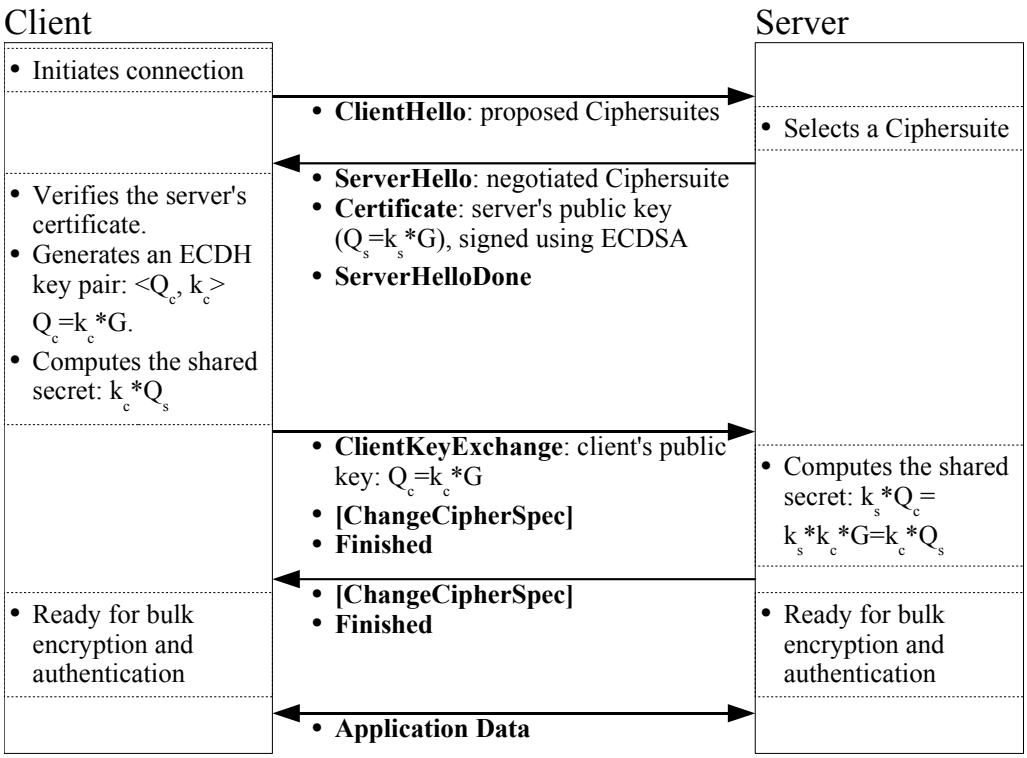


Figure 3.2: Minimal ECC-Based SSL Handshake

ClientHello The *ClientHello* message is the first message the client sends when the underlying connection is established. The main purpose of this message is to inform the server about the client's capabilities, so that both parties can agree on common parameters. The message contains the highest supported version of the SSL protocol, a list of cipher suites and compression algorithms the client can handle, and random data for the key exchange process. Additionally the client can specify the ID of a previous session to abbreviate the handshake (see Section 3.2.3).

When the server receives this message, it selects the highest version of the SSL protocol which is supported by both parties. It parses the list of cipher suites and compression algorithms and selects one that it supports. When there are no common cipher suites or compression algorithms, the server sends an alert message and the handshake fails.

To provide backward compatibility with SSL 2.0, clients often send a *ClientHello* message in the old format, but specify 3.0 as the highest supported version. Servers should be able to process *ClientHello* messages in both formats, but SSL 3.0 should be used for the rest of the handshake.

ServerHello The *ServerHello* message is the server's response to the *ClientHello* message and contains the version of the SSL protocol, the cipher suite and the compression algorithm which will be used.

Certificate The *Certificate* message contains an X.509 certificate, signed by a certificate authority using the algorithm specified in the chosen cipher suite. The recipient verifies the certificate and may terminate the connection when the verification fails.

ServerKeyExchange The *ServerKeyExchange* message is only used when the server has no certificate or when the certificate does not contain a public key suitable to perform the key exchange. In this case the *ServerKeyExchange* message contains an appropriate public key.

CertificateRequest By sending the *CertificateRequest* message, the server can optionally request a certificate from the client. If possible, the client responds with a *Certificate* message or it alerts the server that no certificate is available. This method of client authentication is not widely used in practice and it is more common to handle authentication on the application level by prompting the user for username and password.

ServerHelloDone The *ServerHelloDone* indicates the end of the server's *Hello* and associated messages. After sending, the server waits for the client's response.

ClientKeyExchange The *ClientKeyExchange* message contains the client's public key. After receiving the server's certificate, which contains the server's public key and the elliptic curve parameters, the client generates a new ECDH key pair on the same curve. Then the client computes the shared secret by multiplying the own private key by the server's public key. The server, when it receives this message, can compute the same secret by multiplying the own private key by the client's public key. Both parties use the shared secret and the random data from the *Hello* messages to derive the session keys for symmetric encryption and authentication.

CertificateVerify The *CertificateVerify* message is sent by the server only when the client supplied a certificate. It contains the hash of the shared secret and of all previous handshake messages. The purpose is to allow for explicit verification that the client's certificate has been transmitted correctly.

ChangeCipherSpec The *ChangeCipherSpec* message is actually not part of the handshake protocol; it is handled in the record layer. However, it makes more sense to describe it in the context of the handshake. The effect of this message is, that the previously negotiated ciphers are activated, such that all subsequent SSL records will be encrypted accordingly. It is important that sending this message only activates encryption for data flowing in one direction, and therefore, both parties have to send this message. For example, when the client sends this message, all subsequent SSL records which originate at the client will be secured. The server still has to send it to secure communication in the other direction.

Finished The *Finished* message is the first message after a *ChangeCipherSpec* and therefore the first encrypted message. It contains the hash of all previous handshake messages and is used to verify that the handshake was successful and that no handshake massages have been altered. After this message, the connection is ready for application data.

3.2.2 Differences Between ECC and RSA Handshakes

The only differences between the ECC and the RSA handshake lie in the certificate and in the handling of the *ClientKeyExchange* message.

In the case of RSA, the certificate contains an RSA public key and is signed with RSA. When ECC is used, the certificate contains an ECDH-capable public key and is signed using ECDSA.

In both cases the *ClientKeyExchange* message is used to establish a shared secret, known only to the parties involved in the handshake. When an ECC cipher suite is used, the client generates a random ECDH key pair and sends the public key in the *ClientKeyExchange* message. However, when RSA is used, the client generates a random shared secret, encrypts it with the server's public key, and sends the encrypted secret. The server decodes the message with its own private key and arrives at the same value.

3.2.3 Session Reuse

Because public-key operations are expensive, the SSL protocol supports a mechanism called *session reuse*. Session reuse allows the communicating parties to reuse the parameters of a previous session without the need for a full handshake. Thus the server is required to maintain a session cache. When a client reconnects to the server, the client can propose to reuse a session by specifying the session ID in the *ClientHello* message. When the server is able to do so, it includes the same ID in the *ServerHello* message. Otherwise, by using a different ID, the server requests a full handshake as described above.

When it is possible to reuse the session, both parties restore the previously exchanged session parameters, including the shared secret, and use the secret to generate new symmetric keys. The new keys are different, because they are derived not only from the shared secret, but also from the random data in the current *Hello* messages. The handshake continues with the exchange of the *ChangeCipherSpec* and *Finished* messages, skipping the certificate and the key exchange.

For security reasons, the entries of the server's session cache have a rather short maximum lifetime and they are invalidated immediately in the case of an error.

3.3 Security Analysis of SSL

SSL was designed with the goal of providing security against man-in-the-middle attacks, where the attacker is able to monitor the communication channel and to insert, delete, and alter packets [15]. The attacker may have significant but not unlimited computational resources. The information in this section is based on the analysis of David Wagner and Bruce Schneier [42].

The record layer protocol uses fairly standard and well-analyzed methods to provide for confidentiality and integrity protection. Data transfer is secure as long as the symmetric cypher is secure and the key remains secret. Even the uncovering of the symmetric key leaves the message integrity checking intact, because a different key is used for the keyed message authentication code. This feature is important because US regulations restrict the export of products using strong cryptography and a weak cipher must be used for symmetric encryption. A sequence number is included in the message authentication code which renders replay attacks impossible.

The length of a ciphertext record reveals the length of the plaintext. In the case of HTTPS, traffic analysis can be used to determine the size of the HTTP request and of the response. Sometimes this information can uniquely identify the web page which is being loaded. To foil this attack, SSL should pad the plaintext blocks with a random number of bytes.

The handshake protocol is more critical since most handshake messages are transmitted in plaintext. The *Finished* message, which contains the encrypted authentication code for all handshake messages, allows the two parties to check whether messages have been altered.

The *ChangeCipherSpec* message, however, is not included in this authentication code, and the SSL specification does not mention that the exchange of *ChangeCipherSpec* messages is mandatory. This is a mistake in the specification, and not very carefully designed implementations may be vulnerable when the attacker deletes the *ChangeCipherSpec* messages. Outgoing messages would still be protected, and the attacker would have to recover the symmetric keys in order to decrypt them. The receiver, however, would expect non-authenticated plaintext records.

David Wagner and Bruce Schneier further describe a so-called key-exchange algorithm rollback attack where the attacker fools the communicating parties into using different key-exchange schemes. The client is made to believe that RSA is used while the server is made to believe that Diffie-Hellman is used. When the server sends the *ServerKeyExchange* message containing legitimately signed Diffie-Hellman parameters the client could interpret them as ephemeral RSA parameters when it does not check the size of the parameters. Now the client encrypts the shared secret with wrong RSA parameters which allow the adversary to easily decrypt the secret. This attack, however, can be foiled by a careful implementation.

Concluding it can be stated that the protocol is secure against passive attacks, and, when implemented carefully, offers good protection against active attacks. However, there are shortcomings in the specification of SSL, as warnings of some possible attacks are missing.

3.4 Existing Implementations of SSL

The most wide-spread open-source SSL library is OpenSSL [38]. OpenSSL is built upon an extensive crypto library with ECC support, and ECC cipher suites are included. Because of open patent issues these cipher suites are not enabled by default and a slight modification of the source code is necessary to be able to use them. OpenSSL is not only an SSL library but it also contains functionality to generate, sign and manage certificates.

The versatility of OpenSSL also explains that the size of the library is beyond 1 MB – much too big for embedded systems. A smaller library, developed for embedded systems, is MatrixSSL [28]. MatrixSSL has a library size of less than 50 KB which is still too much for the tight resource constraints of some wireless sensor platforms and originally only RSA is supported. MatrixSSL is under the GNU public license which does not permit the use of the library in closed-source projects unless a commercial copy of the library is purchased.

Chapter 4

Wireless Sensors

A wireless sensor system (Figure 4.1) consists of a microcontroller system with a radio

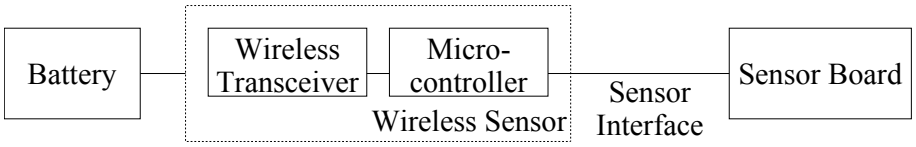


Figure 4.1: Typical Wireless Sensor System

transceiver and an interface to connect actual sensors which are typically available on external sensor boards. These sensor board are application specific and contain various sensors and actuators. The power supply is usually a battery, but other sources, such as solar panels, can be used.

This chapter introduces potential applications of wireless sensors and discusses their requirements. It deals with principles of wireless communication, describes typical hardware and software platforms, and discusses energy conservation and security issues.

4.1 Introduction

At the moment 99 percent of the used sensors are wired. The cost of wired sensor systems is mainly determined by the cost of wiring, whereas for wireless sensors the limiting factor is still the high unit price in the order of $100. This price, however, is expected to drop to $1 or less [44], enabling cost effective wireless sensor networks with great numbers of nodes.

4.1.1 Applications

The potential applications of wireless sensors are not limited to the current applications of wired sensors. The high mobility of wireless sensors and the resulting flexibility of a wireless sensor network enables many applications which would be impossible or infeasible otherwise.

Existing and envisioned applications of wireless sensor networks include industrial control and monitoring, home automation, military sensing, environmental sensing, and health monitoring [6, 44].

Industrial Control and Monitoring In a typical industrial environment there is a relatively small control room, surrounded by a relatively large plant. Sensors are used to collect data about the production process and transmit the data to the control room. In the control room decisions are made based on the sensor readings and wireless actuators can be controlled. Such wireless sensors and actuators can be used in locations where their wired counterparts cannot be used, such as in rotating parts of machines.

Another industrial application is warehouse management and asset tracking. Wireless sensors could be attached to packets and containers and provide live information about their position and condition. In combination with existing RFID tags, wireless sensors could generate a live database of the inventory. They could be equipped with RFID readers and placed all over a warehouse in order to keep track of the inventory at any point of time.

Home Automation An application in home automation is the universal remote control – a PDA-like device which can control and monitor home appliances, lights, curtains, radiators, air conditioning, and door locks. A combination of these services is even more impressive. There could be a good-night button, which turns off the lights, locks the doors, closes the curtains, turns down the heating, and checks if all windows and doors are really shut.

Military Sensing A military application of wireless sensors is the tracking of enemy movements. A vast number of sensors can be dropped in hostile territory to detect and track targets which would otherwise be hidden from radar or satellite-based surveillance.

Wireless sensors could also replace or assist human guards. They could be used to improve mines, such that breaches in mine fields can be detected and mines can be disarmed and located safely when they are no longer needed.

The small size of wireless sensors and infrequent, burst-like communication makes them difficult to be detected. Also in a network with redundant and flexible routing, the loss of a single sensor leaves the whole system intact. These features make wireless network difficult to destroy.

Environmental Sensing In agriculture, wireless sensors can be used to monitor weather and soil conditions to reduce the cost of irrigation and fertilization. In vineyards these data can be used to assess and optimize the quality of the crop. Wireless sensors can be used to track cattle on ranches or to track wild animals in their natural habitats.

Health Monitoring The applications of wireless sensors in health monitoring can be divided in two groups: Monitoring of the performance of athletes and monitoring of patients at their homes.

In the first case, sensors are wearable devices which gather the athlete's pulse and respiration rate and send this information to a personal computer for later analysis. In the second case, sensors can be used to constantly monitor health condition and to notify an emergency response team in the case of unusual values.

4.1.2 Requirements

In many cases the requirements for wireless sensor networks are cost effectiveness and security. However, the exact requirements heavily depend on the application. Cost ef-

fectiveness not only demands inexpensive hardware, but also low power consumption and easy reconfiguration.

Consider the example of a wireless switch which controls the lighting of a room. In this setting, the wireless actuator which control the light bulbs must be connected to the power line anyway and power consumption is a minor issue. The light switch, however, should get by without any wiring and would have a battery. In this case the lifetime of the battery determines the maintenance interval and has great effect on the costs.

An important restriction of wireless sensors is their lack of keyboard and display. As a result, it is difficult to configure them once their program is loaded or when they are deployed. In many applications this is not an issue, because the application is so specific that all sensors need to be reprogrammed for deployment. But consider the example of an off-the-shelf medical monitor which is able to send notifications when a certain parameter is out of range. In this case, it is desireable to have an easy way to configure the sensor and to set the allowed range.

Also the security requirements depend on the application. For example it might not be important to encrypt sensor readings when gathering environmental data, but it is important to make sure that the data have not been altered. In this case, message authentication and integrity protection is sufficient. On the other hand, in a medical application, encryption is important as well to ensure that only authorized persons have access to sensitive information.

4.2 Wireless Networks

4.2.1 Network Topologies

Figure 4.2 shows typical topologies of wireless networks. Nodes in black are devices with

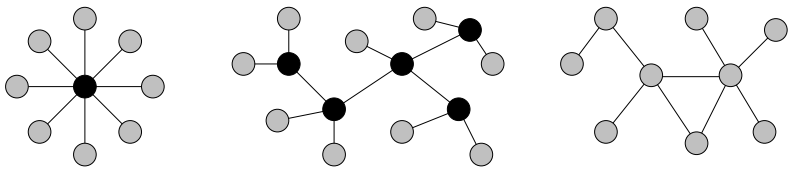


Figure 4.2: Typical Topologies of Wireless Sensor Networks

higher capabilities in terms of available energy and processing power, usually they are connected to power lines and have unlimited energy.

In a star network, each node can only communicate with the central node, the master. This has the disadvantage that all sensors must be within range of the master. The infrastructure network extends the star network by adding more high-capability nodes, which then form the backbone of the network, and each sensor must be in reach of at least one of these nodes. Peer-to-peer networks require no predefined infrastructure and all nodes are equivalent. Routing paths need to be formed dynamically, and the network must be able to deal with node failures.

Star and infrastructure networks are especially suitable when the area of sensor deployment is bounded and electrical infrastructure is available. Applications like building automation and industrial monitoring and control are good candidates. An example for a

star network is Bluetooth, where a personal computer acts as master and the peripherals are the slaves. The cell phone service is an example for an infrastructure network.

In many applications of wireless sensors, however, infrastructure networks are not possible. Consider the example of sensors dropped from an aircraft to monitor a contaminated area. In this example none of the sensors will have a connection to a power line, and the exact locations of the individual sensors, absolute and relative to each other, is not known in advance. In this case, the sensors are on their own to establish network links and routing paths. Routing protocols for these so-called *ad hoc networks* are out of the scope of this text; they are discussed in [44, 26].

4.2.2 Networking Requirements

Conventional wireless communication standards, like IEEE 802.11 (Wireless Local Area Network, WLAN), have the objective to maximize bandwidth and to reduce latency. These objectives are not very important in wireless networks, since there communication occurs sporadically, small amounts of data are transferred, and typical applications have no strict quality-of-service requirements. However, in wireless networks low cost and low energy consumption is of paramount importance.

Reducing cost implies that simpler protocols are used, such that the hardware of the wireless transceiver is less expensive; this simplification may also reduce power consumption. The radio frequency should be in a band which can be used throughout the world without licensing to allow for manufacturing in high quantities.

To reduce energy consumption the duty cycle of the radio needs to be minimized. The duty cycle is the fraction of time the system spends in active mode. Energy saving must be considered in all networking layers, from the physical layer up to the application layer. A considerable amount of work has been spent in the development of medium access control protocols which try to coordinate access to shared medium radio in a way such that nodes can be in low-power modes most of the time.

4.2.3 Medium Access Control Protocols

An abundance of medium access control protocols have been proposed for wireless sensor networks. This section introduces three of the most important ones.

The S-MAC Protocol

The S-MAC protocol [43] tries to reduce energy consumption by avoiding idle listening, collisions and overhearing. The basic principle is that nodes are powered down most of the time and only wake up for short periods to communicate with their neighbors. Thus nodes must be synchronized and keep schedules telling them when the neighbors are awake. The way these schedules are generated on initialization ensures that many nodes share a single schedule and only a few nodes have to keep track of multiple schedules. Because of clock shift, schedules must be resynchronized from time to time.

The S-MAC protocol uses a combination of two methods to avoid collisions. First each node has to listen to the radio to determine whether another node is currently transmitting. If not, the sender and the receiver exchange RTS (Ready To Send) and CTS (Clear To Send) packets including the length of the data to be transferred. Nodes which overhear an RTS or CTS packet can go to sleep because they know that other nodes are going to

communicate and for how long the transmission is going to last. This mechanism also avoids overhearing of data packets with other destinations.

By adjusting the sleeping duration one can make a tradeoff between low power consuption and latency. The S-MAC protocol has a significant energy advantage over IEEE 802.11, and the advantage increases as the message interarrival time and the sleeping period increases.

The B-MAC Protocol

The B-MAC protocol [29] achieves energy savings by taking advantage of a special hardware feature some wireless transceivers offer. The transceiver must be able to detect an ongoing transmission by just tuning in for a very short time, much shorter than the time to transmit a full packet.

For example the Chipcon CC1000 transceiver exposes a method to query the signal level on the channel. The B-MAC implementation periodically samples the signal level and compares it with the level of the noise floor. Before sending, a node uses this method to check whether the channel is free.

The same method is used for low power listening. A node periodically wakes up, samples the channel and goes back to sleep when no signal is present, or enters receive mode otherwise. Each transmission consists of a preamble and the payload. Usually, a preamble is very short and just allows receivers to synchronize with the sender. In the B-MAC protocol, however, receivers sample the channel in a certain interval, and the preamble must be long enough to be detected and to allow for synchronization. Thus the preamble must be longer than the sampling period. Figure 4.3 illustrates how the low power listening scheme works.

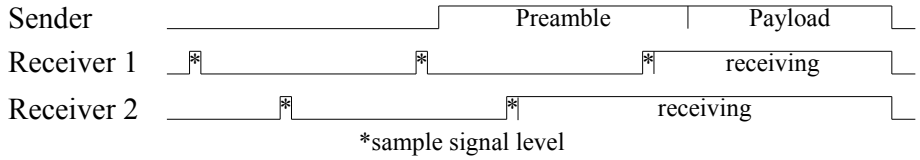


Figure 4.3: Low-Power Listening of the B-MAC Protocol

In the B-MAC protocol one can make adjustments to the sampling interval and the preamble lengths to trade energy for latency. Very long sampling intervals, however, also require the sender of a message to send a very long preamble, wasting energy. Also, too long preambles cause congestion in situations of higher network traffic.

The IEEE 802.15.4 Standard

The IEEE 802.15.4 standard [24] not only specifies the medium access control layer, but also the physical layer for Low-Rate Wireless Personal Area Networks (LR-WPAN).

The IEEE 802.15.4 working group describes an LR-WPAN as “a simple, low-cost communication network that allows wireless connectivity in applications with limited power and relaxed throughput requirements. The main objectives of an LR-WPAN are ease of installation, reliable data transfer, short-range operation, extremely low cost, and a reasonable battery life, while maintaining a simple and flexible protocol.” [24]

The standard is very flexible, as its goal is to support a wide range of applications. The basic channel access scheme is carrier sense multiple access with collision avoidance. Optionally, beacons can be used and the interval between two beacons is divided into a contention period, where every node is allowed to initiate a transmission, and guaranteed time slots to reserve channel time for individual nodes.

There are two classes of devices: *Full function devices* can take the role of a network coordinator, and *Reduced function devices* are very low cost devices, only able to communicate with full function devices, without the need to manage routing information.

The IEEE 802.15.4 standard supports multiple network topologies. Star networks and peer-to-peer networks are discussed in the standard. Star networks consist of one full function device, acting as the network coordinator and multiple reduced function devices. In peer-to-peer networks only full function devices are used, since in this topology each node must have the capability to communicate with each other node.

IEEE 802.15.4 specifies two physical layers working in two distinct frequency bands. The lower band operates in the frequency ranges 868.0–868.6 MHz (for Europe) and 902–928 MHz (for the Americas). The upper band operates in the range 2.400–2.485 GHz and can be used worldwide. The maximum raw date rate is 20 KB/s in the lower band and 250 KB/s in the upper band.

To promote the IEEE 802.15.4 standard, the ZigBee alliance has been founded, trying to assure interoperability between various devices. The ZigBee alliance works on specifications for higher level protocols based on IEEE 802.15.4.

4.3 Wireless Sensor Platforms

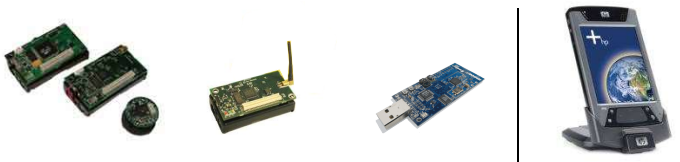
There are different options regarding the choice of wireless sensor platforms. One extreme is the use of regular personal digital assistants (PDA) equipped with some sensors and actuators. The other extreme is the development of specialized single-chip solutions.

Embedded PCs and PDAs can be used where the application permits their size and energy requirements. The main advantage is that there are several operating systems, programming languages, and development tools to choose from. Also, the use of wireless LAN makes such a system interoperable with the existing network infrastructure.

Another option is the use of dedicated wireless sensor boards, made from off-the-shelf integrated circuits, like microcontrollers and wireless transceivers. For an off-the-shelf microcontroller there exist development tools, but not as many as for PDAs and computers. Also, the processing power and the networking bandwith is much more limited. However, dedicated wireless sensor boards come with the great advantages of a small form factor, low energy requirements, and lower cost. They typically have the size a two AA batteries and they are able to operate for months or years from batteries.

The goal of other projects is to create fully integrated system-on-chip solutions. This approach allows for high optimization of the hardware, yielding systems with extremely low power consumption and very small footprint. These advantages come at the price of a costly development process and the lack of tools for software development.

A popular family of wireless sensor platforms, developed by the University of California, Berkeley, is the family of the so-called “motes.” The motes are commercially available and they are supported by TinyOS, an open source operating system. Figure 4.4 shows the characteristics of the most recent members of the mote family, compared with a typical PDA. The Mica family and the Telos motes are introduced in the following sections.



	Mica2	MicaZ	Telos	IPAQ PDA
Processor	ATmega128L	ATmega128L	TI MSP430	Intel PXA270
Data width	8-bit	8-bit	16-bit	32-bit
Speed	4-8MHz	8MHz	8MHz	624MHz
Radio	CC1000	802.15.4	802.15.4	802.11b (WLAN)
Data rate	20 kbps	250 kbps	250 kbps	2-11 Mbps
Range	10-100m	10-100m	10-100m	100m
Memory				
RAM	4 KB	4 KB	10 KB	64 MB
Flash	128 KB	128 KB	48 KB	128 MB
Price	~$100	~$100	~$100	~$500
Battery life	Mo/Yrs	Mo/Yrs	Mo/Yrs	Hrs

Figure 4.4: Comparison of Wireless Sensor Platforms With a Typical PDA [9, 10]

4.3.1 The Mica Family

Figure 4.5 shows the block diagram of the Mica2 mote. The main component is the

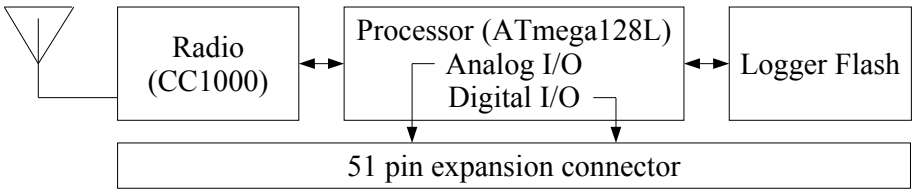


Figure 4.5: Block Diagram of the Mica2 Mote [9]

ATmega128L microcontroller from Atmel [2]. It contains an 8-bit general-purpose CPU with integrated multiplier, 128 KB flash memory for program storage, and 4 KB RAM. The program memory can be protected by a password to prevent others from reading the the code and sensitive information, like cryptographic keys. Additionally, an external serial flash memory with 512 KB is available to store data for later batch processing or transmission.

The microcontroller further contains various analog and digital I/O channels, including an SPI interface, an eight-channel analog to digital converter, and pulse width modulated outputs. Most of these channels are made available through a 51-pin expansion connector, which can be used to attach a variety of sensor and actuator boards.

The microcontroller is connected to a ChipCon CC1000 radio transceiver, which operates at the frequencies of 868 or 916 MHz. The transceiver implements a proprietary physical layer, which achieves a maximum data rate of 38.4 KB/s.

Besides the Mica2, the Mica family contains the Mica2dot and the MicaZ mote. Mica2dot is a version of Mica2 which is substantially smaller. It is shaped like a disk with a diameter of 25mm and operates from a 3V coin cell. Mica2dot has a reduced clock speed of 4 MHz and has an expansion connector with less I/O channels. The MicaZ mote is similar to the Mica2 mote but it uses a different radio transceiver which implements the physical layer defined by IEEE 802.15.4. The advantages are higher bandwidth, higher

reliability, and standards compliance.

To establish a connection to a personal computer, an interface board is required. The interface board can be connected to the PC with a serial cable and the mote can be plugged into the board. This way, application programs can be downloaded to the mote and communication with the running application is possible. A mote with an interface board can be a base station for the wireless network, relaying messages between the PC and the wireless network.

4.3.2 Telos

The Telos platform is an improvement of the Mica family, which was designed with the goal of reducing the power consuption and reducing the transition time between sleep mode and active mode [30]. The main change is the use of a different microcontroller. Instead of the ATmega128L, Telos motes contain the MSP430 microcontroller from Texas Instruments [37]. The MSP430 has a 16-bit RISC (reduced instruction set computer) architecture, integrated with 48 KB program memory and 10 KB RAM. Although the instruction set does not accommodate a multiply instruction, a multiply accumulate unit is integrated and can be addressed like other peripherals.

The MSP430 microcontroller operates with voltages down to 1.8V, which is a significant improvement over the 2.7V required by the ATmega128L. The cut-off voltage of the whole Telos system is 2.1V, defined by the radio transceiver. The lower cut-off voltage has the advantage that a greater fraction of the energy stored in the batteries can be used.

Telos, like MicaZ, uses an IEEE 802.15.4 capable radio transceiver. The antenna is laid out as a wire on the circuit board, making the system physically more robust and less expensive. One version of the Telos circuit board already contains on-board sensors for light intensity, humidity, and temperature, and more sensors can be added to two small connectors.

Where a Mica mote requires an interface board for a connection with a PC, a Telos mote has a built-in USB interface. The USB interface, when connected to a host, is also a power supply and no batteries are required for motes with a permanent connection with a PC, as it is the case for base stations.

4.4 Software Design for Wireless Sensor Networks

Software design for wireless sensor networks is significantly different from traditional programming, where one can rely on operating systems to provide a high level of hardware abstraction. Due to the constrained resources of typical wireless sensor platforms and the highly specialized applications, programmers of wireless sensors have to deal with hardware characteristics and low-level optimizations.

There are two approaches in software design for wireless sensors: The *node-centric* approach focuses on the behavior of a single node in the context of the network, whereas the *state-centric* approach focuses on the state of the physical phenomenon which is being observed by distributed devices [44].

The node-centric approach is more similar to traditional software development. Operating systems or libraries provide a certain level of hardware and networking abstraction, and the programmer specifies the behavior of a node, including the interaction with the network. In contrast to traditional operating systems, operating systems for wireless sensors are highly configurable, such that unnecessary modules can be omitted. Also, the

boundary between the operating system and the application is not as clear and both part may even be compiled together into a single binary.

State-centric programming models allow the programmer to think in higher abstraction levels, looking at the whole system, or relevant parts of it, without having to focus on single sensor nodes. Such systems provide methodologies and frameworks to give abstractions for issues related to concurrency, networking, and resource management. Nodes are logically organized in groups with certain properties, such as geographical vicinity, or the ability to sense the same physical phenomenon. Nodes have assigned roles, for example the role of a group leader, collecting data from its group, or that of a follower. In message passing, the addresses of nodes are abstracted to their properties, like group membership and roles.

The software platform described in the remainder of this section follows the node-centric approach. TinyOS, a small open-source operating system for wireless sensors, and nesC, a component-based programming language are introduced.

4.4.1 The TinyOS Operating System

TinyOS [39] is an open-source operating system, especially developed for wireless sensors. TinyOS consists of a small main program, responsible for booting and task scheduling, and a large collection of modules, implemented in nesC, to provide abstractions for various hardware components.

The module library consists of a hierarchy of modules. The lowest level is called *hardware presentation layer*, and only abstracts the hardware without introducing any state. Higher-level modules might implement rather complex protocols in a hardware independent manner. Figure 4.6 illustrates this concept with the example of a networking stack for Mica2 motes. In this example the top-level component implements the TinyOS

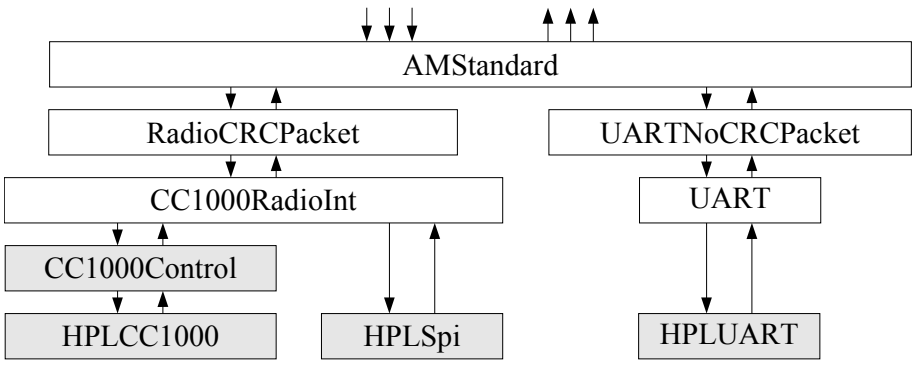


Figure 4.6: TinyOS Networking Stack for Mica2. Grey components are hardware abstractions.

"Active Message" model and can be used to send and receive messages via the radio transceiver and the serial interface in a transparent way.

A TinyOS application is a collection of further modules which interact with the modules of the operating system. The top-level entity of an application is a configuration, defining how the application modules interact with each other and with the modules provided by the operating system. This way, only the necessary parts of TinyOS are compiled into an application, keeping program size and memory requirements as low as possible.

Code in TinyOS can be executed in the context of an event or in the context of a task. Events originate from hardware interrupts and can preempt other events or tasks,

whereas tasks are non-preemptive and run sequentially. For a responsive system, however, one must try to minimize the execution time of a task. To conserve energy, the operating system configures the processor to sleep, whenever the task queue is empty and there are no pending events.

Each application has to include the module `Main`, which contains the bootstrapping code. The `Main` module also fires the first events, which are propagated to all modules of the application to trigger the initialization process and to start the application.

4.4.2 The nesC Programming Language

NesC [40] is an extension of the C programming language, designed to facilitate the implementation of the component-based architecture of TinyOS. Its structure somewhat resembles that of hardware description languages like VHDL. The structural elements of a nesC program are *interfaces*, *modules*, and *configurations*. The latter two contain the implementation of interfaces.

Interfaces

Interfaces define the way, how different modules can communicate with each other. Communication between modules is implemented through method calls, and interfaces specify the syntax of these methods. According to the direction of the method call, nesC distinguished between *commands* and *events*. Modules in a higher layer issue commands to modules in a lower layer, and the communication in the other direction is done through events.

An example of an interface for a simple module which can send messages in some way, would consist of one command and one event. A command would be defined to send the message and an event to notify the sender about the result of this operation. In this example, however, the interface does not specify in which way and over which link the messages will be sent.

Implementations

In nesC there exist two types of implementations: *modules* and *configurations*. Both types can provide and use interfaces, where in nesC interfaces are provided to higher level components, and used from lower level components.

A module contains C-like code, while a configuration contains further modules and configurations. A configuration connects the interfaces of its sub-components and can also expose one or more of these interfaces. By enabling this kind of hierarchy, nesC reflects the design of the TinyOS operating system.

4.5 Energy Management

Besides the tight constraints in computational power, available memory, and networking bandwith, the energy available to a sensor node is usually limited. In many applications it is impractical, or even impossible to change batteries on a regular basis. This might be, because the sensors are physically not accessible after deployment, for example when they are dropped by an aircraft in a hostile environment.

The specific application defines the minimum lifetime of sensor nodes. Sensors mounted on a machine should survive at least the regular maintenance interval, or sensors deployed

on a farm should work throughout at least one growing season. In applications like habitat monitoring, however, the lifetime of the sensors should ideally be infinite.

This section examines available energy sources and investigates where and how sensor node consume energy. Finally, strategies to reduce the energy consumption and to maximize the lifetime of wireless sensors are discussed.

4.5.1 Energy Sources

Most sensor nodes today operate from batteries, whose capacity is the ultimate energy reservoir. As an alternative to batteries, energy scavenging tries to exploit various renewable energy sources available in the environment.

Batteries

Of the different types of batteries, alkaline batteries are the least expensive option and easily available. Rechargeable batteries can be an option for the use in a laboratory, but they have the disadvantage of a high self discharge rate, limiting the lifetime to several months.

Alkaline batteries have a nominal voltage of 1.5V and capacities in the order of 2000mAh. The nominal capacity of a battery is the energy, the battery can deliver under the assumption of a static current draw, until the voltage drops under a certain cut-off voltage. Duracell, for example, specifies the cut-off voltage to be 0.8V [16], which is just more than half the nominal voltage. The minimum operating voltage of the hardware defines the actual cut-off voltage and can be significantly higher. Thus, in real systems, often not the full specified capacity can be used.

The actual capacity of alkaline batteries not only depends on the cut-off voltage, but also on the discharge characteristics. Lower current draw generally yields a higher capacity. Further, the capacity is higher when the current draw is not uniform, but occurs in pulses. Pulsed current draw allows the batteries to recover in the period between two pulses [44]. This property is especially interesting for wireless sensors, which are in a low-power mode most of the time and only occasionally wake up and consume energy.

Energy Scavenging

Energy scavenging [27] converts energy which is available in various forms in the environment into electrical energy. Depending on the location of deployment, different energy sources can be used. These sources include ambient radiation, vibration, and temperature coefficients.

Radiation exists in the forms of radio-frequency radiation and light. Radio-frequency sources only provide very low power unless they are placed next to a powerful transmitter. However, radiation can be directed to the device in order to provide its power. This principle works well with RFID (radio-frequency identification) tags, but typically only provides power between 1 and 100μW. The use of solar panels is already very common and can produce up to 100mW/cm^2 in bright sunlight.

In industrial environments, the vibration of nearby machines can be exploited. This idea works well to power sensors in the wheels of a car, for example. The yield heavily depends on the intensity and on the frequency of the vibration and can be in the order of 800μW near machines. Today's thermoelectric generators have a very low efficiency

for realistic temperature gradients. With $\Delta T = 5°C$, the performance is in the order of $60\mu W/cm^2$.

An example of a system, which employs energy scavenging for wireless sensors is called "Prometheus" [25], developed at the University of California, Berkeley. Prometheus uses solar panels as the energy source and an intelligent system to store energy for dark periods. The developers claim that with only 5 hours of sunlight per month the system can power a Telos mote for 43 years when the duty-cycle of the application is at most 1%.

The Prometheus system has two energy buffers. The primary buffer needs to handle high levels of energy throughput and frequent charge cycles. In contrast to batteries, capacitors survive a nearly infinite number of charge cycles and have no memory effect. The capacitors must have enough capacitance to minimize the usage of the secondary buffer, which is a rechargeable battery. The lifetime of rechargeable batteries is determined by a limited number of charge cycles. To prolong the lifetime of the battery, an intelligent algorithm is used to control the energy transfers between the source and the two buffers.

4.5.2 Energy Consumers

The main energy consumers in a wireless sensor system are the CPU and the radio transceiver. Secondary consumers include sensors, analog-to-digital converters, and external memories. Table 4.1 shows the current draw of Mote platforms in different operating modes.

Table 4.1: Measured Current Draw of the Motes [30]

Operation	Telos	Mica2	MicaZ
Mote Standby	5.1 μA	19.0 μA	27.0 μA
Microcontroller Idle (Oscillator on)	54.5 μA	3.2 mA	3.2 mA
Microcontroller Active	1.8 mA	8.0 mA	8.0 mA
Microcontroller + Radio Receive	21.8 mA	15.1 mA	23.3 mA
Microcontroller + Radio Transmit	19.5 mA	25.4 mA	21.0 mA
Microcontroller + Flash Read	4.1 mA	9.4 mA	9.4 mA
Microcontroller + Flash Write	15.1 mA	21.6 mA	21.6 mA

On the compared mote platforms, the radio contributes most to the overall energy consumption. In the case of the Telos and the MicaZ mote, the radio has a lower consumption in transmit mode than in receive mode. Because the used radio transceiver (ChipCon CC2420) offers no low-power listening mode, it must be in receive mode in order to detect network traffic; this is called *idle listening*.

4.5.3 Strategies to Maximize the Lifetime of Sensor Nodes

The goal of maximizing the lifetime of sensor nodes must be considered in the design of the hardware and software. In the hardware design, not only the low power consumption in standby and in active mode is important, but also low operating voltage to get the maximum capacity from the batteries. A fast transition between standby and active mode reduces the energy that is wasted while the system wakes up. To accelerate this transition, crystals with a low series resistance are used for the oscillators [30].

Power saving in software is accomplished by turning off unused components and configuring used components to sleep whenever possible. This concept can be easily applied to microcontrollers. The radio, however, cannot be simply turned off, because it should still be possible to receive messages from neighboring nodes, and not all transceivers support a low-power listening mode. Various protocols have been proposed to minimize the time the radio transceiver spends on idle listening. These protocols usually require neighboring nodes to be synchronized, such that the nodes wake up at the same time and communicate during a small time slot.

In a wireless sensor application, not the lifetime of a single node is important, but the lifetime of the whole system. Boulis and Srivastava [5], for example, propose a routing protocol which tries to avoid routing via nodes with low energy reserves by incorporating the energy levels into the routing metrics.

Looking at the development of microcontrollers and radios over time, the trend is that the energy consumption of microcontrollers decreases, while the energy consumption of radio transceivers tends to stay constant. As a consequence, it will become more and more favorable to perform many computations locally in the network to reduce the need for communication.

4.6 Security

In order to design a secure wireless sensor network, it is necessary to first analyze the capabilities of potential adversaries. Because of resource limitations, a general-purpose security implementation, which is able to deal with all kinds of threads, is not possible. This section gives an overview over threats to wireless sensor networks and deals with key distribution schemes.

4.6.1 Threats to a Wireless Sensor Network

In their work on security for wireless sensor networks, Avancha *et al.* identify the following threats [3].

Node Outage

Node outage is a threat to every wireless sensor network, even in a scenario where no adversary is present. Nodes may get physically damaged or run out of energy.

The system must be flexible enough that other nodes can take over the job of a dead node. This requires a flexible routing protocol in a multi-hop network.

Node Malfunction

Nodes may stop working correctly and gather wrong sensor readings or corrupt forwarded data. This can happen as the consequence of an attack or by physical damage.

Malfunctioning nodes should be detected and excluded from the network.

Passive Information Gathering

In this attack the adversary only listens to the radio traffic and collects the transmitted data.

As a defense, some sort of symmetric cryptography is required to protect the transmission. One predistributed key for the whole system is sufficient, under the assumption that the attacker is not able to capture a node and extract the key.

Traffic Analysis

Traffic analysis is another passive attack in which patterns of communication are monitored. Even when the data are transmitted in encrypted form, traffic analysis can lead to conclusions about the function of the network and the current state of alert.

The usual method of counteracting traffic analysis is to send dummy messages with a constant frequency. When a real transmission is required, it replaces one of the dummy messages and no change in the communication pattern can be detected. This approach, however, is impractical in wireless sensor networks, because it leads to high radio traffic, draining the batteries even faster.

Subversion of a Node

When a node is captured the adversary may be able to extract the secret keying material. Moreover, the node may leak information about the structure of the wireless sensor network and the employed algorithms. Captured nodes can also be reprogrammed and used to run denial of service attacks.

To prevent subversion, nodes must be designed in a tamper-proof way. Packages are available, which notify the sensor when they are tampered with, to initiate the deletion of sensitive information. Nodes which fail due to energy exhaustion should also automatically erase sensitive information.

Denial of Service

There are various denial of service attacks against wireless sensor networks. A simple attack is to jam the radio link, whereas more sophisticated attacks work against the routing protocol and require the subversion of nodes or the insertion of false nodes. These false nodes may drop messages instead of forwarding them ("black hole"), corrupt messages, or insert new messages to drain the resources of other nodes ("resource exhaustion"). An interesting instance of a resource exhaustion attack in a system using public-key certificates is the insertion of wrong certificates. This way, nodes can be forced to unnecessarily perform expensive signature verifications.

To counteract radio jamming, spread-spectrum modulation can be used, or one can try to route around affected areas. Schemes using timers and counters can be used to detect black holes, when the absence of expected data is tracked. Message authentication to detect corrupt messages also allows nodes to detect introduced messages, which can limit the effect of a resource exhaustion attack.

4.6.2 Key Establishment in Wireless Sensor Networks

To protect confidentiality and authenticity of messages, the communication between pairs of nodes needs to be secured by encryption and a message authentication mechanism. As a prerequisite, neighboring nodes must share a common key.

The problem in wireless sensor networks is that the configuration after deployment is not necessarily known. Thus the key establishment scheme must be able to establish keys

for each arbitrary pair of nodes. However, there is the danger that an adversary captures a node and extracts the secret keying material and the network should be able to revoke compromised keys. Also, capturing a node should not compromise the security of the communication between the other nodes.

The following approaches can be considered for key establishment [8]:

- **Network-Wide Key.** The simplest approach is to use a single network-wide key. When a single node is captured, the security of the complete network is compromised. A possible scenario is that the common key is only used in the beginning for neighboring nodes to exchange pairwise keys. The network-wide key would be deleted after this step, making it impossible to add further nodes at a later time.

- **Pairwise Keys.** One could provide each node with a unique key for every other node in the network. When one node is captured, is cannot reveal any information to compromise the rest of the network. However, this approach is not scalable since the number of keys each node must store is equivalent to the number of nodes in the network.

- **Public-Key Cryptography.** Before deployment, each node gets the public key of the base station, an own public and private key, and the base station's signature of the own public key. In the initialization phase after deployment, nodes exchange their public keys and use a key-exchange protocol to generate symmetric keys. Each node can verify the integrity of a received public key by checking the base station's signature.

  A compromised node does not affect the security of other communication links, and the scheme is fully scalable. However, this system is vulnerable to energy exhaustion attacks. Messages with false signatures may be introduced and cause the recipients to perform unnecessary signature verifications, which is relatively costly.

- **Random Key Predistribution.** In the random key predistribution scheme, a pool of keys is generated prior to deployment and each node is programmed with a random set of keys from the pool. Each of the keys has an identifier used by the nodes to find neighbors with common keys. Two nodes can create the same symmetric key by combining the predistributed keys they share.

  Not all links can be established this way because a node might not share a key with each neighbor, and missing links can be established by exchanging keys via multi-hop routes which already exist. However, there is a possibility that parts of the network are not connected to other parts at all. Programming one node with a greater set of keys decreases the likelihood of this to happen, but increases the damage an adversary can do by just compromising a single node.

  The random key predistribution scheme has the disadvantage, that by compromising a big enough number of nodes, also the keys of links of other nodes can be recovered. However, the number of nodes the adversary needs to capture is rather big, making the attack expensive and detectable.

Probably the main reason why public-key cryptography is not used in wireless sensor networks is that it is believed to be too expensive in terms of computational power and memory requirements. Because of this belief there is a lot of research concerning exotic key-exchange protocols for wireless sensor networks. Recently Gura *et al.* created an

efficient implementation of elliptic curve cryptography for the Berkeley Motes and showed that public-key cryptography is in fact feasible for wireless sensors [21].

Chapter 5

Sizzle – Security Architecture for Wireless Sensors

Sizzle[1] started out as a very small implementation of SSL for embedded systems, and was developed by the Next Generation Cryptography Group at the Sun Microsystems Laboratories. Sizzle has been ported to the Mica platform of wireless sensors and extended with a webserver with support for the HTTPS protocol. Soon a complete architecture for secure interaction with wireless sensors evolved around the SSL implementation.

This chapter first motivates the use of secure webservers in wireless sensors, and then describes the architecture of Sizzle and the environment in which it is used. The last part presents the communication protocol and elaborates on the implementation of the base station.

The parts of this chapter which are not result of my work are marked with the reference to the conference paper which first introduced Sizzle: "Sizzle: A Standards-Based End-to-End Security Architecture for the Embedded Internet" by Vipul Gupta and others [19].

5.1 Introduction

The Next Generation Cryptography Group is strongly involved in promoting ECC as a replacement for legacy cryptosystems like RSA. This is done by enabling ECC in a wide range of applications, from high-performance servers to very constrained embedded systems.

ECC code has been contributed to the open-source libraries OpenSSL and Netscape Security Services, which are used by the Apache web server, the Mozilla web browser, and by many more applications. The support of ECC has been included in Java-based crypto libraries and hardware accelerators will be included in Sun's future processors.

The goals of the Sizzle project are

- to demonstrate that public-key cryptography is feasible on wireless sensors.
- to show that it is possible to implement a standard protocol on wireless sensors.
- to explore new security architectures for wireless sensor networks.
- to demonstrate the performance advantage of ECC in comparison with RSA.

[1] The name *Sizzle* is derived from SSSL which stands for Slim SSL.

- to create interest in ECC.

5.1.1 Extend of My Work

Sizzle has been developed by Vipul Gupta of Sun Labs and by several student interns. The version of Sizzle my work is based on already supported SSL connections with ECC and RSA key exchange, but only for one platform, the Mica2 motes. The system, especially the communication protocol, was not very reliable and most modules still had significant room for optimizations.

My job was to optimize Sizzle to improve its performance and to reduce its resource utilization. I ported Sizzle to the MicaZ platform and finally to Telos, which uses a different microcontroller with a different instruction set. This port involved a complete reimplementation of the public-key operations for ECC and RSA, and some platform-specific optimizations throughout the rest of the code.

I redesigned the original communication protocol to make it more reliable and extended it by an energy saving mode. I reimplemented the networking stack on the device and on the gateway and added a convenient way to interact with the wireless sensor network. I did an extensive analysis of the resulting system in terms of resource utilization, speed, and energy consumption.

The remainder of the thesis presents the resulting system with focus on the parts done by myself. Parts which describe prior work are marked accordingly.

5.1.2 Applications

An implementation of SSL is only suitable for a subset of applications wireless sensors make possible. SSL can be useful in sensor networks with star or infrastructure topology in a controlled environment. As a key-exchange protocol in ad-hoc networks it is probably too heavy-weight, since SSL imposes a significant communication overhead.

An embedded secure web server is useful in applications which require the possibility to configure individual sensors. Sensors might be designed and programmed to cover a certain scope of duties, but the exact application and the interaction with other sensors might not be clear at the time of programming. Application domains where this situation is thinkable are home automation, process control, and medical monitoring.

A company which produces medical equipment, for example, could sell wearable sensors to monitor a patient's heart rate. These sensors would be preprogrammed with the functionality send notifications to a doctor when the heart rate is not in a specified range. The doctor, however, could use a web interface to specify this range and to configure how the notification is sent. Additionally, the web interface allows the doctor to read the current value and to view statistical information, like the minimum, maximum, and average value.

An example for an application in home automation is that of a wireless thermostat. There could be simple temperature sensor with the only functionality to send messages like "too hot" or "too cool" to a different node. The second node contains an actuator able to control the heating, ventilating, and air conditioning of a room. In this example, a web-based interface in the temperature sensor can be used (1) to tell the temperature sensor which actuator or actuators to control, (2) to set symmetric keys for communication with the actuators, (3) to configure the desired temperature range, and (4) to query the current temperature. A web-based interface in the actuator can be used to set a symmetric key for communication with the sensor, and to monitor or change the current setting.

The connection of wireless sensors to the Internet has the advantage that tasks, like the ones mentioned above, can be performed from anywhere in the world. The implementation of HTTP/SSL on a sensor ensures end-to-end security from the web browser all the way to the embedded device.

Applications which do not need a web interface can still build upon SSL with proprietary protocols. Such applications benefit from the fact that the security of SSL has been thoroughly analyzed and that various software libraries already exist. In a factory, for example, a monitoring and control station can use SSL to securely communicate with the devices being monitored or controlled, and use a protocol which is more tailored for the application than HTTP.

5.1.3 Requirements

Based on the intended applications, there are several requirement for a secure embedded web server to ensure security, usability, and extendability.
The following requirements for Sizzle and its environment have been identified:

- **Security.** Communication with the wireless sensor should be confidential and the integrity-protected. Additionally, some sort of access control is necessary, allowing to configure different sets of capabilities for different people. Instead of only link-level security, end-to-end security between the client and the wireless sensor should be implemented.

- **Small Size.** The implementation on the sensor must be small. On Mica platforms only 4 KB of RAM is available, and on the Telos platform, the size of the program memory is only 48 KB. Ideally, the secure web server should only use a fraction of this memory, such that there remains enough space for the actual application.

- **Standards Compliance.** The system should be fully compliant with the SSL and HTTP specification, such that no modifications in the web browsers is necessary.

- **High Speed.** The implementation should be efficient enough to create a good user experience. The measure is the response time of the web server which should be below a certain limit. The limit, however, is very subjective, and the user experience gradually degrades with increasing response time.

- **Energy Efficiency.** Energy saving mechanisms should be implemented to maximize the lifetime of the batteries.

- **Portability.** The same code-base should be usable for various sensor platforms, and platform-dependent code should be separate from the code-base. The communication protocol should be independent of the underlying hardware and use a sufficiently high TinyOS abstraction layer.

- **Reliability.** The system, especially the software on the sensors, should be reliable, such that manual resets of sensors and the base station are not necessary. The implementation should be robust and able to recover from errors.

- **Convenient User Interface.** The interface to the wireless sensors should be simple and convenient. Sensors should be discovered by the base station, and the base station should implement a method to access them. The user should be able to monitor the states (online/offline) of individual sensors.

- **Sample Applications.** Sample applications should be implemented to demonstrate the capabilities of the system. There should be one application to show how to control something, and one application to show how sensor readings can be made accessible.

In a later chapter the resulting system will be evaluated with respect to these requirements.

5.1.4 Design Decisions for Sizzle

SSL is a very versatile protocol and it is not possible to support all of its features on platforms with such tight resource constraints like wireless sensors. However, it is possible to choose a certain small subset of features and apply some tweaks while still remaining fully compatible with the specification. Also advanced features of HTTP can be exploited and the web browser on the side of the client can be tweaked for the interaction with embedded devices. The following list enumerates several design decision made with the goal of reducing the memory footprint, the bandwith, and the computational expense of Sizzle (partially from [19]):

- Sizzle only implements a small set of cryptographic algorithms. The SHA1 and MD5 hashing algorithms are required by SSL, no matter which cipher suite is used. The algorithm of choice for symmetric encryption is RC4, which is significantly faster than other stream ciphers and less complex then block ciphers. ECC and RSA can be used for the key-exchange process, where the implementation of RSA is necessary for compatibility with web browsers not supporting ECC, and to allow for comparisons of speed and energy consumption. The support for ECC and RSA can be individually disabled by compile-time flags to further save program memory. This set of cryptographic primitives enables three SSL cipher suites: RSA-with-RC4-128-MD5, RSA-with-RC4-128-SHA, and ECDH-ECDSA-with-RC4-128-SHA.

- The public key of the server, which is used in the key exchange, is part of the certificate, and thus it is not necessary to generate an ephemeral key pair and to send an explicit *ServerKeyExchange* message. The certificates only contain the absolutely necessary fields. The resulting ECC certificate has 222 bytes and the RSA certificate has 413 bytes.

- The public key in the certificate is associated with particular elliptic curve domain parameters, and the key exchange operation operates on this very elliptic curve. To reduce the complexity of the ECC implementation and to increase the performance, the curve *secp160r1* [7], a 160-bit curve over a prime field, has been chosen. While ECC with 160 bits is considered enough for today's security needs, this curve has the advantage that it is defined over a prime field which allows for a very efficient implementation of reductions.

- Sizzle implements session reuse to alleviate the overhead of full SSL handshakes. Because of the memory limitations, only a small number of sessions can be cached on the server, and relatively small session identifiers can be used. While other SSL implementations use 32-byte IDs, Sizzle only uses 4 bytes to reduce the memory footprint and the bandwidth.

- Client authentication is not supported at all, such that no client certificate needs to be transmitted, and most notably, no certificate parsing and verification code is required on the server. Clients, however, can be prompted for username and password by the web server, as it is usual in today's e-commerce and Internet banking applications. The waiving of client authentication in the SSL handshake has no influence on the compatibility with other SSL implementation because the server has to explicitly request a client certificate by sending a *CertificateRequest* message. Sizzle never does.

- Sizzle supports persistent HTTP – a feature of HTTP/1.1 [14] where the connection between client and server remains open after the first request and can be reused for subsequent requests. Thus an SSL handshake, full or with session reuse, is only required for the first request.

- Sizzle supports only one connection at a time. The state of the SSL protocol and the buffer for incoming and outgoing records requires so much memory that multiple instances are not possible. When there is a connection attempt from a client and a currently a connection with another client is open but idle, the existing connection is terminated gracefully and the new connection is established. This feature is important because with persistent HTTP connections can remain open, yet unused, for a long time.

- Another way to reduce the bandwidth involves a modification of the web browser such that it sends shorter HTTP requests. A regular HTTP request has a size of more than 400 bytes, but most of it is used to inform servers about the capabilities of the web browser and the user's preferences, like the desired language. The majority of the headers, which are transmitted by default, can not be used by a small embedded web server. I implemented the functionality to send minimal requests as a configurable option in the Mozilla browser [20]. When enabled, the browser omits headers like *Accept*, *Accept-Language*, *Accept-Encoding*, *etc.*, and the resulting request has a size of roughly 100 bytes.

5.2 Gateway Architecture

This section explains the architecture which makes it possible to connect wireless sensors to the Internet.

To allow for communication between a wireless sensor running Sizzle and a client running a regular web browser, a third entity – a gateway – is involved. The gateway is a personal computer with a wireless base station and is responsible for connecting the wireless sensor network to the Internet. The gateway maps each available sensor to a distinct IP-port and a client can communicate with a sensor by opening a TCP/IP connection with the gateway on the specific port. The gateway then relays messages between the TCP/IP connection and the radio link in a way that is transparent for the client.

Figure 5.1 illustrates this gateway-based architecture and shows how communication between a client and a sensor takes place. The introduction of a gateway between the client and the embedded devices provides several benefits [19]:

- The gateway serves as a bridge between the embedded devices and the rest of the Internet. It connects to the Internet using a high-speed link and communicates with

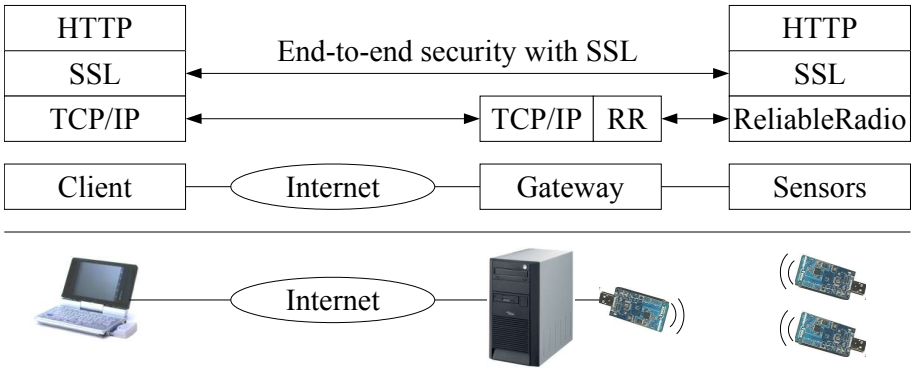


Figure 5.1: Architecture of Sizzle

one or more embedded devices using a lower-speed wireless link optimized for power consumption.

- The gateway provides a single point for controlling and facilitating access to the embedded devices. It can manage a list of available devices and make the list accessible to interested parties. The gateway can implement various mechanisms including address-based filtering to enforce selective access from across the Internet. The gateway is also the ideal place to log all interactions with the embedded devices.
- The gateway can serve as a performance-enhancing proxy. In particular, the TCP protocol interprets packet loss as an indication of congestion. This causes TCP to perform poorly when the connection involves a wireless hop [4]. An approach for alleviating this performance degradation splits the end-to-end path at the wireless link boundary – precisely where the gateway is situated.

  Terminating TCP at the gateway and using a special purpose reliable protocol between the gateway and the embedded devices has several benefits:

  – The special purpose protocol can be made simpler because it need not support end-to-end congestion control across multiple, possibly heterogeneous, links.
  – It can be tailored for the special loss characteristics of the single link.
  – Local packet loss recovery improves overall performance.

There is an important difference between this security architecture and other gateway-based architectures for connecting small devices, most notably WAP 1.0, where the gateway sees all traffic in the clear – decrypting incoming data and re-encrypting them before passing them along. In those architectures, the gateway needs to be trusted and compromising the gateway compromises all connections passing through it. With this approach, the security provided by SSL extends all the way from the client to the embedded web server. All data stay encrypted as they cross the gateway so even if the latter is compromised, an attacker is unable to view or alter any data.

5.3 Networking

For maximum partability, the radio protocol of Sizzle is built upon the active messaging layer of the TinyOS operating system. The active messaging layer provides means for

sending and receiving of packets[2] with a maximum payload of 28 bytes over the radio and over a serial connection.

For communication over the radio, TinyOS implements the B-MAC protocol, whose energy efficiency relies on the availability of a low-power listening mode in the radio transceiver. While this mode is available in the CC1000 transceiver of the Mica2 motes, it is missing in the CC2400 transceiver of MicaZ and Telos. However, the active messaging layer provides an interface which allows higher layers to turn the radio transceiver on and off, and schemes to conserve energy can be added.

The TinyOS networking stack does not incorporate any kind of acknowledgment scheme for reliable communication. Although it is easy for higher layers to explicitly acknowledge the reception of every single packet, this method effectively doubles the number of transferred packets and a more efficient way must be found. The only reliability-related functionality TinyOS already provides is that it performs a cyclic redundancy check to detect and drop bad packets.

Besides performance, important requirements for the radio protocol for Sizzle are reliability, energy efficiency, and the ability to transport large messages. However, only with the basic functionality provided by TinyOS, none of these requirements can be met for all platforms. This section describes a protocol, running on top of the active messaging layer of TinyOS, which offers significant improvements.

5.3.1 Protocol for Reliable Communication

The "reliable radio protocol" (subsequently called RR protocol) is split into two layers: The message layer is responsible for the reliable transmission of messages of arbitrary size consisting of one or more packets. The transport layer implements a mechanism for flow control and adds the concept of connections which can be opened or closed.

Message Layer

The basic reliable radio protocol is a combination of an ACK-based and a NACK-based scheme, as depicted in Figure 5.2. The first and the last packet of a message requires explicit acknowledgment, but not the packets in-between. Packets contain sequence numbers, so that the recipient is aware of missing and retransmitted packets. When a packet is missing, the recipient sends a NACK containing the missing sequence number and the sender retransmits all packets beginning with the given sequence number.

Acknowledged packets serve as synchronization points for the states of the sender and the receiver. The explicit acknowledgment of the first packet ensures that the recipient is ready to receive and that the radio link is still working, and when receiving the acknowledgment of the last packet the sender can be sure that the whole message has been transmitted successfully.

A special case are messages which consist only of a single packet; they may be sent without explicit acknowledgment. This functionality is required by the energy conservation feature described in a later section.

Avoiding Duplicate NACKs. The problem of duplicate NACKs becomes apparent when looking at Figure 5.2.c. In this situation it is likely that the sender already initiated

[2] In this work, the term *packet* refers to the content of a single transmission, and a *message* consists of one or more packets. TinyOS has no concept of messages spanning multiple packets, and uses the term *message* for the content of a single transmission.

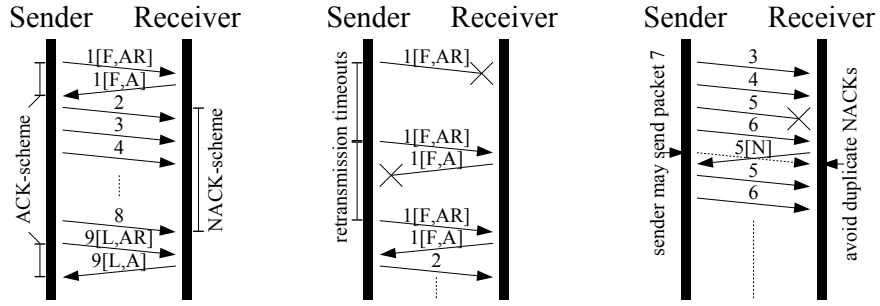


Figure 5.2: Basic Scheme for Reliable Transmission of Messages. Packets are labeled with the sequence number and flags in square brackets. The flags are: F=first packet of a message, L=last packet of a message, AR=ACK required, A=ACK, N=NACK.

the transmission of packet 7 before it receives the NACK for packet 5. Now the protocol has to prevent the transmission of a duplicate NACK, but has to make sure that a required retransmission can take place, for example when the first NACK has been lost.

In this protocol, after a packet with sequence number n triggered a NACK, the receiver does not send another NACK the first time it receives the packet $n+1$. Also, the receiver will not acknowledge the last packet of a message unless no other packet is missing.

After the sender received the NACK it begins to retransmit packets beginning with the requested sequence number and this scheme obviously works. When the NACK is lost, however, there are two scenarios. When packet $n+1$ is not the last packet of the message, more packets will be transmitted and trigger at least one more NACK. Otherwise, when packet $n+1$ is the last packet it requires explicit acknowledgment which will not be sent and the resulting retransmission of packet $n+1$ triggers another NACK. Thus the scheme to avoid duplicate NACKs does not negatively affect the reliability of the protocol.

Packet Headers. To implement this protocol the header fields of a TinyOS packet are not sufficient, and a part of the payload must be sacrificed for additional header fields. A packet consists of the following fields:

- Destination address. The ID of the destination device.
- TinyOS message type. Used to do some filtering in the TinyOS networking stack. This field is not used in the protocol and has a constant value.
- Group of the destination. Can be used in large networks where nodes are arranged in different groups.
- Length of the data.
- Data. Payload with a maximum length of 28 bytes, containing protocol-specific header fields and the actual data:
  - Address of the sender. The ID of the sending device; is used for filtering on the side of the gateway because the source address uniquely identifies the connection.

  - Sequence number.
  - Flags: F=first packet of a message, L=last packet of a message, A=packet is an ACK, N=packet is a NACK, AR=an acknowledgment is required for this packet.
  - Message type. Used by the transport layer to distinguish between data and various types of control messages.
  - Data. Up to 22 bytes of data.
- 16-bit CRC. Value of a cyclic redundancy check used to cull corrupted packets.

The total size of a packet, excluding the hardware-specific preamble, is 36 bytes, of which only 22 bytes can be used. This is a poor ratio, but it was not possible to change the TinyOS packet size. The packet size it is very deeply interweaved in the TinyOS networking stack and changing it breaks too many things.

Transport Layer

The transport layer manages flow control and the state of the connection. It uses control messages (1) to request the establishment of a new connection, (2) to request the closure of a connection, and (3) to communicate whether the device is ready to receive more data or not.

Flow control is required only for data transfer originating at the gateway and is implicit most of the time. The embedded device is always able to receive a message *except* after it just received a data message – only in this case there might be extensive processing on the side of the device. The device then either sends a data message as response or it signals that it is ready to receive data by sending an empty control message ("ready"-message).

Due to resource constraints each device can have only one connection at any given time, and this connection is always between the device and the gateway. The transport layer uses two different control messages to communicate and to request changes of the connection state. The purpose of these messages is (1) to inform the device that a new TCP connection has been established with the gateway, (2) to inform the device that the current TCP connection with the gateway has been closed, and (3) to give the device the ability to request the closure of the TCP connection. On the side of the embedded device, the transport layer notifies the SSL stack when the connection state changes, such that the state of the SSL stack can be reset. At the same time the device is able to close the connection in the case of a protocol error.

To ensure that the device returns to a known state, even in the case of an unforeseen error, the transport layer of the embedded device closes an open connection after a certain idle time and sends the appropriate control message.

5.3.2 Energy Saving

So far this protocol does not incorporate any means of energy conservation. The radio transceiver of the embedded device must always be turned on, even when no connection is open, to be able to receive packets. The gateway is operated from the power line and also supplies the base station, hence energy saving is important only for remote devices and the base station's radio can remain turned on all the time. Because Sizzle is restricted to star networks, remote devices exclusively communicate with the base station. As long as

message transmissions are initiated by the remote device, no synchronization is required even when the device is in a low-power mode most of the time.

In this energy-saving scheme an embedded device is usually powered-down and wakes up periodically to poll for messages. The device sends a control message, a so-called "ready-sleep" message without the need for acknowledgment and waits for a short period of time with the radio in receive mode. When the device does not receive anything from the gateway it enters the low-power mode again. However, when there is a message waiting at the gateway, the gateway begins the transfer and the device stays awake.

The device also turns off the radio right after the reception of a message, which makes sense because the flow control mechanism prevents the gateway from sending another message until the device signals that it is ready. This behavior has a significant influence when the processing of a message takes a relatively long time, as is is the case with the SSL key exchange, where a public-key operation must be performed. Figure 5.3 illustrates the energy saving method with an exemplary message exchange.

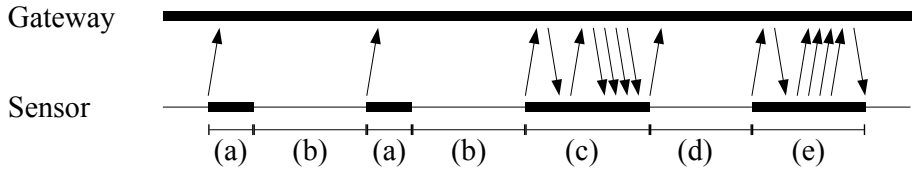


Figure 5.3: Message Exchange in Energy Saving Mode. (a) The device sends a polling message to indicate that it is ready and awake for a short time. (b) The device is in a low-power mode. (c) The gateway has a pending message and sends it right after it receives a polling message. (d) The device turns off the radio while it processes the message. (e) The device sends a message in response.

However, there can be a problem when the gateway does not receive the acknowledgment of the last packet of a message which needs a significant time to be processed. The gateway would try to retransmit this packet several times and would finally give up with an error. Therefore the retransmission attempts on the side of the gateway must last long enough to cover the longest period a device can spend processing a message. The interval between retransmissions should be short to improve the performance, but the interval should be long to avoid congestion of the radio link. The solution to this problem is to use a variable interval which gets extended with every retransmission attempt.

The downside of the energy saving scheme is that the polling interval must be added to the worst-case latency of messages originating at the gateway. A tradeoff must be made between the latency and the extend of the energy conservation. The length of the period in which the device stays awake after sending a polling message is constant and depends on the gateway's response time; in this implementation, 25 milliseconds appeared to be just long enough. With a polling interval of 2.5 seconds, for instance, the duty cycle of the device is 1% – with a reasonable additional latency almost 99% of the energy can be conserved.

In addition to conserving energy this scheme has another advantage: Because the gateway periodically receives polling messages from all devices in its range, it can discover new devices and maintain a list of devices which are currently available.

Another feature of radio transceivers, which supposedly helps to reduce the energy consumption, is the ability to adjust the transmission power. This idea makes sense

in dense peer-to-peer networks where the reduction of the transmission range for local communication helps to reduce congestion in other parts of the network. It also reduces the number of nodes which overhear a packet not intended for them and may help to reduce energy consumption of these nodes.

In a star network, where every node communicates only with the base station, congestion happens only there and reducing the transmission power results in no improvement. The opposite is true: the reduction of the transmission range defies the collision detection mechanism of the media access control protocol. More collisions would occur, undetectable by the senders, when the senders are out of each other's range. Thus it is desireable that all nodes are within range of each other.

The energy saving aspect is also very questionable in this application because a node typically spends much more time listening and receiving than transmitting. The chance of overhearing a packet with a different destination is already small because nodes spend most of the time powered down without the ability to receive. The increased complexity and the disadvantage mentioned above do not compensate for possible small energy savings. Therefore, in this protocol packets are always transmitted with maximum power.

5.3.3 Implementation of the Gateway

The gateway is implemented in Java using the TinyOS communication library. The TinyOS library is responsible for sending and receiving packets via the base station, which is connected with the PC over a serial link. For each available wireless sensor there is one TCP server waiting for incoming connections on a distinct port. After a connection is established the gateway converts between TCP and the reliable radio protocol.

Figure 5.4 shows the architecture of the gateway. The packet multiplexer receives

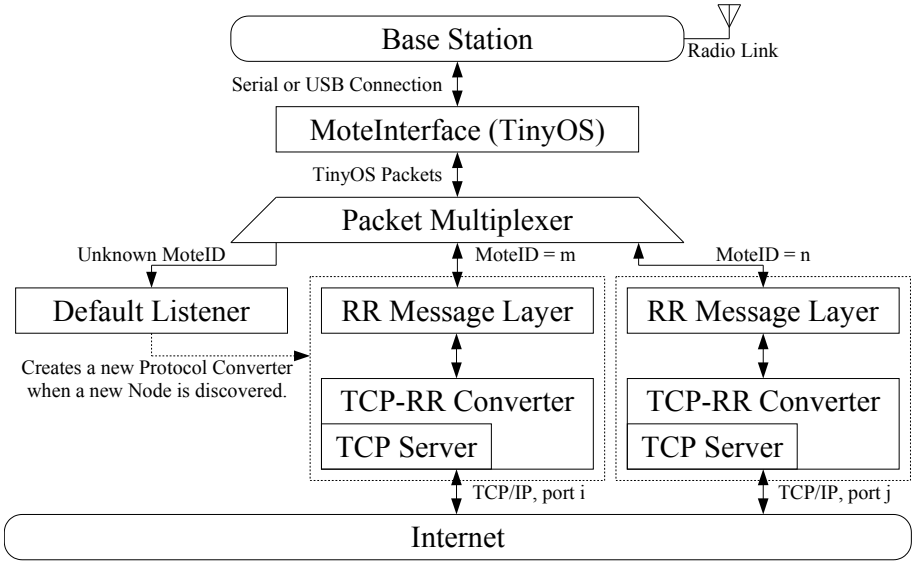


Figure 5.4: Architecture of the Gateway

incoming packets from the TinyOS library and forwards them, according to the ID of the sensor they are coming from, to different instances of the protocol converter. When the ID of the sensor is not known, the packet is forwarded to a default handler which creates a new instance of the protocol converter for the new sensor. It queries a database

of known sensors to see if the newly discovered sensor already has a configuration, like a preassigned port number and user comments. If there is no configuration a new port number is assigned to the device.

The protocol converter consists of two modules which represent the layers of the reliable radio protocol: The RR message layer and the the TCP-to-RR converter. The RR message layer is responsible for the fragmentation and defragmentation of messages and their reliable transmission.

The TCP-to-RR converter runs a TCP server in an own thread which listens for incoming connections on the assigned port. When there is a connection it sends the appropriate control message to the device and then forwards blocks of data. It keeps track of the state of the device and waits with transmissions of data blocks until the device is ready to receive, or until the device sends a polling message when it is in the energy saving mode.

The message layer of the reliable radio protocol is especially efficient when large messages are transmitted. The gateway takes advantage of this property and accumulates data from the TCP connection until a complete SSL record is in the buffer. The length of SSL records is transmitted in the clear in the record header and the gateway does not need to know more about the SSL protocol than it is necessary to find packet boundaries. Messages coming from the radio link, however, are directly forwarded to the TCP connection.

The TCP-to-RR converter also has an idle timer to detect when a node goes offline. This event causes the closure of an eventually open TCP connection, the shutdown of the TCP server, and the destruction of the instance of the protocol converter.

Because Sizzle uses persistent HTTP, but only supports one open connection, it must be possible that an idle connection gets closed when another client tries to access the same device. When there is a new TCP connection pending, the gateway simply closes the old one and sends the appropriate control message to the device. Then the new connection can be established in the usual way.

5.3.4 Accessing the Wireless Network

It is pretty difficult for a user to keep track of all sensors and their associated port numbers. The gateway has all this information and knows about the current state of each individual device. It makes good sense to add a user interface to the gateway which allows for convenient access to all devices and gives feedback on their availability.

Because the interaction with the wireless sensors is web-based, it is seemed to be appropriate to incorporate a web server into the gateway. The returned pages would have highly dynamic content and since the gateway is already implemented in Java, the use of a web server with support for Java servlets makes sense. The perfect choice was the open-source “Miniature Java Web Server” [34], which is very small, supports servlets, and can be easily integrated into an existing Java application. Further, this web server can be configured to use the Java Secure Sockets Extension library to enable secure connections.

The main web page of the interface is a list of all known devices – devices which exists in the gateway’s database, including their user-editable descriptions. The entry of a device which is online is emphasized and includes a link to access the web server on the device. Thanks to this web server on the gateway, the user only needs to remember the address of the gateway, but not the port number of each individual sensor.

Chapter 6

Implementation of Sizzle

This chapter describes the implementation of the secure web server stack for the wireless sensors. First, an overview over the software modules and their interaction is given, and then the implementation is detailed in a top-down order, where the focus lies on elliptic curve cryptography. The chapter ends with a discussion about how to deal with the extremely tight memory constraints which are especially prevalent on the Mica motes.

6.1 Overview

Figure 6.1 shows an overview of the software stack which allows to run web-based applications on wireless sensors. This section describes the various modules and the interaction

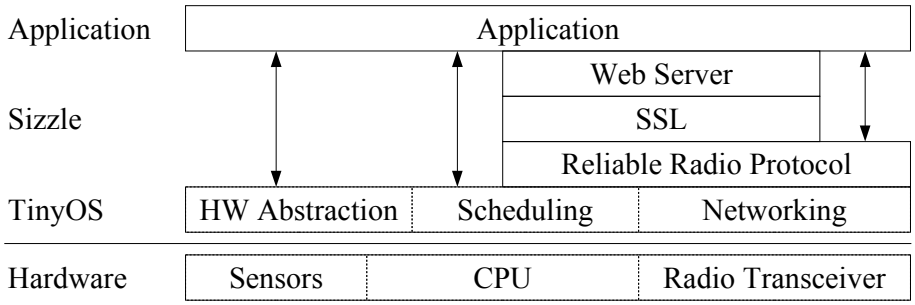


Figure 6.1: Software on the Wireless Sensor

amongst them. The structure of the sensor hardware, the functionality of TinyOS, and the reliable radio protocol have been described in earlier chapters.

Data coming from the gateway is accumulated in a buffer by the reliable radio protocol layer. After the reception of each message a method of the SSL layer is called which is responsible for processing the content of the buffer. The implementation of the gateway, which is capable of finding SSL record boundaries, assures that each message contains exactly one SSL record. After processing the record, the SSL layer can use the same buffer to store a response of one or more SSL records. A method of the radio protocol layer must be called to request further action, like the transmission of the content of the buffer or the closure of the connection.

The buffer has a size of 1KB which is enough to hold large SSL records, like those containing the server certificate, the HTTP request, or a small web page in the HTTP

response. Due to the tight memory constraints a second buffer, which would be required to simultaneously send and receive messages, must be waived. However in the SSL handshake and in the HTTP protocol both parties send alternately and it is always clearly defined which party is next to send. Thus it is not necessary that the device is able to receive and transmit at the same time, and no second buffer is needed. Application-specific protocols on top of SSL, however, may require simultaneous sending and receiving, but in such protocols the message sizes can be expected to be smaller and so can be the size of the buffers.

After the reception of an SSL record, the SSL layer decrypts it if necessary and processes it when the it belongs to the handshake, alert, or change cipher spec protocol. Otherwise the SSL layer passes the decrypted record on to the web server, which generates the response. The SSL layer encrypts the response and requests the reliable radio layer to transmit it. All of these operations are performed in-place in the very same buffer.

The separation between the web server and the application is not as clear, since the web server is part of the application. The web server receives HTTP requests and parses them and the application is responsible for generating the appropriate response. Nevertheless, the application contains functionality which goes beyond that of a web server, like the gathering of sensor readings or the control of actuators. An application might also need to communicate with other sensors completely outside the context of the web server, but the addition of this functionality remains subject of future extensions.

6.2 Demo Applications

The main intention of the implementation of Sizzle was to be able to demonstrate it to potential customers who are interested in the underlying technology. Two sample applications serve this purpose and show how this security architecture can be employed for remote monitoring and controlling. One application, a wireless thermostat, demonstrates how to control an actuator, and a second application, a light sensor, shows how acquired sensor readings can be made accessible.

6.2.1 Wireless Thermostat

The wireless thermostat application already existed and is mentioned in [19]. On the device, the current setting of the thermostat is indicated by light emitting diodes of different color and instead of the lights a real thermostat could be attached. The main web page of the user interface displays the current setting as a string (“off”, “heat”, or “cool”) and contains buttons to change the state.

I extended the existing application by an access control mechanism, where the device prompts for username and password before the setting can be changed. Once the user provided the right credentials, the user is associated with the SSL session and no more prompts are needed as long as the SSL session persists. This way also the SSL session reuse restores the associated user.

6.2.2 Light Sensor

The light sensor application measures the light intensity and stores the samples in a small buffer. A web-based interface allows users to see the current, the minimum, and the maximum light intensity, and displays a graph showing the development over time.

The light sensor can be configured to send notification messages to the gateway when the sensor readings are outside a configurable range. Notification messages are not secured and this feature is merely used to demonstrate that the behavior of wireless sensors can be configured through a secure web-based interface. In a real applications these messages would be secured using the existing cryptographic functionality.

6.3 The Web Server

The web server is an integral part of the application running on the wireless sensor node. For most web pages the server will have to generate dynamic content to display current sensor readings and settings. There are two approaches to generate dynamic pages depending on whether the majority of the pages is static or dynamic.

The first approach (similar to Java Servlets) is useful for pages with little static content. Implemented in C, static strings are copied to an output buffer and the copy instructions are interleaved with the generation of dynamic content. This method has the disadvantage that, in addition to the code to generate the dynamic content, each web page also requires code to copy the static portions.

In the second approach (similar to Java Server Pages) the template of the page is static and dynamic values are inserted at certain positions. In the C implementation, all pages are stored in an array of strings and the requested page is copied to the output buffer. Parts of the page are designated to be overwritten with dynamic content and initially contain spaces or default values. The only code that is needed for each page is to fill in the dynamic values. This method leads to a reduction in code size compared with the first approach. Nevertheless, there must be placeholders in the string constants, and each placeholder must be large enough to accommodate the largest possible dynamic value it may be replaced with. Especially for pages with long dynamic portions a significant overhead in the sizes of the string constants is introduced.

The web pages for the thermostat and the light sensor are mostly static, but a few pages, especially those for user management and the page to display the light intensity graph, are highly dynamic. To efficiently accommodate both types of pages the implemented web server uses a combination of the approaches described above. Each page can contain a static template which is copied from a string constant and then page specific code can fill in values or append the template with dynamic portions (see Algorithm 6.1).

Algorithm 6.1: Generation of Dynamic Web Pages

Input: HTTP request r, set of page templates t_i
Output: Web page p

```
1 Parse the HTTP request r and determine the index of the requested web page i.
2 Copy t_i to the response buffer p.
3 switch i do
4     case 0
5         Optionally replace portions of p with dynamic values.
6         Optionally append more dynamic content to p.
7     ...
8 return p.
```

6.4 The SSL Stack

The design decisions which led to a very small and fast SSL stack are outlined in Section 5.1.4. The major restrictions are that client authentication is not supported and that only a very limited set of ciphersuites can be used. Despite of these restrictions the SSL stack remains fully compliant with the specification.

6.5 Implementation of ECC for the Telos Motes

This section describes the implementation of elliptic curve cryptography for the Texas Instruments MSP430 microcontroller. It motivates the decision to write parts of the code in assembly and discusses to what extend an implementation in assembly is reasonable. The second part of this section describes the implementation of the long integer multiplication and investigates the impact of a slight improvement of the processor architecture on the performance.

The implementation of ECC for the Atmel ATmega processor, used in the Mica family of motes, has been done previously and is described in [21].

6.5.1 Elliptic Curve Arithmetic

Point Multiplication

There are several alternative algorithms to perform elliptic curve point multiplications of the form $Q = k \cdot P$. The performance of these algorithms depends on the structure of the scalar k and is compared assuming the average case where half of the bits of k are set. In this comparison n denotes the number of bits required to represent k: $n = \lceil \log_2 k \rceil$.

The simplest algorithm, the double-and-add algorithm, requires n point doublings and $n/2$ point additions. Algorithms using the non-adjacent form (NAF) of k get along with less additions; with NAF_2, for example, only $n/3$ additions are required. Other NAF-based algorithms with a larger window size further reduce the number of additions but require the precomputation of multiples of P. Precomputation takes time and the precomputed points require extra memory.

Algorithms which rely on precomputed points are especially well-suited when the same point is the operand of multiple point multiplications, as it is the case for the generator G of a curve, a multiplication with which is always needed to create an elliptic curve key pair. The only point multiplication an SSL server has to perform takes place after the reception of the *ClientKeyExchange* message, and the server has to multiply the client's public key (the point) by the own private key (the scalar). The client's public key, however, is an ephemeral key and different every time. Thus it would be necessary to compute the multiples of the point for each multiplication.

Table 6.1 compares the costs of NAFs with different window sizes with the cost of the double-and-add algorithm for multiplications by a 160-bit scalar. The cycle counts are derived from the measured execution times of the implemented algorithms for point addition and doubling.

Because the main objective of the implementation of Sizzle is to use as little resources as possible I decided to implement the binary NAF, NAF_2. NAFs with extended width do not offer performance gains high enough to justify the higher complexity of the code and the higher memory requirements. The width-2 NAF of a scalar can be computed efficiently with just two long integer additions and one shift, and the complexity of the

Table 6.1: Comparison of Algorithms for Point Multiplication. (A=point addition, D=point doubling)

Algorithm	Average Operations	Additional Memory (bytes)	Clock Cycles (millions)	Time relative to double-and-add
Double-and-add	$159D + 80A$	0	4.46	100.0%
NAF_2	$159D + 53A$	0	3.86	86.6%
NAF_3	$160D + 41A$	40	3.61	81.0%
NAF_4	$160D + 35A$	120	3.48	77.9%
NAF_5	$160D + 34A$	280	3.46	77.4%
NAF_6	$160D + 38A$	600	3.55	79.4%

algorithm for the point multiplication is comparable with the complexity of the double-and-add algorithm.

Point Addition and Doubling

As described in Section 2.3.3, the performance of point addition and doubling depends on the used point representation. Mixing Jacobian and affine coordinates instead of using only affine coordinates eliminates the field inversion in these operations at the cost of additional field multiplications and squarings. Also, the algorithms for point doubling and squaring can be implemented more efficiently when the curve parameter $a = -3$, which is the case with the used curve *secp160r1*.

The expected count of long integer operations for a 160-bit NAF_2 point multiplication (based on Table 2.4) is presented in Table 6.2. When using mixed coordinates, 211

Table 6.2: Long Integer Operations for a 160-bit NAF_2 Point Multiplication on the Curve $y^2 = x^3 - 3x + b$ (I=inversion, M=multiplication, S=squaring).

	Affine Coordinates	Mixed Coordinates
Point Additions	$53 \cdot (1I, 2M, 1S)$	$53 \cdot (8M, 3S)$
Point Doublings	$159 \cdot (1I, 2M, 2S)$	$159 \cdot (4M, 4S)$
Make Result Affine	—	$1 \cdot (1I, 3M, 1S)$
Sum	$212I, 424M, 371S$	$1I, 1063M, 796S$

inversion can be saved at the cost of 1064 additional multiplications or squarings. Thus mixed coordinates are beneficial as long as the inversion is slower than approximately five multiplications; this is usually the case when a hardware multiplier is available.

6.5.2 Assembly-Language Versus C Implementation

An ECC implementation is divided in two parts: the finite field arithmetic and the elliptic curve arithmetic. Operations in prime fields use regular integer arithmetic but the sizes of the operands are 160 bits or more and an additional reduction step is needed after each operation. The word size of the processor, however, is only 16 bits, and an operation on long integers must be split into multiple operations on 16-bit words.

When n is the number of words needed to represent a long integer, the number of operations for long integer addition and subtraction is in the order of $O(n)$. The runtime of the basic algorithms for multiplication and squaring is in the order of $O(n^2)$ and these operations have a very big impact, especially when the processor's word size is small. The cost of reductions can be significant, but for this implementation a curve is chosen which is defined over a prime field allowing for efficient reductions. Thus the main focus of optimization efforts must lie on the algorithms for multiplication and squaring, and it makes sense to implement these algorithms in assembly.

The other prime field operations can also benefit from an assembly implementation. For example, the addition of two long integers requires the addition of n pairs of words and the carry-out of one addition must be considered in the following addition. Like most other instruction set architectures the MSP430 implements a carry-flag in hardware and contains instructions to facilitate the addition of long integers. The implementation of a long integer addition in C, however, is cumbersome because the carry must be emulated in software and the compiler generates large and slow code. The access to the hardware carry-flag also facilitates the implementation of the subtraction and of bit-wise shift operations as they are used to multiply or divide by powers of two.

Based on this analysis I decided to implement all prime field operations, except the modular inversion, in assembly, with an interface allowing C code to use these functions. The modular inversion is only done once in the point multiplication and has a very small influence on the total execution time. The binary extended Euclidean algorithm for modular inversion, even when implemented in C, can still take advantage of the highly optimized implementations of the subtraction and the shift operation.

The elliptic curve operations, namely point addition and point doubling, have a structure that does not benefit from assembly optimization. The appropriate algorithms merely consist of a sequence of invocations of prime field operations, and their implementation is a sequence of function calls. The C compiler is perfectly well able to translate function calls to efficient code and leaves no room for optimizations.

6.5.3 Prime Field Multiplication

This section introduces the parts of the MSP430 architecture which are relevant for the implementation of the long integer multiplication. Then the multiplication in operand scanning form is compared with the multiplication in product scanning form. Finally a fast reduction algorithm tailored for the used prime field is presented.

The Multiply-Accumulate Unit of the MSP430

The MSP430 contains a multiply-accumulate unit able to do operations of the form

$$\mathrm{ACC} \leftarrow \mathrm{ACC} + a \cdot b,$$

where a and b are 16-bit operands and the accumulator ACC consists of two 16-bit registers and one additional 1-bit register to hold a potential overflow. This unit is not part of the CPU core but is integrated as a peripheral whose registers can be accessed like regular memory locations. Figure 6.2 shows the structure of the multiply-accumulate unit.

To use the multiply-accumulate unit one first loads one operand into "OP1" and then loads the second operand into "OP2" which triggers the computation. The result is ready after three clock cycles and can then be retrieved from the result registers. When the

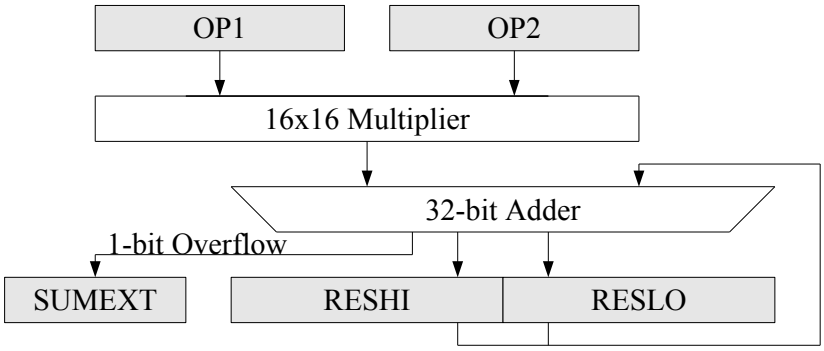


Figure 6.2: Multiply-Accumulate Unit of the MSP430 (Simplified). Shaded registers are accessible from outside.

first operand remains unchanged in subsequent computations it is not necessary to load it again; only "OP2" must be loaded to start the computation.

The "SUMEXT" register is only one bit wide and contains the carry-out of the latest addition. In order to extend the accumulator to a size greater than 32 bits it is necessary to manually add the content of "SUMEXT" to a CPU-internal register after each multiply-accumulate operation.

Instruction Timing of the MSP430

For the investigation of the timing of the multiply-accumulate operations only *move* and *add* instructions are important. *Move* is required to transfer operands and the result between the multiply-accumulate unit and the CPU registers. *Add* is required to manually accumulate the "SUMEXT" register.

Both instructions take a source and a destination argument. The destination can be a CPU-internal register or a memory location referenced by a constant. Additional addressing modes are available for the source; it can also be a constant or a memory location referenced by a register. Table 6.3 shows the relevant instruction times for combinations of these addressing methods. When the source is a memory location referenced by a register,

Table 6.3: Instruction Timing of the MSP430

Source	Destination	Clock Cycles
register	register	1
register	memory referenced by a constant	4
memory referenced by a register	register	2
memory referenced by a register	memory referenced by a constant	5
memory referenced by a constant	register	3
memory referenced by a constant	memory referenced by a constant	6

the register value can optionally be incremented after the operation without influence on the instruction timing. When a single memory location is the source of multiple instructions it may make sense to keep this location in a register to save one cycle per instruction at the cost of occupying one register.

One multiply-accumulate operation takes three clock cycles. However accessing the unit's registers requires multiple clock cycles and usually no additional delays are required. The only case where a delay of one cycle must be added is when trying to read a part of the result directly after loading "OP2" and the part of the result is referenced by a register.

Comparison of Two Multiplication Algorithms

In Section 2.2.2 two algorithms for long integer multiplication are presented. Algorithm 2.3 performs the multiplication in operand scanning form, and Algorithm 2.4 performs the multiplication in product scanning form. Both algorithms require two operations involving the multiply-accumulate unit: the multiply-accumulate operation and the shifting of the accumulator to the right by 16 bits.

The operand scanning form requires a 32-bit accumulator, while the product scanning form requires additional bits, and the content of the "SUMEXT" register must be accumulated in software. The operand scanning form, however, requires the shifting of the accumulator after each multiply-accumulate operation.

Table 6.4 compares both algorithms with respect to the operations involving the multiply-accumulate unit when multiplying two 160-bit integers, each one consisting of ten 16-bit words. These cycle counts assume that the memory locations of "RESLO", "RESHI", and "SUMEXT" are stored in registers to accelerate the operations where one of these memory locations is the source operand. The result is stored in an array in memory. To shift the accumulator one must copy the high word to the low word and load the high word with zero or with the accumulated overflow.

Table 6.4: Operations Involving the Multiply-Accumulate Unit

Operand Scanning Form			
Operation	Cycles	Op. Count	Total
Load OP1	5	10	50
Load OP2	5	100	500
Add RESLO to the Result	5	100	500
Shift the Accumulator	9	100	900
Sum			1950
Product Scanning Form			
Operation	Cycles	Op. Count	Total
Load OP1	5	100	500
Load OP2	5	100	500
Accumulate SUMEXT	2	100	200
Move RESLO to the Result	5	20	100
Shift the Accumulator	9	20	180
Sum			1480

The cycle counts do not consider the operations needed to implement the loops. Since both algorithms have a similar structure I expect the overhead of the control-flow to be similar as well, and the relative advantage of the product scanning form to remain in the fully implemented algorithms. Also the delay of one cycle, which is required between

loading the second operand and accessing parts of the result, is not considered because this delay slot can be filled with a useful instruction.

Implementation of Multiplication, Squaring and Reduction

The resulting implementation of the multiplication in product scanning form has an execution time of 1790 clock cycles. This time is still faster than the theoretical minimum of the operand scanning form.

The algorithm for squaring is essentially the same as the algorithm for multiplication with both operands being equal. This results in a reduction of multiply-accumulate operations when it is possible to compute $\text{ACC} \leftarrow \text{ACC} + 2 \cdot a \cdot b$. The most efficient way to do this with the multiply-accumulate unit of the MSP430 is to load the first operand one time and to perform the multiply-accumulate operation twice by loading the second operand two times. The accumulation of "SUMEXT" in software however must also occur twice. This way one can reduce the number of loads of the first operand by 40, which results in a theoretical reduction of the execution time by 200 clock cycles. The implementation shows an improvement close to the expected value: the squaring of a 160-bit integer takes 1600 clock cycles.

Algorithm 2.5 for reductions modulo $2^n - c$ can be optimized for the case that $c = 2^{31} + 1$. In the following algorithm, q and r are 160-bit integers which first contain the upper and the lower part of the number to reduce.

Algorithm 6.2: Fast Reduction Modulo $2^n - 2^{31} - 1$

Input: Number to reduce (qr)
Output: $r = (qr) \bmod (2^n - 2^{31} - 1)$

```
1 for i = 1 to 2 do
2 |   q̄ ← q
3 |   (qr) ← r + q̄
4 |   (qr) ← (qr) + 2^31 · q̄
5 if q ≠ 0 or r ≥ (2^n − 2^31 − 1) then r ← r − (2^n − 2^31 − 1)
6 return r
```

The assembly implementation of this algorithm, which takes advantage of the reduction of the size of q after each iteration, has an execution time of 200 clock cycles for the most likely case that the final subtraction in line 5 is not needed.

6.5.4 Performance of the ECC Implementation

The execution time of the complete point multiplication is 3.86 million clock cycles which is equivalent to 483 milliseconds at a clock speed of 8 MHz. Figure 6.3 splits this time into the times required by the most important algorithms. The remaining 6.7% consist of other prime field operations, the control-flow of the functions performing elliptic curve arithmetic, and the overhead for method calls.

Out of the 3.86 million clock cycles 67% (2.59 million cycles) are spent in operations transferring data to and from the multiply-accumulate unit. Architectural improvements of the multiply-accumulate unit and of the way it is accessed can potentially yield a significant speed-up of the point multiplication.

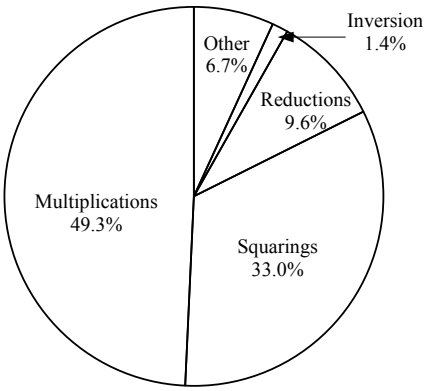


Figure 6.3: Time spent in various Parts of the Point Multiplication

6.5.5 Possible Improvements of the Multiply-Accumulate Unit

Because the multiply-accumulate unit is not implemented in the CPU access to its the memory-mapped registers is relatively expensive. Unfortunately the instruction set of the MSP430 leaves no free opcodes such that no instructions can easily be added. However it is be possible to modify the structure of the multiply-accumulate unit to optimize it for long integer multiplications by reducing the number accesses to registers.

In this section improvements are proposed which only add functionality and do not break backward-compatibility. These modifications are inspired by the work of Großschädl *et al.* [17] who proposed and implemented similar extensions for the SPARC architecture.

One observation of the that can be made from Table 6.4 is that the access to the "RESLO" register and the subsequent shifting of the accumulator is is a very common operation and consumes a lot of clock cycles. For the algorithm in operand scanning form these operations can be implemented in the following way:

```
add RESLO, RESULT[i];   // 5 cycles: add RESLO to the result
mov RESHI, RESLO;       // 5 cycles: move RESHI to RESLO
mov 0, RESHI;           // 4 cycles: load RESHI with 0
```

I propose the addition of another register, "RESLOSHIFT", which mirrors "RESLO" but triggers a shift of the accumulator to the right by 16 bits after a read-access, so that the last two lines of the code example above are not needed. This modification reduces the execution time of the long integer multiplication in operand scanning form by 900 clock cycles, making it superior to the product scanning form.

To accelerate the multiplication in product scanning form, an extension of the accumulator is necessary. Instead of just storing the overflow of the 32-bit addition to "SUMEXT", the overflow could also be accumulated in an additional register, "RESEX". This modification alone saves only 200 clock cycles, but together with the hardware implementation of the shift operation, 380 cycles can be saved in total[1]. The same savings can be expected for the squaring algorithm.

Table 6.5 shows the impact of the proposed improvements on the cycle counts of instructions accessing the multiply-accumulate unit. I never implemented the multiplication

[1]The shift operation would have to shift the extended accumulator including "RESEX". This modification makes no difference for the operand scanning form because in this algorithm there is no overflow of the accumulator and "RESEX" is always zero.

Table 6.5: Impact of the Improvements on the Cycle Counts Associated With the Multiply-Accumulate Unit

Operand Scanning Form	
Original Architecture	1950
Savings	900
Proposed Architecture	1050
Product Scanning Form	
Original Architecture	1480
Savings	380
Proposed Architecture	1100

in operand scanning form and therefore cannot tell whether a full implementation would be faster than the implementation of the product scanning form.

With a total number of 1859 field multiplications and squarings needed to perform the scalar multiplication, these modifications can save roughly 700,000 cycles or 18%. The full scalar multiplication then takes 3.16 million cycles or 395 milliseconds.

6.6 Implementation of RSA

To be able to compare the performance of ECC with the performance of RSA and to make Sizzle compatible with web browsers not supporting ECC key exchange I implemented RSA. The full details of this implementation are out of the scope of this work and I only present the comparison of both algorithms in the subsequent chapter.

6.7 Other Cryptographic Primitives

Besides public-key cryptography, Sizzle requires RC4 for symmetric encryption, the hashing functions SHA1 and MD5, and a random number generator.

6.7.1 Symmetric Encryption

The only symmetric cipher supported is RC4 [33], a very efficient stream cipher allowing for very small implementations.

The implementation is taken from Christophe Devine's collection of C implementations of cryptographic algorithms [12]. Because of the low complexity of RC4 no optimizations were possible.

6.7.2 Hashing Functions

The hashing functions SHA1 and MD5 take a block of data and perform data-dependent modifications on an internal state. SHA1 and MD5 share a similar structure and the following explanation is valid for both algorithms. The modifications of the internal state are performed in various rounds with similar structure, but with different parameters for each pass. The two most important operations, taking place in each round, are bit-wise rotation and one out of a set of logic functions.

The implementations of SHA1 and MD5, like that of RC4, are based on the code from Christophe Devine [12]. These implementations are close to reference implementations and not very well optimized for code size or speed.

The first optimization step aimed at reducing the execution time. The bit-wise rotation of a 32-bit value a by b bits, implemented in C, has the following structure:

$$\text{rotate-left}_b(a) = \text{shift-left}_b(a) \oplus \text{shift-right}_{32-b}(a)$$

The Gnu C compiler (gcc), however, translates this into large and very inefficient assembly code, especially when the size of the operand does not match the word size of the processor. The use of inline-assembly for the rotation halves the execution time of the entire algorithm on the Atmel ATmega and leads to a reduction of roughly one third on the Texas Instruments MSP430.

The second optimization step aimed at reducing the code size. In the original code the round operation is implemented in a function and this function is called with different constant parameters for each round. The body of the implementation contains 64 (MD5) or 80 (SHA1) explicit function calls, and especially the passing of multiple parameters contributes to a large code size.

By moving the parameters for the rounds into an array it is possible to pack all rounds into a single loop, where the round operation is invoked with the values from the array. This way, the function call and the passing of the parameters only occurs once in the code. A further improvement is possible by allowing the compiler to inline the round function to completely eliminate the call. The inlining leads to a speed-up which more than compensates for the performance loss caused by the introduction of the loop. The results of this optimization are the reduction of the code size by one third and a small improvement in performance.

6.7.3 Random Number Generator

The requirements for the random number generator on the server are rather relaxed, because in the supported key-exchange algorithms the burden of generating the pivotal random numbers lies on the side of the client. The client needs a cryptographically secure random number generator to produce the shared secret in RSA handshakes and the ephemeral private key in ECC handshakes. The only random data the server has to generate is part of the *ServerHello* message and is transmitted in the clear.

The server uses a simple 16-bit linear feedback shift register based on an existing TinyOS module. The implementation is very small and fast, but it is shown that the algorithm does not deliver uniformly distributed random numbers [39].

The TinyOS implementation uses a constant random seed, which depends on on the ID of the mote. To at least avoid having the same sequence after every restart, Sizzle uses a variable seed. In Sizzle, the seed comes from a hardware timer value when the first random number is generated after initialization of the software. This happens in the first SSL handshake, which is initiated from an external source at a non-deterministic time. Due to the high speed of the timer and the complexity of the whole system, this method introduces considerable randomness.

6.8 Dealing With Resource Constraints

A major issue in the development for embedded systems are resource constraints. In the development of Sizzle the most restricting constraint was the RAM size, most notably on the Mica motes with only 4KB. The small size of the program memory of 128KB and 48KB on Mica and Telos, however, was never a limiting factor, and being aware of the problem in the design and implementation helped to keep the program size significantly lower.

While the static RAM usage can be determined after linking the application, the dynamic RAM usage is very difficult to estimate. Sizzle has no heap management and the stack is the only dynamic structure to keep track of.

6.8.1 Monitoring the Stack Size

The stack is used to store local variables and return addresses of function calls, and its size is determined by the nesting level of a function. Each nesting level adds a constant value and the size of local variables to the stack size.

Usually global variables are located in the lower part of the memory and the upper part is used by the stack. The stack starts at the highest memory address and grows towards lower memory regions, and when the stack gets too large it overwrites the values of global variables. This behavior leads to malfunctions which can be very difficult to trace because they might seem to be totally unrelated to the cause. Thus it is desireable to determine the maximum size of the stack to check if there is enough available memory.

For example, an early version of Sizzle had stack overflows when doing an RSA key exchange, but this overflow had no consequences until the next SSL handshake took place. The next handshake, no matter whether it used RSA or ECC, simply did not work and is was not clear why for a long time.

The true worst-case stack size can be estimated by measuring the stack size in tests or by analytically determining and adding the stack sizes of each concurrent thread of execution. Both methods are usually not accurate and only approximate the true result from either side:

$$\text{worst size seen in testing} \leq \text{true worst size} \leq \text{analytic worst size}$$

The analytic method is not accurate because typically not all parts of the code, and not all interrupt service routines can be preempted by further interrupt service routines.

Because a manual accurate analytic analysis is very time consuming and error prone, Regehr *et al.* proposed an automatic solution and developed the so-called *stack tool* [31]. The stack tool determines the call tree of a TinyOS program from a binary compiled for an Atmel microcontroller. It analyzes the code starting at each entry point (interrupt service routine or TinyOS task), finds and traces function calls, keeps track of instructions which change the stack size, and determines the possible stack size at each point in the program. Further, the tool considers which parts of the code can be preempted by interrupt service routines and determines the worst-case stack size at every instruction. The output of the tool is a graphical representation of the call tree and the upper bound for the stack size.

Unfortunately this kind of static analysis has limitations when dynamic function calls or branches are used and the stack tool does not work for complex applications like Sizzle. Because stack overflows lead to non-deterministic behavior and eventually to a leakage of

sensitive information they should at least be detected. For this reason Sizzle contains a stack monitor which can be included with a compile-time option.

When the program starts, the stack monitor fills the memory regions which can be used by the stack with a certain pattern. By checking for the presence of this pattern the stack monitor can determine the largest size of the stack up to the time of the check. There is a safety buffer of several bytes after the end of the data region, and when this buffer or a part of it is overwritten the application resets to a known state. This check is done periodically, especially before the device transmits data, to avoid leakage of secret information.

A dynamic web page is included in the application which contains debugging information about the system, including the maximum stack size. After a long uptime and extensive tests, one can be confident that the upper bound of the stack size is not much higher than the measured value.

6.8.2 Strategies to Reduce the Memory Footprint

There are different techniques to reduce the size of the static and the dynamic memory usage of a program. The dynamic memory usage is equivalent to the maximum stack size and can be reduced by avoiding deep nesting levels of function and by reducing the size of local variables.

Allowing the compiler to inline functions is one strategy to reduce the nesting level with the disadvantage of a possibly larger code size. This approach is also discussed in [31].

In the optimization of Sizzle, the reduction of local and global variables made the greatest impact. One strategy is to use *union* types wherever possible. A union allows to declare different variables or data structures residing at the same memory location. This is possible as long as the variables are not used at the same time.

One example is the computation of hash values in the SSL record layer, where always both hashing algorithms, MD5 and SHA1, are used. The algorithms, however, are used sequentially and their rather large state variables can be placed in a union. The same is true for the premaster secret and the master secret in the state of the SSL protocol. The SSL protocol first generates the premaster secret and then derives the master secret from it. The premaster secret is no longer needed afterwards, and it is legitimate to combine both variables in a union.

In short, it it important to analyze the minimum scope of each variable and to find and combine variables whose scopes do not overlap.

Chapter 7

Evaluation of Sizzle

This chapter investigates the memory usage, performance, and energy consumption of Sizzle. Finally it evaluates the resulting system with respect to the requirements which have been established in Section 5.1.3.

7.1 Memory Usage

I used the tool "objdump" to determine the usage of program memory and RAM for global variables. This tool lists the sizes of code and data blocks either of the complete application or of the individual modules. I measured the stack sizes as described in Section 6.8.1. Unfortunately a static analysis of the stack usage was not possible and the results may not represent the worst-case values.

Table 7.1 shows the memory consumption of Sizzle including the simple thermostat application. With RSA key exchange the total RAM usage on the Mica family of devices

Table 7.1: Memory Consumption (in Bytes) of a Wireless Thermostat Application Embedding Sizzle.

	Flash	RAM (static)	RAM (max. stack size) w/ ECC	RAM (max. stack size) w/ RSA
Mica2dot	48,882	3,094	656	950
Mica2	49,116	3,094	656	950
MicaZ	48,516	3,040	629	925
TelosB	40,988	2,780	651	853

approaches the limit of 4096, and ECC has the advantage that it requires about 300 bytes less memory.

Table 7.2 shows how much program memory is required for the various cryptographic primitives and for the implementation of the SSL protocol. Because of its 16-bit architecture the MSP430 can operate on larger operands than the ATmega, resulting in a more compact code. This difference is especially big for the RSA implementation because it mainly consists of long integer arithmetic which is able to take most advantage from larger operand sizes. The MSP430, however, has only a little more than one third of the program memory the ATmega has. With only ECC enabled, the amount of flash memory dedicated to cryptography is 14% on the ATmega, and 26% on the MSP430.

Table 7.2: Bytes of Flash Memory Devoted to Cryptographic Code in Sizzle for the Atmel and MSP Processors. Numbers after the plus sign indicate additional memory used to store certificates, keys, and other related constants.

	Atmel ATmega (128KB max)	MSP430 (48KB max)
RSA	6,444 + 1,349	3,180 + 1,350
ECC	5,348 + 268	3,328 + 268
SHA1	2,046	1,486
MD5	2,442	1,742
RC4	228	216
SSL	7,556	5,848

7.2 Performance

7.2.1 Experimental Setup

Both microcontrollers contain a number of embedded timers with a resolution of 16 bits, able to count clock cycles divided by an adjustable value. A greater divisor extends the measureable range at the cost of a higher granularity and the divisor can be configured such that the measureable range matches the expected execution time of the code.

To measure the execution time, a timer is initialized with zero before the code to measure, and the content of the timer is read afterwards. Either operation on the timer requires only one assembly-statement whose execution time is negligible compared with that of the measured code. The results of the measurements are stored in variables and made accessible through a dynamic web page.

I measured the execution times of a single invocation of each cryptographic function as described above and counted the number of times these function are invoked during the handshake and the data transfer. This way I could calculate the total time spent in the cryptographic functions.

I performed the measurement of the time to transmit individual messages in the gateway. The time to transfer a message from the gateway to the device was measured between the transmission of the first packet and the reception of the acknowledgment for the last packet. A transfer in the other direction was measured between the reception of the first packet and the transmission of the last acknowledgment.

The duration of the SSL handshake was measured by the gateway between the start of the transmission of the *ClientHello* message and the end of the reception of the Server's *Finished* message. In the case of session reuse the transmission of the Client's *Finished* message marks the end of the handshake and stops the measurement. The duration of the data transfer was measured in a similar way between the transmission of the request and the reception of the response.

7.2.2 Results

This section presents the results of the measurement of the execution times of the handshake, the data transfer, and finally, the cryptographic primitives.

SSL Handshake

Figure 7.1 shows the times needed to perform SSL handshakes with different sensor platforms. As expected, the abbreviated handshake with session reuse is fastest and the

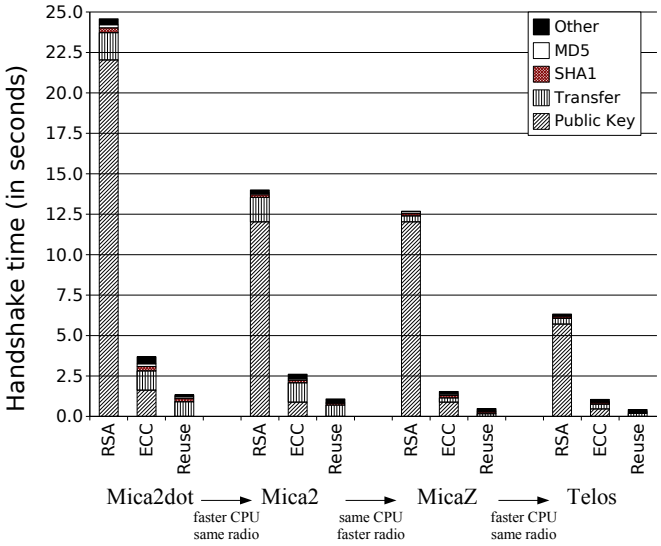


Figure 7.1: Handshake Times on Different Platforms

handshake with RSA key exchange is slowest. The time for RSA handshakes is dominated by the public-key operation while in the ECC handshake the data transfer plays a significant role, too.

A full RSA handshake takes 6 to 25 seconds depending on the platform (where the RSA decryption contributes 5.7 to 22 seconds); a full ECC handshake takes 1 to 4 seconds (with the ECC point multiplication contributing 0.5 to 1.6 seconds); and an abbreviated handshake with session reuse takes 0.5 to 1.5 seconds.

The figure also shows the impact of speeding up the CPU and the wireless network on the handshake times. The RSA handshake benefits most from CPU improvements whereas the abbreviated handshake benefits most from networking improvements.

Data Transfer

Figure 7.2 compares the times required to fulfill HTTP(S) requests with different page sizes for various platforms. The measurements were obtained with short HTTP requests of approximately 100 bytes and responses of 100, 500, and 900 bytes.

Secure data transfers are generally slower and the factors contributing to this difference are: the verification and computation of message authentication codes, the overhead of transmitting them, and the time required to perform symmetric decryption and encryption.

Yet the difference between secure and insecure communication is only small and almost imperceptible for the user. On Mica2dot and Mica2 motes, HTTPS transfers are 100ms to 300ms (depending on the page size) slower then HTTP transfers; on MicaZ and Telos motes the difference lies between 30ms and 100ms.

To obtain the time for a complete transaction an eventual SSL handshake must be considered. However, because of Sizzle's ability to use persistent HTTP a handshake is

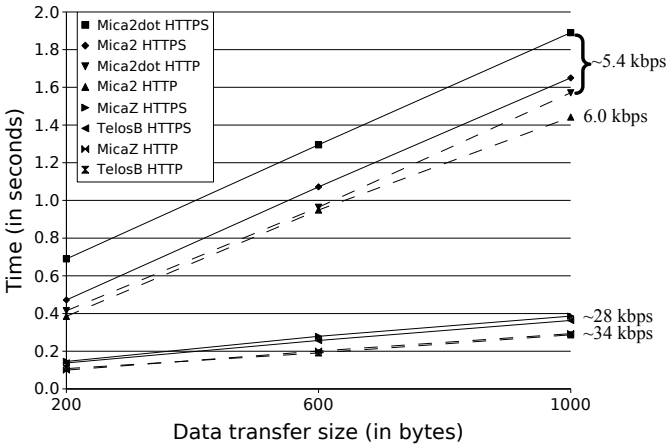


Figure 7.2: Data Transfer Times on Different Platforms

required only for the first request and subsequent requests can be handled without further delays.

7.3 Energy Consumption

7.3.1 Experimental Setup

In order to obtain plots of the power consumption during various operations I derived the current draw of the Telos mote from the voltage drop across a resistor, measured using a digital storage oscilloscope. During the measurements the operating voltage was 2.6V which is close to the mean voltage in the lifetime of the batteries. I used Matlab to compute the power consumption and finally to integrate the power to arrive at the amount of energy consumed in the SSL handshake and the data transfer.

The changing voltage drop across the resistor introduces an error because the supply voltage of the mote varies and the current is dependent on the supply voltage. I performed static measurements of the power consumption of the three most common operating modes of the mote (idle, CPU active, radio active) and compared the results with the power levels in the dynamic plots. This way I could correct this error and gain confidence in the accuracy of the dynamic measurements.

To correlate the events of sending and receiving packets with the power consumption I recorded the time stamps of packet transmissions. I modified the TinyOS radio driver on the base station in a way that it indicates the transmission and reception of a packet by the logic levels of different output pins. I used a resistor network to generate distinct voltages for these events and measured the resulting voltage progression on the second channel of the oscilloscope. A post-processing in Matlab finally resulted in discrete points of time where packet transmissions take place.

In addition to plots of the power consumption over time I calculated the energy required by the individual cryptographic building blocks. Since the power consumption of the CPU in active mode is rather constant it is possible to calculate the amount energy consumed per clock cycle. Having measured the cycle counts of the cryptographic primitives I could derive their cost in terms of energy.

7.3.2 Energy Consumption of the SSL Protocol

SSL Handshake

Figure 7.3 shows the power consumption during an SSL handshake with ECDH key exchange and correlates it with the transmission of packets over the radio. It becomes

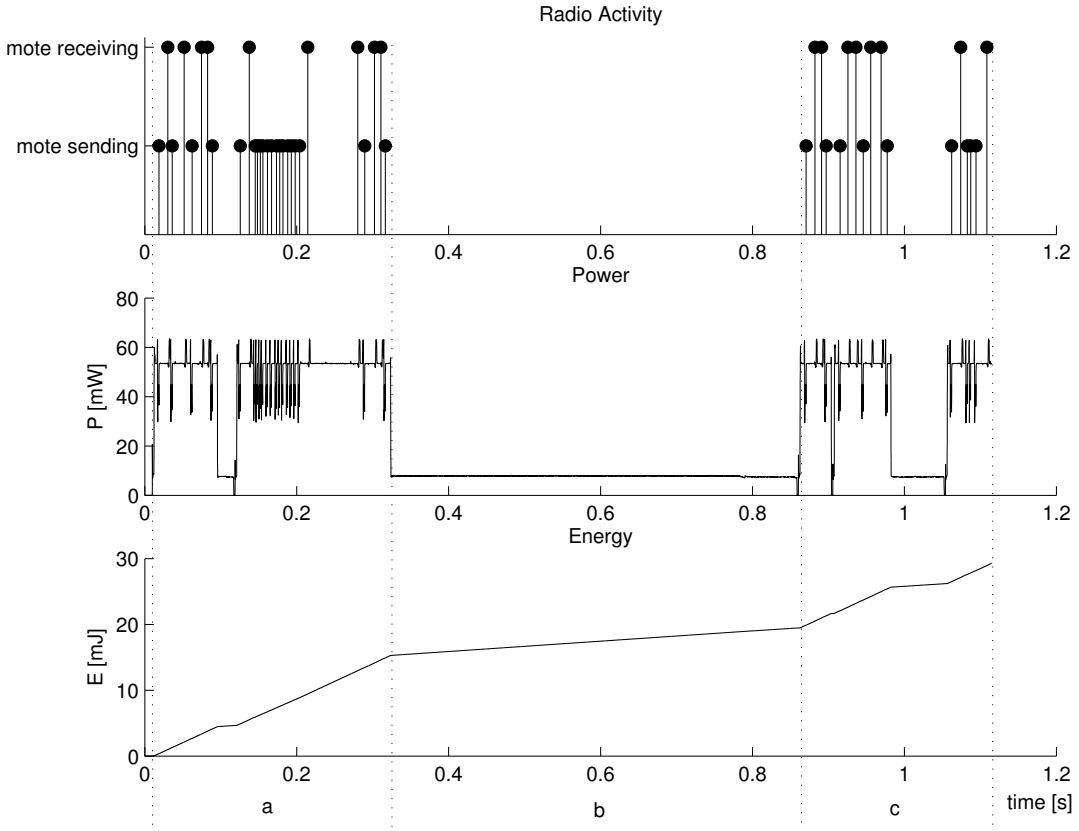


Figure 7.3: Power Consumption During an ECC Handshake. (a) Transmission of *Hello* messages, *Certificate* and *ClientKeyExchange*. (b) Public-key operation. (c) *ChangeCipherSpec* and *Finished* messages.

apparent that most of the energy is consumed by the radio and that the actual key exchange only has a minor impact. As illustrated in Table 7.3 the picture is different when RSA key exchange is used. In this case the energy used by the public-key operation exceeds the energy used for communication. Comparing the different types of handshakes one notices that more than 60% of the energy and 80% of the time can be saved by using ECC instead of RSA. The abbreviated handshake, which takes advantage of a previously made key agreement, reduces the costs even further.

The long block of data transferred in Figure 7.3.a is the certificate which is always the same for each individual sensor. The transmission of the certificate consumes roughly 5mJ which is 17% of the total energy consumption of the ECC handshake. A possible improvement of Sizzle's architecture would involve the caching of certificates at the gateway and the insertion at the appropriate point in the handshake.

Table 7.3: Comparison of the Costs of Three Different Types of SSL Handshakes

	Absolute Cost		Relative to RSA		
	Time [ms]	Energy [mJ]	Time	Energy	Energy in Communication
RSA Handshake	6,410	76.9	100.0%	100.0%	41%
ECDH Handshake	1,110	29.7	17.3%	38.6%	82%
Session Reuse	422	16.5	6.6%	21.5%	93%

Data Transfer

Figure 7.4 shows the radio activity and power consumption of the wireless sensor while processing an HTTPS request. The main difference between this plot and the plot for insecure HTTP requests is the duration of part (b) in which cryptographic functions are executed.

Figure 7.5 shows the energy consumption as a function of the size of the transferred data, potentially using multiple HTTPS requests following a single ECC or RSA handshake. A plot for insecure HTTP is also included to indicate the additional energy overhead of security.

The points where the plots intersect the Y axis denote the energy spent in an SSL handshake (76.9mJ for RSA, 29.7mJ for ECC), while the slopes indicate the energy needed to transfer unit data. The slopes for ECC and RSA plots are identical and only 15% steeper than the slope for insecure communication. These plots assume SSL record size does not exceed 1KB, the slope will be lower if larger SSL records are utilized.

While the energy cost of an SSL handshake is relatively high, it soon becomes marginal when persistent HTTPS is used to amortize that cost across multiple data transfers (see Figure 7.6). The point where the cost of data transfer exceeds the handshake cost is at about 1.3KB for ECC and at 4.3KB for RSA.

Cryptographic Primitives

Table 7.4 presents the costs of the implemented cryptographic primitives in terms of energy and time. The MSP430, when operating at 8MHz, consumes 7.8mW of power

Table 7.4: Energy Costs and Performance of the Cryptographic Building Blocks

	Energy [uJ]		Time [cycles]	
Asymmetric	Key Exchange	Ratio	Key Exchange	Ratio
RSA	46,690.00	12	47,889,408	12
ECC	3,780.00	1	3,877,376	1
Symmetric	Key Setup	Per Byte	Key Setup	Per Byte
RC4	10.40	0.03	10,668	34
Hash	One Time	Per Byte	One Time	Per Byte
MD5	1.56	0.24	1,604	248
SHA1	1.05	0.26	1,076	265

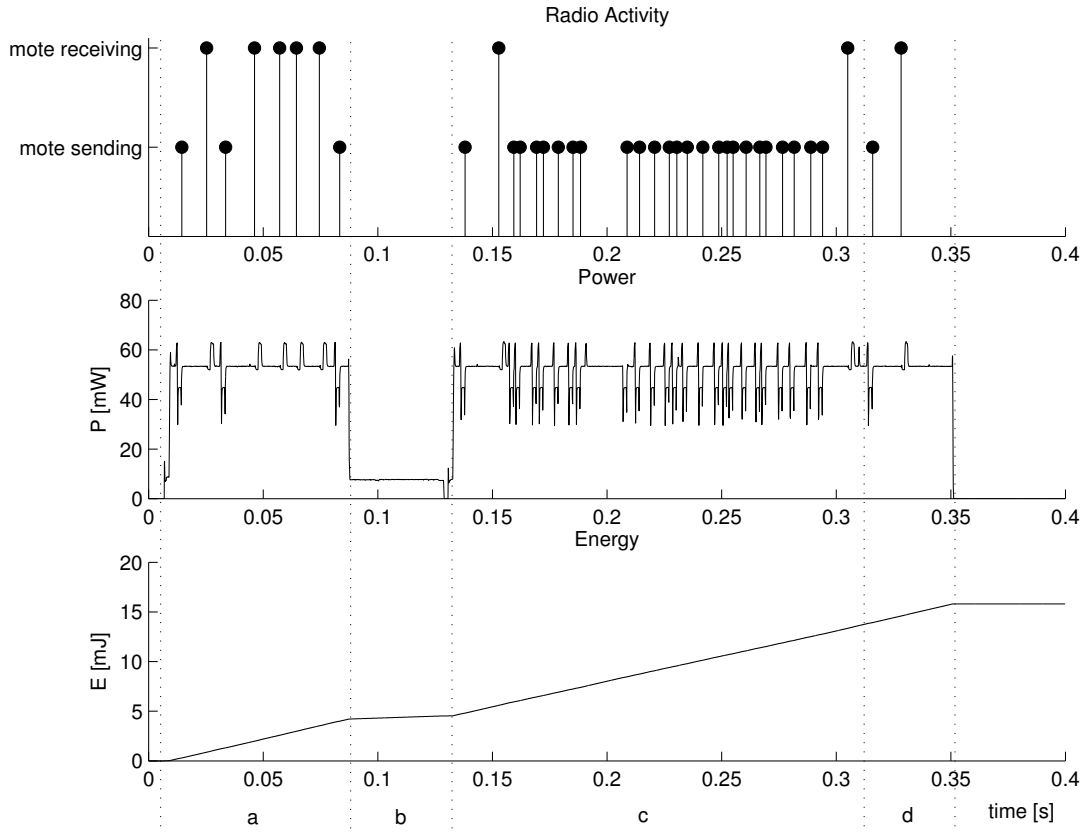


Figure 7.4: Application Data Transfer With Persistent HTTPS. (a) Reception of the HTTPS request (100 bytes). (b) Processing of the request and generation of the response (including symmetric cryptography and message authentication). (c) Transmission of the response (500 bytes). (d) Transmission of "ready-sleep" before the mote powers down.

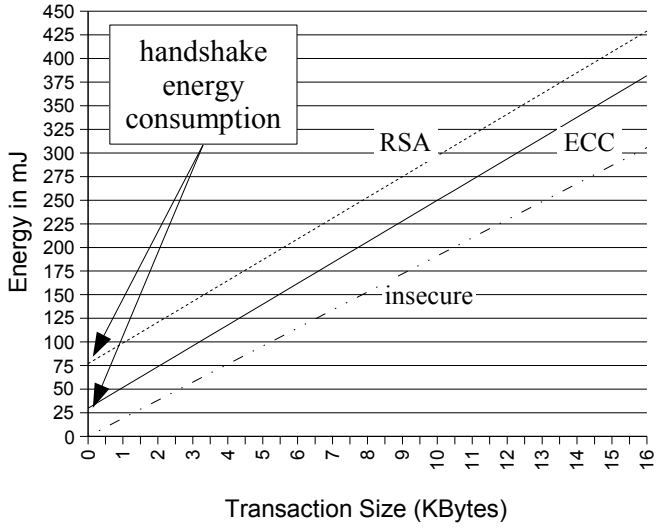


Figure 7.5: Energy Consumed as a Function of the Application Data Size

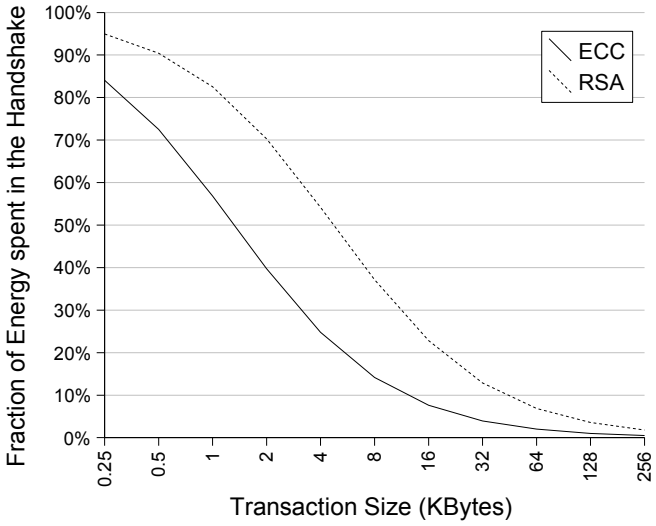


Figure 7.6: Fraction of the Energy Spent in an SSL Handshake Depending on the Transaction Size

which translates into 975nJ per clock cycle. The cycle counts of the various operations were measured and used to calculate the energy costs.

The cost of public-key cryptography seems to be overwhelming at the first sight but it amortizes well when larger amounts of data are transmitted. For example, the cost of an ECC key exchange is equal to the cost of encrypting 110 kilobytes of data using RC4 and to the cost of hashing about 15 kilobytes using MD5 or SHA1.

7.3.3 Energy Saving Mode of the Radio Protocol

The energy saving mode of the radio protocol is described in detail in Section 5.3.2. The radio sleeps most of the time and wakes up in regular intervals to send a polling message to the gateway. If there is a pending message to be sent to the device the gateway has a short period of time to initiate the transmission. In the other case, when this time elapses, the device powers down again.

Figure 7.7 shows the plot of the power consumption during this period and reveals that each polling attempt consumes roughly 1.5mJ of energy. An interesting value to calculate the lifetime of batteries is the average power the device consumes when it only polls for messages. The average power is a function of the polling interval and can be approximated using the following formula:

$$P_{avg} = 0.016mW + \frac{1.5mJ}{t_{interval}},$$

where 0.016mW is the power consumption of Telos in idle mode.

When the polling interval is 10 seconds, for instance, the resulting average power is 166μW. Given a pair of batteries with a capacity of 15,300J, this scheme can run for 213 days under the assumption that 20% of the capacity is devoted to polling. This is a significant improvement over the expected lifetime of roughly three days when the radio transceiver remains in receive mode all the time.

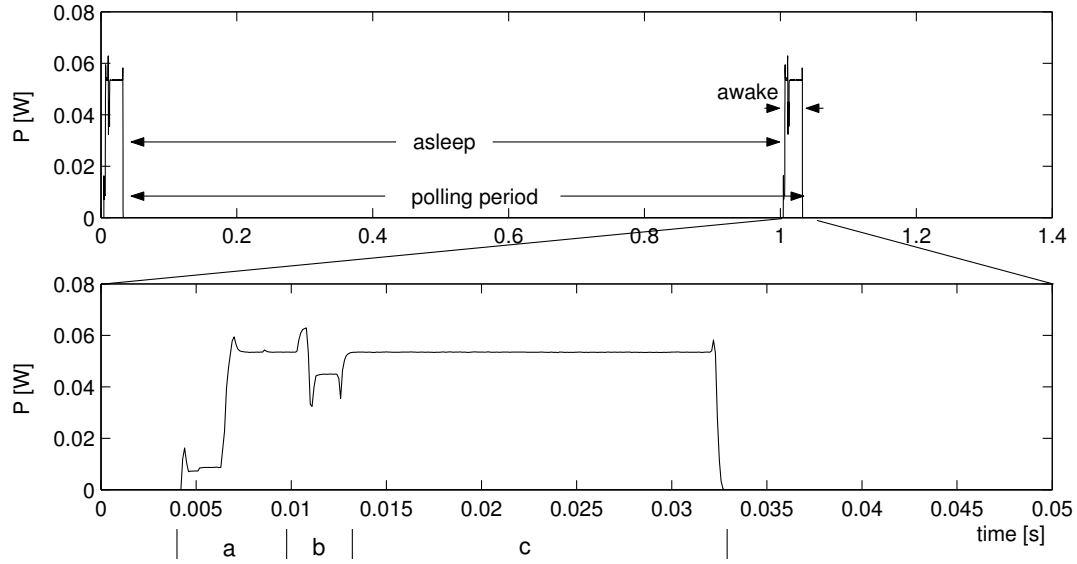


Figure 7.7: Measured Power Consumption While the Mote Polls the Gateway for Pending Messages. (a) The CPU wakes up and activates the radio. (b) The mote transmits the "ready-sleep" message and leaves its radio turned on for a short period. (c) If the gateway does not respond during the short period, the mote powers down again.

7.3.4 Lifetime Estimation

This section gives estimations for the lifetime of the battery in different use-cases. To be able to do this one has to make certain assumptions about the kind of batteries used and their characteristics.

Battery Characteristics

For this energy analysis I only consider the use of a pair of alkaline batteries [16] because of their low price and high availability. The capacity and the shape of the discharge curve are dependent on the current draw. In low-power applications the batteries deliver their full rated capacity and the discharge curve tends to be more linear than in higher-power applications.

In this case a linear voltage drop from 3V to 1.6V for the pair of batteries and a typical capacity of 2600mAh, or 9360As is assumed (see Figure 7.8). With a mean voltage of $2.3V$, the total energy is

$$E_{total} = U \cdot I \cdot T = 2.3V \cdot (9360As/T) \cdot T = 21500J.$$

The cut-off voltage of the Telos mote is determined by the CC2420 radio chip which requires 2.1V to operate; thus only a fraction of the energy can be utilized, in this case 71%. In Figure 7.8 the shaded area represents the usable fraction.

$$E_{avail} = 71\% \cdot E_{total} = 15300J$$

The manufacturer's claim that 85% of the initial capacity remains after four years leads to an estimated constant self discharge of 4% per year.

$$P_{self-discharge} = 4\% \cdot 21500J/(t_{sec-per-year}) = 27.8\mu W$$

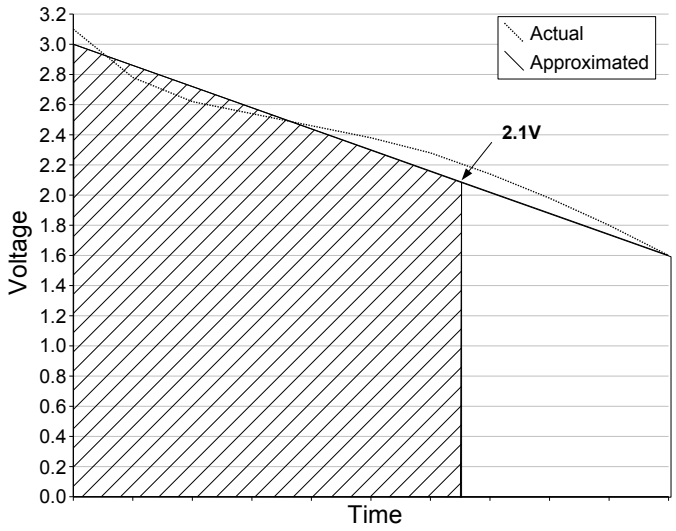


Figure 7.8: Battery Voltage Over Time, Assuming Constant Current Draw.

All energy figures and computations are based on a supply voltage of 2.6V which is close to the mean voltage in the batteries' life cycle, as they discharge from 3V to 2.1V.

Use Cases

When used as a web server, the latency of a user request is an important criterion which is influenced by the rate the device polls for incoming connections, as described above. Increasing the polling rate significantly increases the average energy consumption, and therefore decreases the battery lifetime. Figure 7.9 estimates the lifetime for different polling rates and shows the impact of HTTP transactions. Curves are shown for different

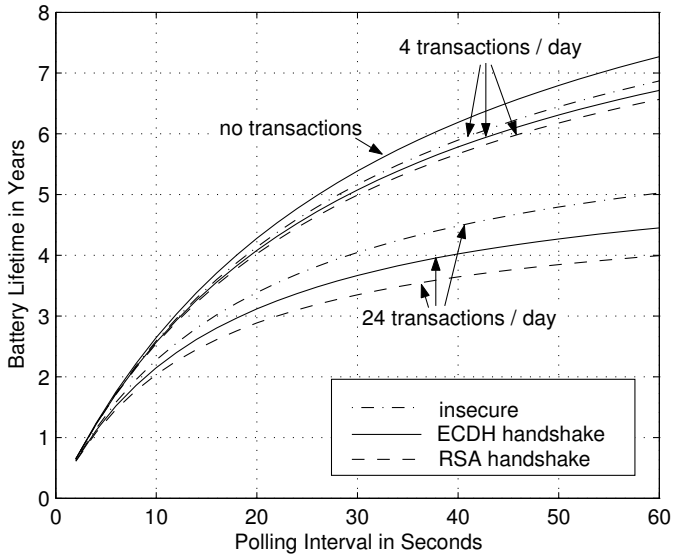


Figure 7.9: Battery Lifetime Estimation for the Web Server as a Function of the Polling Interval

numbers of average transactions per day using insecure and secure communication, with RSA and ECC handshake. I assume that one transaction consists of a handshake and the transmission of 10 HTML pages with a length of 500 bytes each. When keeping the polling interval in a user-acceptable range, the energy consumed by communication and security has almost no impact.

A second scenario tries to isolate the impact of security from the impact of idle listening. In this scenario the device does not poll for incoming connections but initiates transfers by itself. The device accumulates sensor readings in a buffer and transmits them whenever the buffer is full. A realistic size for the buffer is 4KB, given 10KB of RAM and the fact, that the Sizzle implementation already uses about 4KB of that. To illustrate the effect of the frequency of handshakes I compared the case where the device does a complete handshake for each transfer with the case where a new handshake is done only once a day. The results of these computations are presented in Figure 7.10.

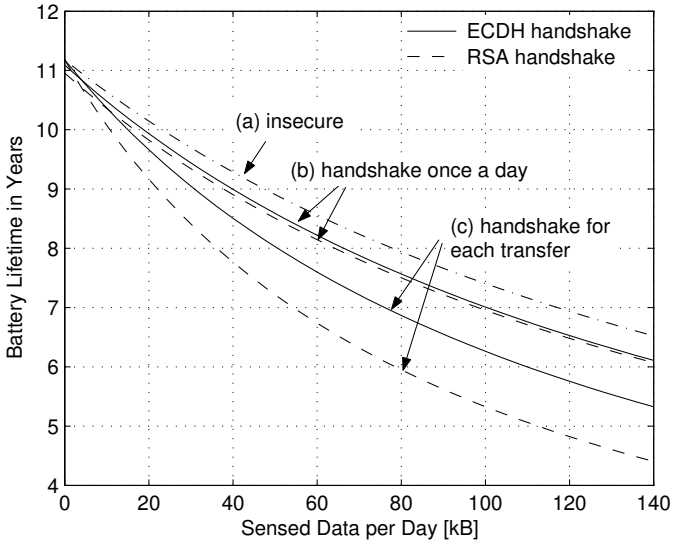


Figure 7.10: Battery Lifetime Estimation When the Device Initiates a Transfer Whenever a Buffer Is Filled With Sensor Readings. In case (a), the transactions are not secured, in case (b), a new SSL handshake is done once a day, and in case (c), a new handshake is done for each transfer.

7.4 Evaluation With Respect to the Requirements

In Section 5.1.3 a number of requirements for Sizzle have been presented. This section evaluates the final system with respect to these requirements.

- **Security.** Sizzle implements end-to-end security between the embedded web server and the client. Authentication of the server is done during the SSL handshake whereas the client authenticates itself to the server by providing username and password once the secure connection is established. Different users with different sets of capabilities can be managed.
- **Small Size.** The implementation is small enough to fit into any of the platforms under consideration. However the RAM usage is close to the 4KB limit on the Mica

platforms. While a simple demo application embedding Sizzle gets along with this amount of memory, more sophisticated applications will likely require more. The memory requirements on the Telos platform are rather relaxed and more than 6KB RAM remain available for the application. On this platform the available program memory is limited to 48KB, leaving only 8KB for extensions. Program memory usage can be reduced by enabling only one key-exchange mechanism, either ECC or RSA.

- **Standards Compliance.** The system contains a minimal but compliant implementation of SSL and HTTP and is able to interact with different standard web browsers. It has been tested with Mozilla, Internet Explorer, Safari, and Opera.

- **High Speed.** On the Telos mote the time required for a full ECC handshake is about one second and a web page is transferred in about 300 milliseconds. Thus the maximum delay the user experiences is 1.3 seconds and likely to be less with session reuse and persistent HTTP. These figures are better than expected and provide a good user experience. With the employment of the energy saving scheme, however, additional delays are added and a tradeoff between a fast response time and energy efficiency must be made.

- **Energy Efficiency.** While keeping the maximum response time of the system within a user-acceptable range of ten seconds, the mote can run for up to two years with a pair of standard alkaline batteries. Longer lifetimes can be achieved at the cost of longer response times.

- **Portability.** The code is modular and the same code-base is used for all different platforms. The SSL stack can be compiled to use the cryptographic code provided by OpenSSL and this way even run on Unix-like systems.

- **Reliability.** Sizzle was used extensively in demo sessions and proved to be reliable enough for demo use. Real-world applications will probably require several improvements or partial redesigns.

- **Convenient User Interface.** The interface to the wireless sensor network is simple and convenient. Through a single web page, provided by the gateway, the user can monitor the state of each individual sensor and have easy access to it.

- **Sample Applications.** Sample applications for demo purposes are available and show how to control a wireless actuator and how to access sensor readings through a web interface.

Conclusions

In this thesis I presented the design of a secure web server architecture for wireless sensors with several energy conserving features. I detailed the implementation of the HTTPS stack on the Telos platform and examined the energy consumption of the SSL handshake and the bulk data transfer, as well as the cryptographic primitives.

I created an efficient ECC implementation for the 16-bit Texas Instruments MSP430 microcontroller, able to perform a point multiplication in less than 500 milliseconds (assuming a clock frequency of 8 MHz). An ECC-based SSL handshake with a Telos device takes roughly one second, and once the connection is established, typical HTTPS requests take about 300 milliseconds.

To conserve energy I implemented a duty-cycle based scheme in which the web server polls for incoming service requests and remains in a low-power mode the rest of the time. With this approach the sensors can operate for more than one year from a pair of alkaline batteries under realistic scenarios, as opposed to three days when the radio transceiver remains in receive mode all the time. In the energy balance of the web server the polling for service requests still has the the greatest impact and leaves much room for improvements. However, hardware support for low-power listening is a prerequisite.

The measurements of the SSL protocol show that when using ECC rather than RSA, nearly 60% of the energy and 80% of the time can be saved. The energy cost of bulk data transfer over SSL is quite small (around 20 μJ per byte) and represents about 15% overhead compared to insecure data transfer. Overall support for session reuse and persistent HTTPS ensures that the introduction of SSL does not increase energy costs significantly.

It remains to be seen whether the idea of including of a secure web server in wireless sensor devices will catch on, or if other methods of interaction will become popular. Nevertheless, Sizzle demonstrates that public-key cryptography and even the implementation of a standard security protocol is feasible on very constrained 8-bit and 16-bit systems.

In the past much research effort was directed towards creating exotic security protocols for wireless sensor networks, such as random key predistribution, in the belief that public-key operations are too expensive. By demonstrating that public-key cryptography is not only feasible, but can also be efficient, Sizzle will have a great impact on future research in the area of wireless sensor security.

Appendix A

Abbreviations

ACK Acknowledgment

AES Advanced Encryption Standard

CPU Central Processing Unit

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

ECDLP Elliptic Curve Discrete Logarithm Problem

ECDSA Elliptic Curve Digital Signature Algorithm

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol, Secure

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

LAN Local Area Network

LR-WPAN Low-Rate Wireless Personal Area Network

MAC Can have the following meanings, depending on the context: Media Access Control (computer networks), Message Authentication Code (cryptography), Multiply-Accumulate (digital signal processing).

MD5 Message Digest Algorithm 5

NACK Negative Acknowledgment

NAF Non-Adjacent Form

PDA Personal Digital Assistant

RAM Random Access Memory

RC4 Rivest Cipher 4

RFID Radio-Frequency Identification

RISC Reduced Instruction Set Computer

RSA Rivest-Shamir-Adleman

SHA1 Secure Hash Algorithm 1

SPI Serial Peripheral Interface

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

WLAN Wireless Local Area Network

WPAN Wireless Personal Area Network

Bibliography

[1] C. Allen and T. Dierks. The TLS protocol — version 1.0. Internet proposed standard RFC 2246, Jan. 1999.

[2] Atmel Corporation. ATmega128 datasheet. `http://www.atmel.com/`.

[3] S. Avancha, J. Undercoffer, A. Joshi, and J. Pinkston. Security for wireless sensor networks. In *Wireless Sensor Networks*, pages 253–275. Kluwer Academic Publishers, 2004.

[4] H. Balakrishnam, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. In *IEEE/ACM Transactions on Networking*, 5(6):756–769, Dec. 1997.

[5] A. Boulis and M. B. Srivastava. Distributed low-overhead energy-efficient routing for sensory networks via topology management and path diversity. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pages 107–116, Mar. 2005.

[6] E. H. Callaway. *Wireless Sensor Networks: Architectures and Protocols.* Auerbach Publications, 2003.

[7] Certicom Research. SEC 2: Recommended elliptic curve domain parameters, Sept. 2000.

[8] H. Chan, A. Perrig, and D. Song. Key distribution techniques for sensor networks. In *Wireless Sensor Networks*, pages 277–303. Kluwer Academic Publishers, 2004.

[9] Crossbow Technology, Inc. Mica2, Mica2dot, and MicaZ datasheets. `http://www.xbow.com/`.

[10] Crossbow Technology, Inc. TelosB datasheet. `http://www.xbow.com/`.

[11] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer, 2002.

[12] C. Devine. C source code for AES, RC4, 3DES, SHA-1, SHA-2, MD5. `http://www.cr0.net:8040/code/crypto`. Visited on 2005-12-17.

[13] D. E. Eastlake and P. E. Jones. US secure hash algorithm 1 (SHA1). Internet informational RFC 3174, Sept. 2001.

[14] R. T. Fielding, H. F. Nielsen, and T. Berners-Lee. Internet draft: Hypertext transfer protocol - HTTP/1.1.

[15] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol — version 3.0. Internet Draft, Transport Layer Security Working Group, Nov. 1996.

[16] Gillette Company, The. Alkaline technical bulletin. `http://www.duracell.com/oem/Pdf/others/ATB-5.pdf`.

[17] J. Großschädl and E. Savaş. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 133–147. Springer Verlag, 2004.

[18] V. Gupta, S. Blake-Wilson, B. Möller, C. Hawk, and N. Bolyard. ECC cipher suites for TLS, May 2005.

[19] V. Gupta, M. Millard, S. Fung, Y. Zhu, N. Gura, H. Eberle, and S. C. Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded internet (best paper). In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pages 247–256, Mar. 2005.

[20] V. Gupta and M. Wurm. Configurable option to reduce the size of HTTP requests. In *Mozilla Bug Tracking System - Bug 300811*, 2005. `https://bugzilla.mozilla.org/show_bug.cgi?id=300811`.

[21] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.

[22] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[23] R. Housley, W. Ford, T. Polk, and D. Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Internet proposed standard RFC 3280, Apr. 2002.

[24] IEEE Computer Society. IEEE Std 802.15.4, 2003.

[25] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor networks. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 463–468, Apr. 2005.

[26] C. S. R. Murthy and B. Manoj. *Ad Hoc Wireless Networks: Architectures and Protocols*. Prentice Hall, 2004.

[27] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. In *Pervasive Computing*, 4(1):18–27, Mar. 2005.

[28] PeerSec Networks, Inc. MatrixSSL – Open source embedded SSL. `http://www.matrixssl.org/`. Visited on 2005-12-17.

[29] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 95–107, Nov. 2004.

[30] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 364–369, Apr. 2005.

[31] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Embedded Software — EMSOFT 2003*, volume 2855 of *Lecture Notes in Computer Science*, pages 306–322. Springer Verlag, 2003.

[32] R. Rivest. The MD5 message-digest algorithm. Request for Comments 1321, Network Working Group, Apr. 1992.

[33] R. Rivest. *The RC4 Encryption Algorithm.* RSA Data Security, Inc., Mar. 12, 1992. (Proprietary).

[34] D. Rogatkin. Miniature Java web server. `http://tjws.sourceforge.net/`. Visited on 2005-11-28.

[35] B. Schneier. *Applied Cryptography.* John Wiley & Sons, 1996.

[36] N. Smart. *Cryptography: An introduction.* McGraw-Hill, 2003.

[37] Texas Instruments. MSP430x1xx family user's guide. `http://www.ti.com/`. Visited on 2005-11-07.

[38] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org/`. Visited on 2005-12-17.

[39] TinyOS Community Forum. An open-source OS for the networked sensor regime. `http://www.tinyos.net/`. Visited on 2005-11-07.

[40] UC Berkeley WEBS Project. nesC: A programming language for deeply networked systems. `http://nescc.sourceforge.net/`. Visited on 2005-11-07.

[41] S. A. Vanstone. Next generation security for wireless: Elliptic curve cryptography. *Computers and Security*, 22(5):412–415, Aug. 2003.

[42] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce (EC '96)*, pages 29–40, Nov. 1996.

[43] W. Ye, J. S. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 3, pages 1567–1576, June 2002.

[44] F. Zhao and L. Guibas. *Wireless Sensor Networks: An Information Processing Approach.* Morgan Kaufmann, 2004.