Reykjavik University

NSN Project Final Report

TSense: Trusted Sensors and Support Infrastructure

Kristján Rúnarsson
kristjanr09@ru.is

Benedikt Kristinsson
benediktk09@ru.is

Supervisor:
Kristján Valur Jónsson
kristjanvj@ru.is

21.09.2010

Contents

Abstract

Distributed measurement and monitoring systems are vital to the functioning of many critical systems and are due to be ever more important as systems grow in size and complexity. This raises questions regarding the trustworthiness of the measurement processes, critical for making prudent manual or automatic control decisions.

In this work, we address one particular aspect of secure distributed measurement systems. We describe the design and construction of a trusted sensor prototype, a small tamper-proof device which gives cryptographic guarantees of data authenticity, and optionally, confidentiality. Further, we describe the design and construction of a comprehensive client/server based measurement system, using the trusted sensor to provide end-to-end trust guarantees.

Our prototype system was successfully implemented and tested on a small-scale test system of four separate network connected nodes.

Útdráttur

Dreifð nettengd mælikerfi eru stór þáttur í rekstri margra mikilvægra kerfa og munu væntanlega verða enn nauðsynlegri eftir því sem kerfi stækka og verða flóknari. Þessi þróun vekur spurningar varðand öryggi slíkra mæliferla og sérstaklega það traust sem hægt er að bera til þeirra. Traustar niðurstöður eru mikilvægur þáttur í því að hægt sé að taka réttar handvirkar eða sjálfvirkar stjórnákvarðanir.

Við skoðum einn þátt öruggra dreifðra mælineta í þessu verkefni. Við lýsum hönnun og smíði trausts skynara, sem er lítið tæki í óopnanlegri umlykju (e. tamper-proof) sem gefur kriptografíska tryggingu fyrir gagnatrausti og duld. Einnig lýsum við hönnun og smíði heildstæðs biðlara-miðlara mælinets sem notar trausta skynjarann til að gefa viðtakanda áreiðanlegar niðurstöður. Frumgerð okkar og stoðkerfi var smíðuð og sett upp á litlu prófunarkerfi með fjórum nettengdum tækjum.

Chapter 1

Introduction

Secure gathering and aggregation of measurements is a vital component in most modern networked systems, such as large computer networks, power grid monitoring and control, and industrial SCADA systems. The size and complexity of modern networked measurement systems raises questions regarding scalability, dependability and security. Security and trustworthiness of the collected data is in our opinion of particular interest in such systems, given that critical decisions, e.g. regarding resource allocation, failure response and customer billing are based on reliable information; bad information can easily render a system useless, or worse, cause damage due to improper management decisions.

In this work, we address one facet of measurement system security. We explore the concept of a *trusted sensor* – *tsensor* – a small self-contained device with limited communications capabilities. In essence, we explore the issues and challenges involved in securing measured data at the earliest possible point in a measurement system – at the sensors themselves. Our proof-of-concept prototype, *tsensor*, consists of a small embedded microcontroller, a simple sensor array[1] and basic serial communications. A production unit would be embedded in a permanently sealed and tamper-proof package, ensuring that manipulation of the unit, including extraction of cryptographic keys, would be infeasible. The tsensor is attached to a host system via an USB port, transforming that host, although inherently untrusted, into a trustworthy source of measurements for a networked monitoring system.

The goal of this project is to explore the issues involved in providing (within computational bounds) end-to-end integrity and confidentiality guarantees between sensor and sink (the recipient of the measurements). These issues include implementing and running cryptographic protocols on resource constrained hardware and exploring the minimal required support infrastructure for a complete working secure measurement system.

Security goals and adversarial modeling

We use a client/server model, in which clients are pairs of tsensors and untrusted end-systems of some sort and servers are measurement sink servers. Sink servers and tsensors are implicitly trusted: tsensors by the embedded cryptographic keys and assumed tamperproofness, and the sink servers by the assumption of direct control and hardening. Therefore, the clients, the general purpose end-user systems, are the sole corruptible node type in the system. We focus on *integrity*, and to a lesser extent *confidentiality* issues, in the classic CIA[2] trilogy of computer security. We neglect *availability* issues, since we focus on stealthy data modification attacks, rather than the

[1] The prototype uses simple resistance-based measurement devices, a NTC thermistor and photoresistor. A wide range of analog as well as digital one- and two-wire sensors are available and can be used for more sophisticated measurements.

[2] Confidentiality, Integrity and Availability.

generally "noisy" availability ones, such as denial-of-service attacks. For simplification, we will talk about a single adversary corrupting some fraction of the client population.

We consider insider attackers[3], that is, the end-systems corrupted by the adversary. Outsider attackers, both active and passive, are assumed to be excluded by the encrypted and authenticated tsensor/sink communications. We assume stealthy adversaries (Przydatek et al., 2003), that is, ones which seek to modify the aggregate computed by the sink in such a manner that they remain undetected for extended periods of time. The adversary can coordinate the actions of the clients under its control, that is, we assume colluding corrupted entities.

We assume the client, even if corrupted, will always communicate with a single sink, and further, its correct designated sink. That is, a corrupted client will never try to divert communications to an entity controlled by the adversary. The rationale behind this assumption is that the goal of the adversary in our model is to stealthily influence the aggregate measurement, not to "steal" the individual readings or disrupt the monitoring system itself.

To summarize, the adversarial goal is to corrupt some fraction of the clients such that it can stealthily influence the final aggregate of the collected measurements. The security requirements are that if a client delivers a result produced by a trusted sensor, the corresponding sink can ascertain (within computational bounds) that the results are authentic. Optionally, we guarantee (within computational bounds) that if readings are delivered to the sink, they are confidential with respect to any third party, sensors, clients or any intermediary nodes.

Contribution

The contributions of this work are the following:

- Multi-platform implementation of the industry standard AES encryption algorithm, executable on 32- and 64-bit Intel/AMD systems under Linux and OS X/BSD as well as on the 8-bit Atmel ATmega328 microprocessor. The block cipher building block was used to implement CBC-mode encryption and decryption, and CMAC authentication. A cryptographic library for symmetric encryption and authentication is part of our codebase for the TSense project.
- Development of a trusted sensor prototype, for use in a comprehensive client/server based secure measurement system with a trusted third party authentication service. The proof-of-concept system includes:
  - a trusted sensor prototype – *tsensor* – based on an Arduino Duemlianove[4] with an Atmel ATmega328 embedded microprocessor.
  - sink server software for collecting and processing measurements received from a number of tsensors.
  - authentication server software for strong identification of tsensors (the trusted third party).
  - a set of cryptographic protocols for authentication, key exchange and data transfer, tying all components together into a comprehensive measurement system.

All the tsense code, including sensor, supporting infrastructure and utilities, is open-source and maintained at `http://code.google.com/p/tsense`.

[3] Insider attackers are ones which are participants in the system. Insiders have knowledge of the communications protocols used and may attempt to modify them to obtain an advantage. Furthermore, insiders have access to a subset of the cryptographic key material in the system, at minimum their own private keys.

[4] `http://http://arduino.cc/en/Main/Hardware`

Chapter 2

Method

We took a systems design and implementation approach in our work, designing and implementing a working prototype system. Our plan of attack was formulated in accordance with the ultimate goal – a working prototype system after 3 months of work.

2.1 Phases and design decisions

The main project phases were

1. Phase: Design
   (a) Overall system design. Outline the concept and interaction of components.
   (b) Design of cryptographic protocols for authentication, re-keying and data transfer.
   (c) System components design
      i. sensor prototype software.
      ii. sink and authentication server software.
2. Phase: Implementation of cryptographic primitives
   (a) AES block cipher encryption and decryption
   (b) CBC-mode encryption and decryption
   (c) CMAC for authentication
3. Phase: Implementation and integration
   (a) Authentication, re-keying and transfer protocol implementation
   (b) tsensor software development
   (c) sink and authentication server software development
   (d) System integration
4. Phase: Unit and system tests on a test bed system

Phases 2 and 3 overlapped in part. Unit tests (part of phase 4) were done continuously over the development cycle of all components.

2.1.1 System design decisions

The main requirements for the TSense system were:

1. The system should be as simple as possible, while at the same time giving reasonable security guarantees, w.r.t. confidentiality and integrity.
2. The cryptographic primitives should be as strong as possible, within the limits posed by the resource constrained sensor hardware.
3. All cryptographic algorithms should be open and non-proprietary.
4. System integrity should be ensured. In particular, an adversary should not be able to insert cloned sensors or simulate multiple sensors.
5. The sensor should be as compact and cheap as possible.
6. The cryptographic software library for the sensor should be cross platform.

2.1.1.1 Simplicity

Client/server systems are the simplest networked systems to design and implement. We decided upon this configuration for our prototype system. The client in this respect is a sensor/client pair, while the server is the sink. We decided to use a separate authentication server to allow for system scalability by deploying multiple sink servers; a single authentication server maintains the requirement that the secret device keys are distributed amongst the fewest nodes possible.

2.1.1.2 Security

We initially considered public key cryptographic algorithms, which would have allowed us to simplify the system further by excluding the authentication server. In such a system, each tsensor would hold a public/private keypair and the public key could be safely distributed amongst all sinks. Factoring and discrete logarithm based public key algorithms require very large keys – 1024 bits at least for currently accepted security levels – rendering this approach impractical on the ATmega328 in our opinion. However, elliptic curve public key cryptosystems (Koblitz, 1987; Hankerson et al., 2004) require much shorter keys, on the order of the key lengths required for symmetric ciphers of comparable security.

In the end, we decided to base our implementation solely on a single symmetric cryptographic primitive – the AES block cipher. This approach saves the otherwise required coding and testing effort required for implementing an additional asymmetric algorithm. We therefore reserve the asymmetric cryptographic protocols for future work. However, note that the symmetric encryption algorithm is an important ingredient of asymmetric transport protocols, such as SSL; the much slower asymmetric algorithm is generally only used for symmetric key exchange, while the symmetric algorithm secures the data transfer itself. Thus, the work done on this symmetric algorithm would still be fundamental in any future work aimed at expanding this system to include an additional asymmetric algorithm.

2.1.1.3 Open cryptographic algorithms

Shannon's maxim "the enemy knows the system" is generally a prudent assumption. Kerckhoffs' law[1] states roughly the same principles, demonstrating the longevity of open cryptographic design. Briefly, these design principles assume that any adversary, internal or external, knows intimately all aspects of the system, except for cryptographic keys. In particular, this includes knowledge of the algorithms and protocols employed, which an adversary may attempt to manipulate to further its goals. Layered security is certainly prudent in most practical scenarios, whereas reliance on

[1] *La cryptographie militaire*, Journal des sciences militaires, vol. IX, pp. 5–38, 161–191, 1883

security by obscurity is usually regarded as poor practice. We assume open cryptographic protocols in our design, basing the security solely on the secrecy of the shared private keys. For this reason, we selected the Rijndael block cipher (FIPS, 2001), which was selected as the Advanced Encryption Standard (AES) in an open competition[2].

2.1.1.4 System integrity

The Sybil attack (Douceur, 2002) is a real threat to any distributed system. In this attack, an adversary simulates multiple colluding nodes on a single (or relatively few) highly capable platform. An adversary could easily simulate thousands of tsensors on a single high-end PC, unless measures are taken to prevent such an attack. Cloning and multiple insertion of a single sensor is a related, but less damaging, attack. Various solutions have been proposed for the Sybil attack, but the only absolute one is using strong and unforgeable node identities. This is the approach we take. Each tsensor has a strong and unforgeable (within computational bounds) identity in the form of a globally unique public ID and a 128-bit secret key, randomly generated for each device and shared only with the authentication service. The identification and authentication protocol, described in Section 2.5, ensures (within computational bounds) that no more than one instance of any tsensor can be active in the system at any given time. Since the trusted authentication server holds the private keys and identities of all sensors manufactured, we can claim that no adversary can simulate a non-existent sensor.

System integrity requirements also dictate that the secret sensor keys should exist on as few nodes as possible. Each key is unique and permanently "burned" onto the tsensor device. The symmetric nature of the authentication protocol requires the key to be present on at least one other node. Minimal distribution of the secret key is achieved by using a trusted third party, the authentication server.

2.1.1.5 Small and inexpensive sensor

We base our sensor device on a small microcontroller, the Atmel ATmega328, widely used in sensor nodes and embedded applications. Our development platform is an Arduino Duemilanove experimentation board with an ATmega328 in a 28-pin DIL package. The board is about 5x7 cm in size. The cost of this board is $29.95 (+ shipping and import costs) in quantities of one from `http://www.sparkfun.com` in June 2010. A smaller surface mount package for the ATmega is available, which means that a production PCB for a trusted sensor may be as small as 2x2 cm. The sensor itself and tamper-proof housing unavoidably adds to the size and cost, but a relatively small and inexpensive unit can certainly be realized in a production setting.

2.1.1.6 Cross-platform cryptographic library

Our cryptographic library was implemented and tested on all the currently supported target platforms, Arduino Duemilanovae with ATmega328 processor, as well as 32- and 64 bit Linux and OS-X/BSD operating systems on Intel/AMD hardware. We based our decisions in this respect on two factors:

- Single well-tested code base is obviously much easier to maintain.
- Interoperability issues in a heterogeneous distributed system are less likely to be an issue if all nodes run the exact same implementation.

2.2 The TSense system

The TSense system is shown in its simplest form in Figure 2.1. The module types are

[2] `http://csrc.nist.gov/archive/aes`

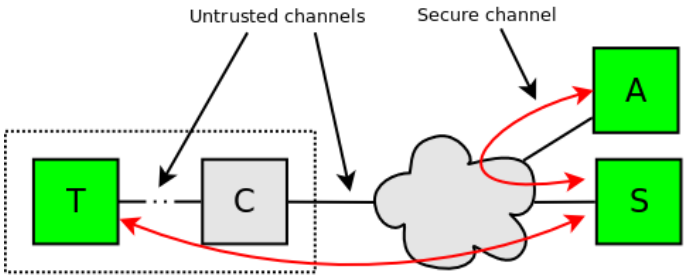


Figure 2.1: System overview. Interaction of the entities of the TSensor system. Green: Trusted entities. Gray: Untrusted entities (compromizable), Black channels: Untrusted, Red channels: Secure – end-to-end encrypted and authenticated.

T: The trusted sensor – *tsensor* – measures some array of quantities, e.g. temperature, pollutant particle count or luminosity, and publishes cryptographically authenticated results to a client (C). The published results may additionally be encrypted. tsensor would be a fully enclosed and tamper-proof unit in a production setting.

C: The client – *tsclient* – hosts the sensor, e.g. physically attached or USB-connected. The client is the corruptible (adversary controlled) entity in our model. A honest client will always forward measurements unmodified to the sink S, whereas a corrupted C may attempt modify the readings to influence the aggregate computation by S.

S: The sink server – *tssinkd* – collects measurements forwarded from a number of client nodes over standard TCP/IP. A network may include multiple networked sinks to achieve some degree of scalability. In such a system, each of the sinks is a *clusterhead* for a number of clients. In this work, we regard all sinks to be trusted.

A: The authentication server – *tsauthd* – is the trusted third party (see Section 2.5) in our system and handles authentication of sensor/client pairs at the time of insertion into a working measurement network. We regard the authentication server to be incorruptible for the purposes of this project. A single authentication server is assumed in the TSense system, which is a reasonable assumption from a scalability standpoint, since authentication takes place much less frequently than data collection.

Communications channels are as follows:

tsensor $\longleftrightarrow$ *tsclient*: The tsensor and tsclient communicate over a serial link, provided by an USB-serial emulator. The tsclient, and by extension both of its communications channels, are considered untrusted. The tsensor and tsclient communicate using a dedicated serial communications protocol.

tsclient $\longleftrightarrow$ *tsink*: The tsclient and tsink communicate over an IP network, using a set of dedicated protocols described in Section 2.5. Other transport layer protocols than TCP may be used; our decision to use standard socket communications was merely one of convenience. tsclient and tsink use a standard unencrypted TCP connection; all security is provided by tsensor/sink operations, rendering critical operations inaccessible and unmodifiable (within computational bounds) by the untrusted tsclient and any third parties.

tsink $\longleftrightarrow$ *tauth*: The tsink and tauth communicate over an IP network, using TCP/IP and Transport Layer Security (TLS). TLS provides strong mutual authentication, confidentiality and integrity for message transfer between these two critical system components. Both have installed X.509 certificates and hold each others public keys.

2.3 Cryptographic primitives

2.3.1 The block cipher – AES

A symmetric block or stream cipher is the workhorse of any confidentiality-preserving transfer protocol. Stream ciphers are generally faster than block ciphers, but harder to handle correctly. In particular, care must be taken to never re-use initialization vectors (IVs) when using stream ciphers. Examples of stream ciphers are e.g. found in the *eCRYPT* stream cipher portfolio[3]. Suitable secure block ciphers include the industry-standard AES (Rijndael) (FIPS, 2001; Daemen & Rijmen, 1999, 2000), Blowfish (Schneier & Whiting, 1997), Skipjack (NSA, 1998; Brickell et al., 1993) and TEA/XTEA (Wheeler & Needham, 1995; Needham & Wheeler, 1997)[4].

We selected the Rijndael algorithm (AES), with the shorter 128-bit key length, for our prototype. This is the weakest AES variant, but the relatively short key preserves valuable space on the *tsensor* device. It is however quite strong enough for our purposes, since no known attacks exist against AES, except for reduced round (reduced strength) variants. One of the design requirements of the Rijndael algorithm was that it should be implementable on small 8-bit devices with limited memory. The algorithm has been implemented successfully on small devices, such as the YubiKey[5] one time password generator.

We decided to write our own implementation of AES, rather than use an existing library. This choice was made in order to be able to support the various platforms in our system with the same code base. Our implementation of AES is close to the byte-oriented pseudocode published in the AES standard (FIPS, 2001), although tuned for performance. More efficient implementations of AES are feasible using word oriented algorithms, in essence pre-computing the AES round functions in four lookup tables and reducing the encryption and decryption processes to four table lookups and XORs (Daemen & Rijmen, 1999, Section 5.2). However, such implementations depend on rather large lookup tables, a drawback for memory constrained devices. Further, such algorithms are highly architecture dependent and therefore tricky to write in a multi-platform manner.

We will denote encryption $C = \mathcal{E}_K(P)$ with a key K, on a plaintext (message) P, resulting in ciphertext C. Conversely, we use $P = \mathcal{D}_K(C)$ for decryption with a key K. For symmetric encryption, we have that $P = \mathcal{D}_K(\mathcal{E}_K(P))$.

Our protocols are designed for *cipher independence*, meaning that any secure block cipher and any mode of operation (as discussed in the following section) could be used in the TSense system. A particularly logical choice is to use the full strength AES-256, in case a higher security margin is desired, especially for the permanent device key.

2.3.1.1 Mode of operation

The AES block cipher, as all other block ciphers, can be used in several *modes of operation* (Dworkin, 2001). The most basic one, Electronic Codebook (ECB), encrypts each block independently using the same shared key. This leads to potential information leakage as identical plaintext encrypts to identical ciphertext, which may allow adversaries to determine correlations and perhaps deduce information. Stronger modes of operation generally feed parts of the plaintext or ciphertext into the encryption/decryption process for subsequent blocks, resulting in unpredictable variations, even for identical plaintext blocks. We used the Cipher Block Chaining (CBC) mode of operation in our implementation (Dworkin, 2001). The CBC mode is widely used, for example in the TinySec transport layer protocol (Karlof et al., 2004), in conjunction with the Skipjack cipher.

The CBC-mode of operation can be used with any block cipher. For plaintext blocks $P_1, P_2, \ldots, P_n$ and the corresponding cipertext blocks $C_1, C_2, \ldots, C_n$, we have the encryption and decryption

[3] `http://www.ecrypt.eu.org/stream`

[4] See `http://www.users.zetnet.co.uk/hopwood/crypto/scan/cs.html` for a more comprehensive list of suitable block- and stream ciphers.

[5] `http://www.yubico.com/products/yubikey`

operations

$$C_i = \mathcal{E}_K(P_i \oplus C_{i-1})$$

and

$$P_i = \mathcal{D}_K(C_i) \oplus C_{i-1}$$

where $C_0 = IV$, an *initialization vector*.

2.3.1.2 The initialization vector

The initialization vector (IV) must be known to both the encrypting and decrypting parties and should be replaced, at least periodically. Varying the IV ensures variation in cipertexts encrypted from identical plaintexts under the same key. The IV is not required to be secret, and can be sent in plaintext along with the message, but its integrity must be preserved, for example by encryption (secret IV) (Menezes et al., 1996, pp. 230). The IV must be unpredictable for use in CBC-mode (Dworkin, 2001). One suitable method is to encrypt a nonce[6] under the same key as used for the encryption of the message plaintext.

The nonces used in our protocol to prevent replays are generally secret; they also serve the purpose of providing variability in the produced ciphertext. Using these nonces as IV source is therefore not a satisfactory solution, as decryption would obviously be impossible. We propose the following compact method for IV generation. We choose a two byte random number, the IV nonce, which is included (in plaintext) in the standard message header (after the message ID). The IV nonce is encrypted (or MACed), suitably padded, using a key derived from the encryption key being used to encrypt the present message. The 128-bit result is used as the IV for the message encryption and decryption. Using a short random nonce saves (potentially) on messaging costs, but the trade-off is the additional single key derivation and block encryption at both ends for each message exchange. This method is in our opinion secure enough for our purpose, while being much simpler than that presented by Karlof et al. (2004) in their TinySec protocol. Our present proof-of-concept implementation uses a statically assigned IV for all operations, but future versions will use the method proposed above.

2.3.1.3 Padding

The smallest unit that a block cipher operates on is a single block. The block size in AES is 16 bytes, hence the plaintext needs to be padded if it is not divisible by 16. That is, given the data string x of length n_x, a padded string x' is produced if $16 \nmid n_x$ such that $16 \mid n_{x'}$. Otherwise, x is used as-is. We use zero padding, that is all the bytes required to be padded are padded with zero (Menezes et al., 1996, Algorithm 9.29). The lengths of the encrypted sections in the protocol messages are either known beforehand or the length is sent along in the message, which removes the need of unambiguous padding.

There are other possible padding schemes. One applicable scheme is to pad the data string x with a single 1 bit, followed by 0 bits until the last block is filled (Menezes et al., 1996, Algorithm 9.30), (Dworkin, 2001, Appendix A). Another method is to pad the string x with m-bytes, where m denotes the number of bytes needed to pad the string to produce x' (Kaliski, 1998).

2.3.2 Message Authentication Code (MAC) algorithm

Algorithms for generating *Message Authentication Codes (MACs)* are an important counterpart to symmetric ciphers. A MAC is an authentication tag which can be applied to a message, allowing the recipient to verify its authenticity in a cryptographically secure manner. Of course, both the sender and receiver must hold a shared key. The MAC tag is analogous to digital signatures, generated by public key signature algorithms. Common MAC algorithms include HMAC (Bellare

[6] Number used ONCE

et al., 1996), which uses using cryptographically secure hash functions as its building block, and CBC-MAC (Bellare et al., 1999) and CMAC (Dworkin, 2005), which are based on block ciphers. We chose CMAC (Cipher-based MAC), the stronger of the two, as our MAC algorithm and follow the guidelines of Song, Poovendran, Lee, and Iwata (2006) and Song, Poovendran, and Lee (2006). Choosing a block cipher based MAC algorithm means that we do not have to implement another primitive, such as a cryptographically secure hash function.

We denote MAC construction as $T = \mathcal{T}_K(M)$, for a key K. T is a tag (MAC) of the message M. Note that M can be either plaintext or ciphertext. MAC verification is a corresponding operation, in essence comparing the received MAC and message with an independently constructed MAC. That is, for a transmitted message $(M \parallel T)$, accept the received message $(M' \parallel T')$ as authentic if $\mathcal{T}_K(M') = T'$.

Note that any secure MAC algorithm and authenticating encryption mode, as described in the following section, can be used in the TSense system.

2.3.2.1 Authenticating encryption

Authenticating encryption algorithms provide confidentiality and authenticity guarantees in a single primitive. In terms of block ciphers, we have several proposed authenticating modes of encryption, some examples being Counter with CBC-MAC (CCM) (Dworkin, 2004), Offset Codebook (OCB) (Rogaway et al., 2003) and Galois Counter Mode (GCM) (Dworkin, 2007). Authenticating stream ciphers are discussed by Teo et al. (2009). In the TSense project, we use the conceptually simpler composition of generic encryption and MAC algorithms, as described by Bellare and Namprempre (2007).

According to Bellare and Namprempre, generic encryption and MAC primitives can be composed in the following manner:

E&M – Encrypt-and-MAC: Encrypt and MAC the plaintext separately and combine:
$\bar{\mathcal{E}}(K_e \parallel K_m, P) = \mathcal{E}_{K_e}(P) \parallel \mathcal{T}_{K_m}(P)$.

MtE – MAC-then-encrypt: MAC the message, append to the plaintext and then encrypt:
$\bar{\mathcal{E}}(K_e \parallel K_m, P) = \mathcal{E}_{K_e}(P \parallel \mathcal{T}_{K_m}(P))$.

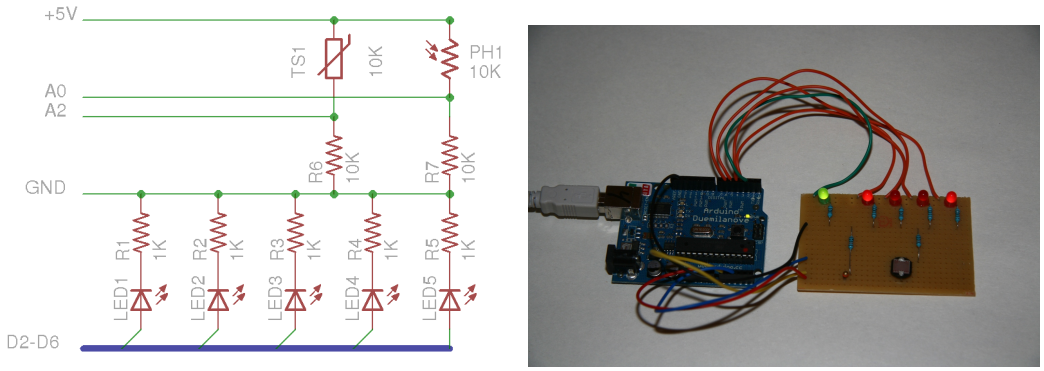
EtM – Encrypt-then-MAC: Encrypt the message, MAC the ciphertext and append:
$\bar{\mathcal{E}}(K_e \parallel K_m, P) = C \parallel \mathcal{T}_{K_m}(C)$, where $C = \mathcal{E}_{K_e}(P)$.

Here, $\bar{\mathcal{E}}(K_e \parallel K_m, P)$ denotes authenticating encryption of a message P using shared (symmetric) keys $K_e \parallel K_m$, where K_e is the encryption key and K_m is the MAC key. Two separate keys are required, as re-using a key for two different purposes is considered a bad practice.

All three composition paradigms are strong enough for our purposes, since we use a block cipher and MAC for which no known (efficient) attacks exist. However, EtM is considered by Bellare and Namprempre to be the strongest of the three, and was therefore chosen as the composition paradigm for our cryptographic protocols.

2.4 System components

Our system consists of the four node types, previously introduced in Section 2.2. We will now proceed to describe each module type independently in more details. Refer to the source code and documentation, available at the project site `http://code.google.com/p/tsense`, for further details.



(a) (b)

Figure 2.2: The tsensor prototype. (a) Schematic diagram of the sensor board. (b) Picture of the Arduino Dumilanovae experimentation platform and assembled sensor board.

2.4.1 *tsensor* – Trusted sensor module

The tsensor gives us the ability to "bootstrap" trust in a networked measurement system, in which intermediary nodes, including the clients hosting the sensors, are untrusted. In order to provide this guarantee of trust, the tsensor cryptographically tags all readings with a message authentication code (MAC), providing receiver-end (sink) verifiability. The tsensor is intended to be a small, tamper-proof device, guaranteeing that this trusted module or its produced results cannot be tampered with – the physical protection[7] prevents unauthorized access to the device internals, while the cryptographic protocols ensure (within computational bounds) that no adversary can modify or intercept messages.

The requirements for the tsensor are

1. to measure some quantity and deliver securely to a trusted sink (collection server).
2. to serve as a strong, unique and unforgeable identity for the platform to which it is attached (client).
3. to be as small and cheap as possible.

Our choice of hardware for the tsensor prototype was in the spirit of these requirements. We choose the Atmel[8] ATmega series of CPUs, specifically an ATmega328 (Atmel, 2010). The ATmega328 is a RISC-based microprocessor, intended for embedded systems, capable of running at up to 20 MHz. It has 32K of on-board Flash program memory, 2K of RAM and 1K of EEPROM. Several digital I/O and analog inputs are provided. ATmega-series CPUs are widely used in small devices, for example in wireless sensor nodes (Akyildiz et al., 2002).

We choose the Arduino Duemilanove[9] prototyping board as our development platform in this project. The Duemilanove comes with an ATmega328 in a 28-pin DIL package and includes all the peripherals necessary for running the CPU. Additionally, a USB-to-serial converter is provided, allowing the board to be connected directly to a host computer for programming and data delivery. The Duemilanove was augmented with a very simple sensor interface board, consisting of a couple of sensors[10] (NTC thermistor and a photoresistor) and LEDs. The setup is shown in Figure 2.2.

[7] Our prototype is presently anything but tamper-proof, as can readily be seen from Figure 2.2. However, we assume a production model would be very well protected against physical tampering. Tamper-proof or tamper-resistant embedded systems are presently being produced by e.g. Freescale `http://www.freescale.com`.

[8] `http://www.atmel.com`

[9] `http://arduino.cc/en/Main/ArduinoBoardDuemilanove`

[10] Our simple sensor array is strictly intended to show a proof-of-concept. A much wider array of sophisticated analog, as well as one- and two-wire digital, sensors is available, which can be directly interfaced with the ATmega for more sophisticated measurement applications.

2.4.1.1 Software development

The ATmega-series CPUs can be programmed in assembly language or using a C++ variant and the avr-gcc/avr-g++ compilers. The Arduino prototyping system includes an IDE for C/C++ development, with one button compile and upload capabilities, which we used for the tsensor software development.

The 32K of program memory is rather restricted by current computing standards, but still allows development of fairly sophisticated programs. As stated previously, we chose an byte-oriented symmetric block cipher for all operations, rather than trying the limits of the hardware with a more computationally intensive asymmetric algorithm and digital signatures. We also implemented only one base cryptographic primitive – the AES block cipher – to conserve program memory space.

The 2K of RAM on the ATmega328 proved to be a more severe restriction. The RAM is needed to hold a small amount of operating system state, as well as the program heap and stack space. Our earliest tsensor prototypes were plagued by hard to diagnose spurious crashes, seemingly due to program memory corruption. The reason for these crashes was that allocating the AES lookup tables (s-box, inverse s-box and round-constant lookup tables), a total of 528 bytes, in RAM caused heap and stack memory collisions. We then moved all statically assigned data to the 1K EEPROM, which solved this particular problem.

We designed the software for maximum re-usability on the various system components. Concretely, the same implementation of AES in CBC mode and CMAC compiles for the ATmega328 on the tsensor and for 32- and 64-bit Intel/AMD platforms for the sink- and authentication server under Linux and OS-X/BSD.

2.4.1.2 Device identification data

Each tsensor holds unique identifying information in its EEPROM. In a production device, this data would be unmodifiably "burned" into a ROM or a similar permanent storage device. The device data is as follows:

Public ID: A 6-byte globally unique device identifier, consisting of a 2-byte manufacturer ID and a 4-byte device serial number. Note that the public ID can easily be expanded to any size in a production setting. The public device ID can, as the name implies, be freely disclosed to anyone. A command message issued by a client node over the serial line is used to retrieve this information.

Private ID: The private device ID is a 128-bit value, assigned by the manufacturer on assembly. The private ID is used as the initial encryption key K_{AT} in the authentication protocol, as further described in Section 2.5. The private ID is assumed to be completely inaccessible on the tsensor device, although the symmetric nature of the system requires a copy to be kept by the trusted authentication service. The authentication component is further described in Section 2.4.4. The private ID is a 16-byte long random number, generated for each sensor.

Manufacturer information: (optional)

- Manufacturer name and address
- device serial number
- device manufacture date
- software version

2.4.1.3 Running the tsensor

The tsensor receives its power through the USB-connection and is on while it is plugged in. It starts its operation in a standby mode that does not collect or provide any data. No data is

provided by the sensor until the session and encryption keys have been successfully delivered, as described in Section 2.5. The authentication process is bootstrapped by the client (hosting node) by requesting and forwarding the authenticated public device identification package, as further described in Section 2.5.1.

External interface The tsensor interface board has a total of five LEDs. A green status LED blinks for standby and initialization modes and glows steadily in fully initialized execution mode. Four red LEDs are used to indicate intermediary authentication protocol states, data transfer operations and error states. The four red LEDs are not strictly necessary and may be omitted from a production sensor.

Serial API The sensor is controlled interactively through a set of control messages, which are issued by the client over the serial (USB) connection. This set of messages includes commands to set the sampling interval, sample buffer size and current time. A vital security requirement of the tsensor serial API is that secret keys (permanent per-device private id, session or encryption keys) should ever be accessible, except indirectly as they are used to encrypt and authenticate communications.

The most important feature of the serial API protocol is the device id request and response messages, which bootstrap the sensor into a working measurement system, as described in Section 2.5.1.

2.4.1.4 tsensor utilities

A set of utilities was developed to configure and diagnose the tsensor.

tsconfig is a utility for configuring a tsensor. The script is executed from the command line with a specified device public ID and "burns" the device EEPROM. This includes setting up the AES lookup tables, manufacturer information, public device ID and generating a fresh 128-bit random private key. The utility *generatekey* is used for the key generation and uses the Unix system random source `/dev/random` to generate random keys. *tsconfig* and *generatekey* are both part of the TSense code.

tsdiag extracts diagnostic information from a USB-connected tsensor device. All information accessible using the serial API is displayed. In the prototype, the entire EEPROM can be dumped, which breaks the basic security premise of the device – the secret key is completely exposed. However, this feature is for development purposes only, and would be removed for a production device.

2.4.2 *tsclient* – TSense client-side software

The *tsclient* is a small and completely untrusted software component installed on an equally untrusted platform. The sole purpose of this software is to interface with the USB-connected tsensor and forward its messages (unmolested) to the sink server. The goal of this project is in essence to force this untrusted piece of software and the platform it resides on to provide only trustworthy information to the sink. We regard the attached tsensor to be strong enough to uniquely identify the client – not as trusted but guarantees (within computational bounds) that the Sybil attack (Douceur, 2002) is impossible in this system. In this sense, the tsensor functions similar to a SmartCard authentication system.

tsclient is written in python (v. 2.6), using the serial and socket libraries. Using a scripting language like Python for the client software is within the spirit of the project: the program is in plaintext on the client computer, allowing anyone to analyze and even modify it. No user written

enhancements (or malicious tampering) should reduce the security guarantees of the overall system. We will attempt to support this claim in Section 3.5.

2.4.2.1 Running tsclient

tsclient is a python script, executed on the command line. See the source code documentation for details on command line switches and settings. The user needs the connection information for the attached tsensor (USB connection string and baud rate), as well as for the sink server (IP address and port). On startup, the tsclient requests an partially encrypted and MACed identification package (see Section 2.5.1) which it then forwards (unmodified) to the designated sink server. This initial exchange bootstraps the authentication protocol and the insertion of the sensor/client pairing into a measurement system. After the initial device ID request, the tsclient is simply a proxy, forwarding encrypted messages unmodified back and forth between tsensor and sink. tsclient supports only a single tsensor and a single sink in the current version. Future enhancements include supporting multiple sensors and multiple sinks per sensor for increased robustness.

2.4.3 *tssinkd* – The TSense sink daemon

tssinkd – The TSense sink daemon collects authenticated measurements from one or more tsensor client/sensor systems. In addition, it participates in the authentication and re-keying protocols, as is further described in Section 2.5. For this purpose it uses the same AES encryption library developed for tsensor (see Sections 2.3 and 2.4.1.1) for encrypted communication with the tsensor. The TSense sink server is a traditional Unix daemon written entirely in C++. It uses old school forking to service incoming requests on a child-per-request basis. Because of this the sink daemon needs a persistence layer to store state information. This state information consists of a sensor profile which contains the current session key of the sensor in question, an associated random key and the sensor's unique ID. Using these two keys the server can the generate the key schedules it needs to handle an incoming request. In addition to using a persistence layer to store sensor profiles the sink server also requires a persistence layer to store measurement data collected from the various tsensor units it services.

The sink daemon software, it's persistence layer and the systems on which it resides are considered trusted for the purposes of this project. During implementation some corers were cut. As has already been mentioned, communication with tsensor via the untrusted client is considered secure. Communication with the TSense authorization daemon (tsauthd) is, however, conducted over an industrial strength TLS tunnel (OpenSSL) with mutual authentication via x509 certificates. The persistence layer chosen, for the sake of speedy development, was MySQL since it is ubiquitous and well documented. Neither the persistence layer nor the database it self were secured or encrypted, but for the purposes of this proof of concept system they are assumed to be secure. In a production environment, the sink daemon would require a secure connection to an encrypted database, and the sink daemon software and the machine it resides on would have to be located in a secure facility. Finally, it would be advisable to switch from the MySQL library to a proper C++ based database abstraction layer. This should be fairly easy, since care was taken to properly separate the sink server code from the database code, so the transition to a new database library should be smooth.

2.4.4 *tsauthd* – TSense authentication service

tsauthd – The TSense authentication daemon handles the integration of sensor/client pairs into a working measurement system. The authorization sever holds the private keys (private IDs) of every tsensor unit. Each tsensor must contact the authorization server via the untrusted client and the sink server to obtain a session key before the sensor can start providing measurement data. Much like sink server, the authorization server is a traditional Unix daemon written entirely in C++ and uses old-school forking to handle requests on a child-per-request basis. All communication

between the tsensor units and the authorization server is secure. The authorization server uses the AES library developed for tsensor (see Sections 2.3 and 2.4.1.1) to communicate with tsensor but uses OpenSSL/TLS to communicate with the sink server. The authorization server uses the same persistence layer as the sink server to store device profiles consisting of the private key and unique ID of every tsensor in the system. The same caveats regarding the security of the persistence layer apply to the authorization server as did in the case of sink server.

As is the case with the sink server, the machine on which the authorization server resides and the associated persistence layer, are considered to be highly trusted and hardened for purposes of this proof-of-concept project. In a production environment, tsauthd would have to be deployed on a secure machine, connected to an encrypted database over a secure connection and would have to be installed in a high security facility.

2.5 Cryptographic protocols

Our protocol consists of four parts:

1. the authentication of the *tsensor* device, and by extension, unique identification of the hosting client.
2. the session key exchange between *tsensor* and sink.
3. periodic re-keying of the *tsensor* to install, and maintain freshness of, transport encryption and authentication keys.
4. data transfer of encrypted and authenticated data between tsensor and sink.

Messages are encrypted with our AES-128 library and CMAC of ciphertexts is appended to ensure data integrity. The aim of the design is to make the protocol as secure as possible, while still maintaining a small payload.

2.5.1 Authentication and key exchange protocols

The authentication protocol is a symmetric protocol, using a trusted third party, based on the Needham-Schroeder authentication and key exchange protocol (Needham & Schroeder, 1978). A modified Needham-Schroeder is also used in the Kerberos protocol (Neuman & Ts'o, 1994). The main modification in our protocol is that sensor/client pairs do not communicate directly with the third party, but through (trusted) sinks. Further, the authentication process is bootstrapped by an untrusted entity, that is, clients hosting trusted sensors. Restricting access to the authentication service to solely the trusted sinks adds an extra layer of protection, as only the trusted sinks need direct knowledge of its address, in essence a bit of security-by-obscurity for A. The original Needham-Schroeder was vulnerable to replay attacks (Denning & Sacco, 1981), which can be fixed by using nonces and/or timestamps (Needham & Schroeder, 1987). We use primarily nonces in our protocols to counter replay attacks. Using timers to guarantee freshness is problematic in our case, as the sensor relies on the untrusted client to set its time. Instead, we use randomly initialized monotonically increasing counters as nonce sources.

As discussed in Section 2.3.2.1, we use the *Encrypt-then-MAC (EtM)* (Bellare & Namprempre, 2007) composition paradigm, in which the plaintext is first encrypted and then a MAC of the ciphertext appended. We use $\mathcal{T}_K$ in our protocol to indicate *tagging* (MACing) of the message ciphertext, using a shared symmetric key K. For brevity, we denote the encryption with a key K_{XY}, that is a key shared pairwise between entities X and Y, as $\mathcal{E}_{XY}(\bullet)$, instead of the more accurate form $\mathcal{E}_{K_{XY}}(\bullet)$. Similarly, we denote decryption with the key K_{XY} as $\mathcal{D}_{XY}(\bullet)$, and tagging of the entire encrypted message simply as $\mathcal{T}_{XY,e}$. We emphasize that the same key is never used for both encryption and tagging of a message.

2.5.1.1 Authentication

The authentication and session key exchange proceeds as follows:

$$C \rightarrow T : (queryId)$$

The client C (untrusted) queries the sensor T for its public ID.

$$T \rightarrow C : (idresponse, T, \mathcal{E}_{AT}(T, N_T) \parallel \mathcal{T}_{AT,a})$$

The sensor T returns its public ID (denoted by T above), along with an encryption of the ID and a nonce N_T, using the permanent device key K_{AT}. A CMAC, constructed using a derived authentication key $K_{AT,a}$ (see Section 2.5.5 on key derivation) of the message is appended to ensure authenticity. The key pair K_{AT} and $K_{AT,a}$ is shared between sensor T and authentication service A. The purpose of the nonce N_T is to provide freshness guarantees and prevent replay attacks. It is only sent in encrypted form in this construction.

$$C \rightarrow S : (idresponse, T, \mathcal{E}_{AT}(T, N_T) \parallel \mathcal{T}_{AT,a})$$

C passes the public id of T and the encrypted packet to S, unmodified since it does not hold the keys to decrypt it. In fact, neither does S. The encryption key is, as indicated by the subscript, only shared between the sensor and authentication service.

$$S \rightarrow A : (\mathcal{E}_{AS}(idresponse, T, \{\mathcal{E}_{AT}(T, N_T) \parallel \mathcal{T}_{AT,a}\}) \parallel \mathcal{T}_{AS,a})$$

S forwards the unmodified message from T to A, which will then read the encrypted packet, check the authenticity of it and respond accordingly. S forwards the packet unmodified, but adds extra security by encrypting by the key K_{AS}, which the sink S shares with the authentication service A, and the derived key $K_{AS,a}$. In our implementation, we assume the keys K_{AS} and $K_{AS,a}$ are negotiated implicitly by the TLS mechanism.

2.5.1.2 Session key exchange

If T is accepted, then A generates a random 128-bit session key K_{ST} and sends it to S, along with the re-keying (renewal) interval rt_{ST} and the original nonce N_T. As before, the subscript ST indicates that the session key is pairwise shared between a sensor T and sink S.

$$A \rightarrow S : (\mathcal{E}_{AS}(keytosink, T, K_{ST}, rt_{ST}, \{\mathcal{E}_{AT}(N_T, K_{ST}, rt_{ST}) \parallel \mathcal{T}_{TA,a}\} \parallel \mathcal{T}_{AS,a})$$

A constructs an encrypted packet, containing the original nonce N_T and the session key, which S must forward unmodified to T (through C). The nonce allows T to ascertain that it was in fact A that generated the session key – it is highly improbable that any other party, including the otherwise trusted sink, can generate this package correctly. This provides bilateral implicit key authentication (Menezes et al., 1996, pp. 498) between sensor and authentication service. In essence, both nodes use correct (according to some rule) encryption as proof of identity, that is, the communicating partner holds the corresponding secret key.

rt_{ST} is a 2-byte integer, which specifies how often the session key may be used before renewal. Relatively short duration encryption keys help to make cryptanalytic attacks infeasible.

If T is not validated, the the authentication service will respond with `MSG_T_ERROR` (not yet implemented).

$$S \rightarrow C \rightarrow T : (keytosense, \mathcal{E}_{AT}(N_T, K_{ST}, rt_{ST}) \parallel \mathcal{T}_{AT,a})$$

S forwards the (unmodified) encrypted package, containing the nonce N_T and session key to T through C. T opens the package, by decrypting with K_{AT}, and retrieves K_{ST}.

The session key has now been delivered to T and S.

2.5.1.3 Initial handshake and transport key delivery

The session key K_{ST} has now been delivered to T and S. This key is only used for re-keying, that is, delivery of transport encryption and authentication keys which are used for the data transfer.

The Needham-Schroeder protocol includes a handshake step in which the entities receiving the session key make sure they both have the correct key installed. We also use the handshake for the initial exchange of the encryption and authentication keys for the data transfer protocol – the most frequently used keys in our protocol.

The handshake must take place right after the delivery of K_{ST} to T. The transport key exchange is performed repeatedly during the session, as dictated by the specified re-keying interval, to refresh the keys.

$$T \rightarrow C \rightarrow S : (rekey, T, \mathcal{E}_{ST}(T, N_T) \parallel \mathcal{T}_{ST,a})$$

Here, a sensor T uses the session key K_{ST} to request a fresh set of transport keys from the sink S. A freshly generated nonce N_T provides the necessary freshness guarantees and bilateral implicit key authentication.

$$S \rightarrow T : (newkey, T, \{\mathcal{E}_{ST}(T, N_T, R, rt_{ST}) \parallel \mathcal{T}_{STe,a}\})$$

S returns the new *key material*, a fresh 16-byte random number R to T. The random number is used to derive the data transfer keys K_{STe} and $K_{STe,a}$, as described in Section 2.5.5. rt_{ST} is the key renewal timer, indicating how often the keys derived from R can be used before next re-keying. The encryption key K_{STe} is derived from R, which provides bilateral implicit key authentication between T and S upon subsequent data encryption and decryption operations.

In the first round after session key delivery, this exchange serves as a handshake to ensure both T and the respective S hold the same session key K_{ST}. The message code `MSG_T_REKEY_RESPONSE` is used for regular re-keying requests and `MSG_T_REKEY_HANDSHAKE` is used for the first-round handshake.

2.5.2 Data transfer protocol

The data transfer protocol uses the encryption key K_{STe} and the derived authentication key $K_{STe,a}$, both derived from the key material delivered in the re-keying step (see Section 2.5.5 on key derivation). The current protocol is push-based (the sensor periodically publishes data), although it can be trivially modified to allow for pull-based (polling) operation.

$$T \rightarrow C \rightarrow S : (data, T, length_{\mathcal{E}}, \mathcal{E}_{STe}(T, t_T, D) \parallel \mathcal{T}_{STe,a})$$

T encrypts the measurement data D, MACs it and delivers to C, who will forward it to S. T is the public sensor ID and t_T is the four byte (Unix) timestamp of the first sample in the data payload. Encrypting the sensor public ID provides the receiving sink with the means of authenticating the transmitting sensor (implicit key-based authentication).

The measurement data D in on the following form:

$$[length_D \parallel m_1 \parallel m_2 \parallel \ldots \parallel m_n]$$

where $length_D$ is the total length in bytes and $m_i \in \mathbb{Z}$ is a measurement record. In our current protocol, we limit the size of each measurement to a single byte. Any number of bytes per measurement can be used without modifying the protocol.

The transport keys K_{STe} and $K_{STe,a}$ must be periodically refreshed, as dictated by the re-keying interval, specified on delivery by the sink.

2.5.3 Finish message

A well-behaved client should send a FINISH message to its attached sensor, as well as the currently associated sink, when it wishes to terminate a session. This allows all devices to properly clean up their internal state. However, clients may terminate ongoing sessions without any notice, for example due to crashes or link failures. It is therefore important that all nodes maintain a soft state – clean up their internal state periodically to eliminate unresponsive nodes.

Our current protocol includes a soft state for all intermediary protocol stages in the tsensor, as well as finish message handling. Further robustness issues, such as soft state cleanup on the sink nodes, is reserved for future work.

2.5.4 Handling the unexpected

The protocols outlined above assume well-behaved participants, which is not necessarily the case. Timeouts, malfunctions, message losses, malicious actions and program bugs are all factors to be considered when designing robust protocols. We reserve most of the robustness features for future work; components generally fail, hopefully gracefully, in the current version when unexpected events occur. Further protocol robustness is reserved for future work.

2.5.5 Keys and key derivation

We observe the principle that a cryptographic key should never be used for more than one purpose. For example, the same key is never used to encrypt and authenticate a message. We therefore always assume the existence of a key pair (K_e, K_m) for encryption and authentication. We have a number of such different key pairs – private sensor ID (permanent master key), per-session generated session keys and limited lifetime encryption keys. For each of the encryption keys, we *derive* corresponding authentication keys, using a suitable one-way function. A cryptographically secure hash function, such as sha-256, may be used for this purpose, but we elect to use our CMAC primitive in this work. We assume an encryption key K_e exists (permanent, generated or delivered) and we wish to derive the corresponding authentication (MAC) key. The derivation procedure is as follows:

$$K_m = f_\Psi(K_e) = \mathcal{T}_\Psi(K_e)$$

where f is a key generating function (CMAC in our case), Ψ is a publicly available constant (128-bits) and K_e is the encryption key used as *key material* for the authentication key K_m. Here, we assume CMAC is a cryptographically secure one-way function. Hashing the key material, even with a known encryption key Ψ, in essence produces a "random-looking" result from the "random-looking" key material. The assumption of cryptographic security implies that the output should be uncorrelated with the input. An adversary must therefore work much harder to manipulate an encrypted and authenticated message.

The keys and key derivation used in the TSense project are the following:

K_{TA} is the secret identity of the sensor T. It's unique for each sensor and shared only with A. An authenticity key is derived from K_{TA}: $K_{TA,a} = f_\alpha(K_{TA})$. The permanent secret keys are only used for initial authentication.

K_{ST} and $K_{ST,a} = f_\beta(K_{ST})$ are the session encryption and authentication keys, shared between a sink S and sensor T. The key is randomly generated by A, and delivered securely to S and T, on successful identification of T in the authentication phase. The session key is used only for re-keying.

K_{STe} and $K_{STe,a}$ are the encryption and authentication keys for the data transfer protocol. They have a limited lifetime and must be replaced periodically by executing the re-keying protocol. The re-keying protocol does not deliver the key K_{STe}; rather, the *key material* R (a 128-bit random number) is delivered. Key derivation is $K_{STe} = f_\gamma(R)$ and $K_{STe,a} = f_\varepsilon(K_{STe})$.

α, β, γ and ε are randomly generated but publicly available 128-bit constants.

Note that the secure channel between sink and authentication server requires the key pair $(K_{AS}, K_{AS,a})$. However, in our implementation, these keys are implicitly negotiated and derived by the TLS channel between sink and authentication servers.

2.5.6 Generation of cryptographic keys

The system depends on one secret key – the 128-bit private device ID K_{TA} – generated and burned to the device EEPROM at time of manufacture. This key is a 16-byte long random value, generated by reading from the Unix `/dev/random` device. `/dev/random` allows access to an entropy pool, maintained by collecting environmental "noise" generated by the operating system and device drivers. This is generally regarded as much closer to true randomness than pseudorandom generators, although vulnerabilities have been found in some implementations (Gutterman et al., 2006). The non-blocking `/dev/urandom` can optionally be used for on-line key generation (session and transport keys) in the protocol to decrease latency.

We provide an utility, `generatekey`, for generation of random cryptographic keys. This utility is available along with the rest of the TSense code at the project code repository.

Chapter 3

Results

Our work began with construction of small prototypes, in order to get to grips with the Duemilanove and the Arduino development environment. Although we hit some initial problems, the experience of working with the Arduino system was overall a pleasant one. The basic system design decisions were then discussed in the group, which settled on the fully symmetric crypto approach. We then proceeded to code and test the cryptographic primitives on the supported platforms. The cryptographic protocols were designed in parallel with the code development. Once all the ducks were in a row, we proceeded to code the protocols in a multi-platform library, again for minimum coding effort and maximum maintainability. The final step was to integrate and test the system as a whole. We will describe our results in this section.

3.1 Verification of cryptographic primitives

Unit testing of cryptographic primitives and the various system components was used throughout the software development process on all supported platforms:

- Arduino Duemilanovae with Atmel ATmega328 processor, used for tsensor development
- Laptops with Intel and AMD 32- and 64 bit processors, running Ubuntu Linux and OS/X used for client, sink and authentication server development.
- Virtual machines, running Ubuntu Linux, used for sink and authentication server development and testing.

Tests included the following:

- Unit tests of the AES block cipher (encryption and decryption) using available test vectors, e.g. the key expansion and encryption test vectors from the FIPS (FIPS, 2001) document and the AES known answer test (KAT)[1] dataset.
- Unit tests of the CBC-mode encryption and decryption on all supported platforms and for plaintexts of various lengths, from one block (16-bytes) to several blocks in length. The bulk of our testing consisted of checking the consistency of encryption and subsequent decryption of randomly generated data chunks of various lengths, from sub-block size to several block sizes. We tested both data of even block lengths and partially filled blocks to reveal potential padding issues.
- Unit tests of the CMAC authentication algorithm, using randomly generated data as well as the test cases provided by Song, Poovendran, and Lee (2006).

[1] `http://csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES.zip`

- Unit testing data provider (test client), which can replace the sensor/client pair in testing against sink and authentication servers. This module is written in C and implements the protocol code as the tsensor device.

All unit tests are included in our open source implementation at `http://code.google.com/p/tsense`. We refer to the publicly available code for further verification of our implemented primitives.

3.2 Performance of cryptographic primitives

3.2.1 AES encryption and decryption in CBC mode on Arduino

The precise timing instruction `micros()`[2] on the Arduino was used to time successive encryption and decryption operations on randomly generated data of one to several blocks in length. An Arduino Duemilanovae board with an ATmega328 running at 16MHz was used for the test. Briefly, the results were that our implementation achieves an average throughput of 1213 blocks/sec for encryption and 592 blocks/sec for decryption on the Arduino. The worse performance of the decryption operation is most likely due to the more complex mixColumns transformations required for decryption. Future optimizations will address this.

Given that the data stream from the sensors is not continuous and each packet is rather short, we conclude that the throughput of the AES encryption algorithm is quite sufficient for our project.

Comparison with the XTEA block cipher

We did a quick comparison test of the XTEA (Needham & Wheeler, 1997) block cipher, based on the reference algorithm given on `http://en.wikipedia.org/wiki/XTEA`. XTEA was designed to be a very efficient block cipher, especially on resource constrained systems. The entire implementation takes about 16 lines of C code. It is interesting to see how this light-weight cipher compares in performance with the more secure AES. The test was performed in the same manner as that for our AES implementation. The results were a throughput of 1316 blocks/sec for encryption and 1574 blocks/sec for decryption. The recommended 32 rounds were used for both operations. We conclude that our AES implementation compares favorably with the XTEA one, given that the XTEA test operated in the slightly simpler ECB mode and has a shorter block length of 64 bits. Accounting for the differing block length shows a very acceptable throughput for our AES implementation, as shown in Table 3.1.

	Encrypt (blocks/sec)	Encrypt (Kbytes/sec)	Decrypt (blocks/sec)	Decrypt (Kbytes/sec)
AES	1213	18.9	592	9.3
XTEA	1316	10.3	1574	12.1

Table 3.1: Comparison of AES and XTEA performance on the Arduino platform (ATmega328 at 16MHz). Note that AES has a block length of 16 bytes compared to the shorter 8 byte block length of XTEA.

3.2.2 AES encryption and decryption in CBC mode on Intel-based platforms

The same code as timed in Section 3.2.1 was compiled in a test framework on a 32-bit Intel laptop with a Centrino Dual CPU, each core running at 1GHz, with 4MB L2 cache. Only one core of the CPU was used in the test. The test machine ran Ubuntu Linux 9.10 (kernel 2.6.31-22). Chunks

[2] `http://arduino.cc/en/Reference/Micros`

of random data, from 1 to 256 blocks at 16 bytes each, were generated, encrypted and decrypted. The encryption and decryption operations were timed using the C `gettimeofday` function, each measurement averaged over several repetitions. The results were that our implementation achieves roughly 2.7 Mbytes/sec. This result was also verified using the Unix `time`[3] utility.

Our test servers achieved throughput of roughly 4.3 MBytes/sec running the same test. The test servers are 32-bit virtual machines running Ubuntu Linux 10.04.1 LTS (2.6.32-24). The single CPU runs at 1995 MHz and has a 4MB L2 cache.

The AES implementation used in this project is byte oriented and close to the original published pseudocode (FIPS, 2001). A reference implementation, written by the supervisor for another project, achieves roughly four times the throughput using the more efficient 32-bit table-based algorithm, described by Daemen and Rijmen (1999). Updating our present library to utilize such features is reserved for future work.

We regard these tests as strictly "wallclock" indicators, but nevertheless useful for evaluating the performance of our implementation. Our conclusion is that the AES algorithm on a typical sink platform can achieve throughput which is orders of magnitude greater than that of the sensors, meaning a single sink server can service several hundred continuously transmitting sensors, especially factoring in the bursty nature of the communications.

3.3 *tsensor* memory footprint

Code size is quite important on a small device, such as the ATmega controller. Our compiled tsensor code binary is currently 14.5 KB of the total available program memory (flash) of 32KB. The AES test code binary, used for the timing tests in Section 3.2.1, is 11.2 KB, while the XTEA test code binary is 6.7 KB. Most of the difference in binary size between the two tests is due to the more complex AES algorithm. However, a size difference of less than factor two is quite acceptable, in our opinion, for the more secure AES block cipher. Note also that our full tsensor implementation with the AES block cipher currently occupies less than 50% of the available program memory of the ATmega328, which means that more complex cryptographic primitives, such as public key crypto, can most likely be implemented in future versions.

The RAM memory use is even more critical, as discussed earlier. Unfortunately, there are currently no methods to accurately measure the RAM use in the current Arduino library. However, we can estimate the amount of free RAM by the value of the heap and stack pointers. An serial API command for reporting the free RAM is included in the current tsensor version. At startup, the free RAM is reported as 1638 bytes, some of which is used by the controller itself and the Arduino bootloader. After full initialization, key expansion and allocation of a 20 byte measurement buffer, the available RAM is reported as 734 bytes.

RAM use can be improved in our current implementation, for example by expanding keys only as needed – AES-128 uses a key schedule of ten 16-byte keys, derived from the original 16 byte one. Currently, we store the session keys (encryption and authentication) expanded, which requires 320 bytes of RAM. Expanding the seldom used session key on demand would therefore free up considerable RAM. Constants for key derivation can also be moved into the EEPROM, which still has considerable free space.

3.4 System verification

We refer to our earlier discussion in Section 3.1 on the unit testing of the cryptographic primitives – the building blocks of the protocols. Regarding the correctness of our protocols, we refer to the publicly available code at `http://code.google.com/p/tsense` and the analysis of Section 3.5.

[3] `http://unixhelp.ed.ac.uk/CGI/man-cgi?time`

The daemons (sink and authentication servers) were tested on 32-bit Linux and 64-bit OS X/BSD platforms, with the main test environment being 32 bit Linux virtual machines, running Ubuntu 10.04 and the 2.6.32 kernel. Both the Linux and OS X/BSD machines used Intel CPUs and neither the daemons nor the encryption or network libraries they use were tested on machines with big-endian processors. Doing so might reveal bugs in the networking and encryption libraries.

The implementation of the cryptographic protocols – authentication, key-exchange and data transfer – was done on a test bed system, using a setup similar to the one shown in Figure 2.1. Laptops running Linux and a MacBook running OS X served as hosts (clients) for Duemilanove-based tsensors. Two virtual machines, running Ubuntu 10.04 LTS (kernel 2.6.32) served as sink and authentication servers. The clients used DHCP to obtain dynamic network addresses, while the two servers had fixed IPs. The sensor board was USB-connected to its host (client).

The protocols were tested individually at first and then combined. In summary, we were successful in getting the whole system up and running, albeit under fairly well controlled circumstances. Notably, handling of abnormal and unexpected events is lacking in the current version. However, we believe we have shown the feasibility of our implementation in a proof-of-concept setting.

3.5 Soundness of the cryptographic protocols

The following discussion is not intended to be a formal cryptographic analysis of the protocols, but rather an informal discussion about their security properties. More formal analysis is reserved for future work.

The code for the protocol is publicly available at `http://code.google.com/p/tsense`, satisfying Kerckhoffs' principle that "a cryptosystem should be secure even if everything about the system is public knowledge". The code was also written with security in mind, aiming to leave as few vulnerabilities as possible. Great care was taken to prevent the possibilities of buffer overflow, and because of the technical nature of the protocol itself, format-string attacks are not possible. Although this aims to provide local security and not directly security over the data, the system can only be considered secure as long as all parts are secure. If remote code execution were possible on either the tsensor, tssinkd or tsauthd — then an attacker could be able to alter data.

Since the cryptography in the protocol is based on AES it can only be considered secure as long as AES is considered secure. At the time of writing (September 2010) there are no known attacks against the AES cipher, except reduced round (reduced strength) variants, e.g. the super-SBox attack against 8-round AES-128 (Gilbert & Peyrin, 2009). We are using the full 10-round AES-128, giving considerable security margin. Further, the protocols are independent of the actual cipher used, meaning that higher strength AES or a completely different block cipher can be easily substituted.

The security of the entire system hinges on the secrecy of the private device key – the permanent keys burned onto each tsensor device. Extracting a single permanent key does compromise a single device, but not the entire system, since the keys are (with high probability) unique per device. However, recovering a limited amount of private device keys does give an attacker complete control over the corresponding identities. We assume key recovery is hard due to physical hardening of the device (tamper-proofing) and the serial protocol itself, giving an acceptable security level for the entire system and graceful degradation (rather than complete collapse) in case of limited number of key recoveries.

Confidentiality is provided in by using strong encryption. Additionally, bilateral implicit key authentication (Menezes et al., 1996, pp.498) is used in all protocols; correct decryption of shared, but secret, information, e.g. nonces, is an implicit proof that both parties hold the pairwise shared private key.

Authenticity is provided by applying an MAC to all messages, ensuring that message alterations can (with high probability) be detected by the recipients. The MAC keys are derived from the

encryption keys, observing the principle that a key should never be re-used for two different purposes. Again, this assumption only holds as long as the attacker does not have knowledge over the encryption keys.

Key derivation by means of an cryptographically secure one-way function[4], ensures that the derived key is statistically independent from the one it was derived from. Hashing a random-looking number, in our case by the equivalent of a keyed hash function, produces an equally random-looking result. The cryptographic properties of the one-way function imply that there should be no correlations between the input and output, resulting in statistically independent derived keys.

Cryptographic keys degrade (theoretically), however slightly, by every use. Limiting the use of the permanent device secret key to only authentication and session key exchange helps to preserve the security lifetime of the sensor device. Similarly, limiting the use of the session key to strictly re-keying helps to preserve its integrity. However, if an attacker would get hold of the current session key, he can get hold of all future session keys as well, since the new key-material is encrypted with a key derived from the current session key before it is sent. Strictly limiting the lifetime of transport encryption and authentication keys further helps to preserve the integrity of the system.

We have discovered some potential vulnerabilities in our present version of the protocol, which will be considered in future work. For example, adding the public device ID to the encrypted key-to-sink payload would give the tsensor two factors to identify the authentication server, instead of just one (the nonce). More "entropy" has to be added to the data transfer protocol, perhaps in the form of a nonce or dummy random number of several bytes. A promising approach is to add an sink to sensor ACK message for each data update (the current protocol is strictly fire-and-forget) which would carry a next-round data transfer nonce. This would also allow the sink to detect and respond to missing data updates, for example ones dropped by malicious clients.

3.6 Cost of materials and production sensor size

We base our sensor prototype on an Arduino Duemlianove experimentation board, mainly for the convenience of programming the ATmega328 in such an environment, rather than having to use a separate programmer. The cost of this board is $29.95 (+ shipping and import costs) from `http://www.sparkfun.com` in September 2010. An Atmel ATmega328 is available for $4.40 in quantities of 100+ from Sparkfun, but this would require custom circuit boards and some additional support hardware. The cost of sensors varies according to their sophistication, but the NTC thermistor and photoresistor used in our project cost less than $2 a piece. A reasonable estimated price for an assembled and tamper-proof tsensor is less than $20 in quantities using our demonstration sensors.

Production sensors would use a much smaller custom PCB and a surface mount version of the processor. Estimated size for a production unit is less than 2x2 cm, using a surface mount ATmega328 package. Even smaller sizes can be achieved using custom ICs. The size of the final package depends on the type of sensor and the hardening to be applied, but we can conclude that it is certainly practical to manufacture a reasonably small fully enclosed tsensor.

Our conclusions are that an ATmega-based tsensor is a relatively cheap and small unit, which fulfills our initial goal.

[4]We assume (without proof at this point) that the CMAC function has this property.

Chapter 4

Discussion

The primary goal of this project was to investigate the issues involved in securing measurement data as close to the source as possible – at the sensor it self. One factor to consider in this respect was the feasibility of implementing a strong cryptographic primitive, fulfilling the current security demands, even on resource constrained hardware, such as the ATmega. Early on, we decided to focus on symmetric encryption, rather than asymmetric, as this is was easily implemented, given the project time frame. Symmetric encryption is also less resource intensive, which was a bonus, since we had only a limited notion of how well the Arduino Duemilanove would be able to handle the workload we had in mind for it. Finally implementing symmetric encryption would still prove to be a useful subset of our larger goal, which is to implement a public key cryptosystem for TSense.

Our goal of producing a cross platform AES implementation, complete with a CBC-CMAC component, went smoothly, apart from a few minor easily resolved hiccups. The Arduino Duemilanove system was able to handle 128-bit AES and would probably be able to handle 256-bit AES, either off the shelf or with minor hardware enhancements, such as extra ROM space. The performance of our implementation was better than we expected and compared favorably with a the lightweight XTEA block cipher (see Table 3.1), which was especially pleasing since this cipher was purpose designed to be efficient and light-weight. Furthermore, AES is more complex that XTEA and our AES implementation is therefore considerably bulkier in terms of the size of code and the resultant binaries. Nevertheless our application fit easily in the 32 kilobyte program memory of the Arduino, and more importantly, did not exhaust the meager 2 kilobytes of RAM. This was especially satisfying, since our AES and CBC-CMAC implementations are not fully optimized. Further work on the AES library component would doubtless yield improvements in memory footprint and throughput.

In summary, while we did not achieve all that we set out to do, namely the implementation of a public key based system we still feel that we managed to achieve the larger overall goal of producing an end-to-end secure sensor system, featuring encryption and security features, that live up to current demands for such systems, but still managed to do so without compromising the goal of keeping the cost of the sensor system low. Most importantly, we demonstrated the feasibility of absolute end-to-end secure data stream, allowing positive receiver end verifiability of measurements.

Much work remains to be done in order to produce a robust system. In particular, error handling of all components must be completely overhauled. Error messages and timeouts (soft state) must be added to the protocols to handle unexpected events and authentication denials. Future versions will also support multiple potential sinks per client/sensor pair to allow for graceful fail-overs.

Utilizing asymmetric cryptography, at least for initial identification and session key exchange, was one of our initial goals and will be addressed in future work. This would allow private sensor keys to be truly private, that is, only reside burned onto the tamper-proof sensor device itself. All sinks can then safely hold the corresponding public keys, without sacrificing any security. This would in turn decrease the reliance on the authentication server; in fact, its role would be reduced to a fairly non-critical public key distribution service. Public key cryptographic algorithms are rather

well known, but their feasibility on resource constrained hardware less so. We will consider such algorithms, in particular elliptic curve encryption and signature algorithms, in future projects.

Efficiency was not the primary focus of this project. Rather, we concentrated on developing a working prototype that satisfies rigorous security requirements. Future work includes optimizing the protocols for more resource constrained networks, e.g. wireless transports. One way of increasing efficiency is to decrease security, e.g. by using shorter encryption and MAC keys, or sacrificing encryption and solely use MACs, even truncating the MAC. A more satisfying solution is to explore efficient authenticating block or stream ciphers. One way of increasing cryptographic protocol efficiency is to use *ciphertext stealing* (Schneier, 1996), which removes the need for padding and the associated waste of resources. We will explore such issues in future work.

Last but not least, the current device is anything but tamper-proof, despite the fact that this is one of our most crucial requirements. We will explore the issues involved in manufacturing an actual production sized and fully tamper-proof tsensor in future projects. For the purposes of the current proof-of-concept, we believe we have shown the feasibility of the concept, but simply did not have the time or resources necessary to produce a hardened prototype.

References

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., & Cayirci, E. (2002, March). Wireless sensor networks: a survey. *Computer Networks*, *38*(4).

Atmel. (2010). *Data sheet: Atmega48a/48pa/88a/88pa/168a/168pa/328/328p.*

Bellare, M., Canetti, R., & Krawczyk, H. (1996). Keying hash functions for message authentication. In (pp. 1–15). Springer-Verlag.

Bellare, M., Kiliany, J., & Rogawayz, P. (1999). *The security of the cipher block chaining message authentication code.*

Bellare, M., & Namprempre, C. (2007). *Authenticated encryption: Relations among notions and analysis of the generic composition paradigm.* London, UK: Springer-Verlag.

Brickell, E. F., Denning, D. E., Kent, S. T., Maher, D. P., & Tuchman, W. (1993, July). *SKIPJACK review. interim report. the SKIPJACK algorithm.*

Daemen, J., & Rijmen, V. (1999, March). *AES proposal: Rijndael.*

Daemen, J., & Rijmen, V. (2000). The block cipher Rijndael. In *CARDIS '98: Proceedings of the the international conference on smart card research and applications* (pp. 277–284). London, UK: Springer-Verlag.

Denning, D. E., & Sacco, G. M. (1981). Timestamps in key distribution protocols. *Commun. ACM*, *24*(8), 533–536.

Douceur, J. R. (2002). The Sybil Attack. In *IPTPS '01: Revised papers from the first international workshop on peer-to-peer systems* (pp. 251–260). London, UK: Springer-Verlag.

Dworkin, M. (2001, December). *NIST special publication 800-38a: Recommendation for block cipher modes of operation: Methods and techniques.*

Dworkin, M. (2004, May). *NIST special publication 800-38c: Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality.*

Dworkin, M. (2005, May). *NIST special publication 800-38b: Cipher modes of operation: The CMAC mode for authentication.*

Dworkin, M. (2007, November). *NIST special publication 800-38d: Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC.*

FIPS. (2001, November). *Federal Information Processing Standards Publication 107. Announcing the Advanced Encryption Standard (AES).* [online] `http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

Gilbert, H., & Peyrin, T. (2009). *Super-sbox cryptanalysis: Improved attacks for aes-like permutations.* Cryptology ePrint Archive, Report 2009/531. (`http://eprint.iacr.org/`)

Gutterman, Z., Pinkas, B., & Reinman, T. (2006, March). *Analysis of the linux random number generator.*

Hankerson, D., Menezes, A., & Vanslone, S. (2004). *Guide to elliptic curve cryptography.* Springer.

Kaliski, B. (1998, March). *RFC-2315: PKCS #7: Cryptographic message syntax.*

Karlof, C., Sastry, N., & Wagner, D. (2004). TinySec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on embedded networked sensor systems* (pp. 162–175). New York, NY, USA: ACM.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation*, *48*(177), 203-209.

Menezes, A. J., Oorschot, P. C. van, & Vanstone, S. A. (1996). *Handbook of applied cryptography.* CRC Press.

Needham, R. M., & Schroeder, M. D. (1978). Using encryption for authentication in large networks of computers. *Commun. ACM*, *21*(12), 993–999.

Needham, R. M., & Schroeder, M. D. (1987). Authentication revisited. *SIGOPS Oper. Syst. Rev.*, *21*(1), 7–7.

Needham, R. M., & Wheeler, D. J. (1997). *TEA extensions* (Tech. Rep.). Computer Laboratory, University of Cambridge.

Neuman, B. C., & Ts'o, T. (1994, September). Kerberos : An authentication service for computer networks. *IEEE Communications Magazine*.

NSA. (1998, May). *SKIPJACK and KEA algorithm specifications.* (Version 2)

Przydatek, B., Song, D., & Perrig, A. (2003). SIA: Secure Information Aggregation in Sensor Networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems* (pp. 255–265). New York, NY, USA: ACM.

Rogaway, P., Bellare, M., & Black, J. (2003). OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, *6*(3), 365–403.

Schneier, B. (1996). *Applied cryptography* (second edition ed.).

Schneier, B., & Whiting, D. (1997). Fast software encryption: Designing encryption algorithms for optimal software speed on the intel pentium processor. In *FSE '97: Proceedings of the 4th international workshop on fast software encryption* (pp. 242–259). London, UK: Springer-Verlag.

Song, J., Poovendran, R., & Lee, J. (2006, June). *RFC-4494: The AES-CMAC-96 algorithm and its use with IPsec.*

Song, J., Poovendran, R., Lee, J., & Iwata, T. (2006, June). *RFC 4493: The AES-CMAC algorithm.*

Teo, S.-G., Al-Mashrafi, M., Simpson, L., & Dawson, E. (2009). *Analysis of authenticated encryption stream ciphers.*

Wheeler, D. J., & Needham, R. M. (1995). TEA: a tiny encryption algorithm. In *Lecture notes in computer science* (Vol. 1008). Springer.