Friedrich-Alexander-Universität
Erlangen-Nürnberg

Studienarbeit

Secure Task Migration and Interprocess Communication in Reconfigurable, Distributed, Embedded Systems

by

Thomas Schneider

Matrikel-Nr.: 2105703

Supervision:
Dipl.-Ing. Dirk Koch
Prof. Dr.-Ing. Jürgen Teich

July 10, 2007

This document was produced with the typesetting system LaTeX2e.

Contents

Acknowledgements

First and foremost I would like to thank my thesis supervisor, Dipl.-Ing. Dirk Koch, who has shown a large and consistent interest in my project from the beginning to the end. Numerous scientific discussions and his deep knowledge in reconfigurable computing and hardware programming languages have greatly improved this work.

I wish to express my sincere gratitude to Prof. Dr.-Ing. Jürgen Teich, Head of the Department of Hardware-Software-Co-Design, University of Erlangen-Nuremberg, Germany for waking my interest in reconfigurable computing with his excellent lectures I attended during my studies and giving me the opportunity to write my thesis at his department.

Thanks a lot for the extensive support from the staffs of the Department of Hardware-Software-Co-Design, especially to Dipl.-Phys. Andreas Bininda for his substantial tool support and Dipl.-Ing. Thilo Streichert for his feedback on the embedded operating system.

My warm thanks are due to my father, Dr.-Ing. Klaus Schneider, for his continuous support including borrowing books, printing tons of papers, and most of all many fruitful conversations during the last years.

Last but not least I would like to thank my roommate Korbinian Riedhammer, my brother Matthias Schneider, and Thomas Holleczek for many helpful remarks on the manuscript.

Studienarbeit:
"Secure Task Migration and Interprocess Communication in Reconfigurable, Distributed, Embedded Systems"

Student: Thomas Schneider
Supervision: Dirk Koch and Prof. Jürgen Teich

Fundamentals: Nowadays, embedded systems like automotive applications consist more and more of FPGA (Field Programmable Gate Array) based ECUs.
Such FPGAs support to configure just parts of its logic and interconnect resources at run-time without any interference with the rest of the system. This process is called "partial runtime reconfiguration". Due to the progress in silicon industry, it is possible to integrate complete systems on a single FPGA-chip (SoC).
In the project ReCoNets [HKT04, KSD+06, SKHT06] we examine design methodologies for such embedded systems made upon small networks of hardware reconfigurable nodes and connections. ReCoNets presents a novel framework for increasing fault-tolerance and flexibility by separating functionality from the structure. Based on FPGAs in combination with a CPU, tasks implemented in hardware or software can migrate from one node to another in case of a node defect.

Description: In order to allow secure task migration, tasks must be *digitally signed*, e.g., signed by an authorized software or hardware manufacturer. This ensures that a ReCoNet can recognize if a hardware or software task is unauthorized or manipulated such that only trustworthy tasks will be executed. This enables a secure update functionality of both hardware and software in the field, e.g., via UMTS or WLAN. In addition, *certified nodes*, e.g., certified by an authorized hardware manufacturer, guarantee that tasks migrate only to trustworthy nodes. So, no malicious attack can exchange or add a node to a ReCoNet. Furthermore, *digital rights* ensure that each node can only execute specific groups of tasks. This allows to set *security levels*, in order to restrict some critical functionality. For example, it may be possible to restrict safety critical tasks to more reliable nodes in a ReCoNet.
Beside the authentication of tasks and nodes, the communication must ensure *integrity* (no changes) and *authenticity* (secure assignment to a sender) of messages, for example by *message authentication code* algorithms (MAC).
The goal of this assignment is the conceptual design and implementation of a secure layer for the communication and the task migration in a ReCoNet. Whenever useful, the work should utilize FPGA facilities. This may include instruction set extensions as well as static or dynamic reconfigurable hardware accelerators. In detail, the following problems have to be solved:

- Concepts for a secure task migration based on digitally signed tasks, certified nodes, and digital rights for the task execution on specific nodes. This includes the circumstance that tasks as well as shadow tasks may migrate inside the network.

- Concepts for a secure interprocess communication in a ReCoNet based on message authentication codes.
- Implementation and verification of the concepts on a prototype system consisting of ESM [BMA+05] platforms in combination with the softcore-CPU NIOS II [Alt06].
- Integration of the secure layer into the ReCoNet infrastructure based on the operating system MicroC/OS-II.
- Quantitative evaluation of the achieved security with respect to the cost (hardware and software).
- Writing the report and documentation.

Beside the assignment, Mr. Schneider is expected to write a detailed documentation of all design files. This includes an installation manual and a description of the test System. It is assumed that Mr. Schneider is familiar with programming in C/C++ and VHDL prior to the start of his work.

1 Introduction

`Reconfigurable, distributed, embedded systems` are a synergy of specialized `embedded systems`, reliable `distributed systems`, and flexibile `reconfigurable systems` like automotive, avionic or body-area networks that consist of communicating nodes specialized for certain purposes. The reliability and flexibility of these applications can be massively enhanced by introducing reconfigurability on node level as well as on network level.

In the research project RECONETS run by the University of Erlangen-Nuremberg - Department of Computer Science - Hardware-Software-Co-Design, the aspects of `fault-tolerance`, `availability` and `flexibility` of reconfigurable, distributed, embedded systems are being investigated [ReC]. Based on Field-Programmable Gate Arrays (FPGAs) in combination with a CPU, tasks implemented in hardware or software can migrate from one node to another in case of a node defect. If not enough hardware/software resources are available functionality can change its implementation style at runtime, i.e. a task can either run in hardware or software respectively.

Currently, the concepts of *dynamic HW/SW partitioning*, *shadow tasks*, *HW/SW morphing*, *HW/SW migration*, and *HW/SW checkpointing* increase `reliability` of the RECONETS in case of failure of links or nodes.

This thesis investigates how to extend the RECONETS to be able to detect and prevent *intentional attacks* on the system like adding untrusted nodes to the network, modifying messages (man-in-the middle), changing hard-/software stored in a single node (viruses, trojan horses), or executing untrusted software in the system.

The thesis is structured as follows:

Section 1.1 summarizes the investigated *security objectives of a ReCoNet*.
Section 1.2 shows what kinds of *attacks on FPGAs* are known and how to prevent them.
Chapter 2 introduces the *basic concepts of cryptography* used within this thesis.
Chapter 3 describes the developed *security architecture for the* RECONETS.
Chapter 4 explains its *integration into the existing* RECONETS *infrastructure*.
Chapter 5 gives an *outlook* on further work. Chapter 6 is a *summary* of the work presented in this thesis.

1.1 Security Objectives of a ReCoNet

In order to use a ReCoNet in a security critical environment like a car or an airplane additional requirements on *integrity*[1] and *authenticity*[2] have to be fulfilled as shown in Fig. 1.1.

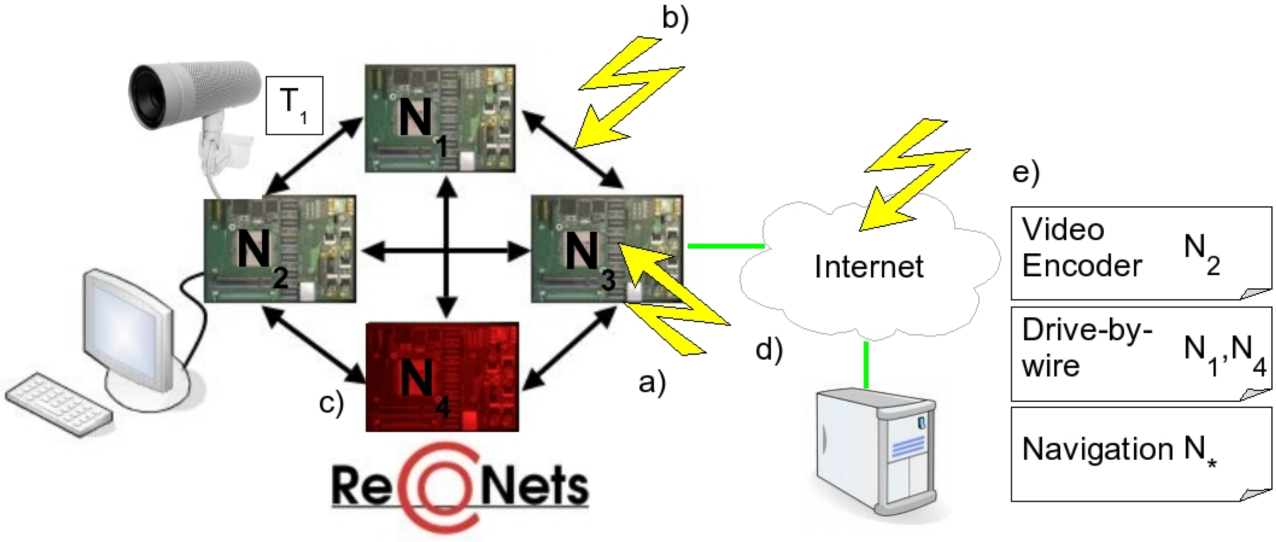

Figure 1.1: *Aspects of integrity and authenticity in a ReCoNet*:
a) Integrity and authenticity of HW- and SW-modules
b) Integrity and authenticity of messages
c) Authenticity of nodes
d) Update in field
e) Restrict binding of tasks to nodes

In the following these security objectives and possible attacks on the system are explained.

a) Integrity and authenticity of HW- and SW-modules

An attacker should not be able to alter hardware- or software-modules in the memory of the nodes (integrity) and also not be able to add untrusted modules like viruses to the system (authenticity).

b) Integrity and authenticity of messages

Messages exchanged between the nodes must not be alterable (integrity) and have to originate definitively from the node that claims to have sent the message (authenticity). Any modifications of messages are detected.

[1] *Integrity* ensures that accidental or intentional modifications of data are detected.
[2] *Authenticity* allows an unambiguous mapping from data to its initiator.

c) Authenticity of nodes

An attacker must neither be able to connect untrusted nodes to the network nor to clone or replace an existing node (red node in Fig. 1.1). Each node of the ReCoNet must identify itself to the other nodes in order to guarantee that it is a trusted node that is allowed to take part in the ReCoNet.

d) Update in field

Hard- and software modules can be updated offline via a data medium connected to the ReCoNet or online over a public network like the internet in a trustworthy way. The modules can originate from different trusted manufacturers that are allowed to produce specific kinds of modules only. An attacker can neither modify the modules during submission nor update the systems with untrusted, malicious software.

e) Restrict binding of tasks to nodes

Hard- or software-tasks can be restricted to run only on dedicated nodes by *digital rights*. Different restrictions of a ReCoNet are covered like shown in Fig. 1.1:
The ”Video Encoder” can only run on node N_2 as this node has direct access to the connected camera. (*connected periphery*)
The task ”Drive-by-wire” is allowed to run on nodes N_1 and N_4 only as these nodes have less periphery connected and therefore are more reliable as the frequency of interrupts for the CPU might be lower. (*reliability*)
The task ”Navigation” is allowed to run on every node of the ReCoNet.

1.2 Attacks and Countermeasures on FPGAs

Each node of a ReCoNet is a reconfigurable, FPGA-based system for which known attacks and effective countermeasures against them have been summarized in [WGP03]. The security architecture of the ReCoNet defined in this thesis is based on these protections of single nodes. In [WGP03] the authors categorize known attacks on FPGAs into the following categories and explain different countermeasures:

- *Blackbox Attacks*: ”The attacker inputs all possible combinations, while saving the corresponding outputs. The intruder is then able to extract the inner logic of the FPGA, with the help of the Karnaugh map or algorithms that simplify the resulting tables.” This is practically only feasible on very small FPGAs as the complexity of this attack grows exponentially with the size of the FPGA: In each possible state the attacker would have to input all possible inputs to extract the inner logic of the circuit.

- *Readback Attacks*: "Readback is a feature that is provided for most FPGA families. This feature allows to read a configuration out of the FPGA for easy debugging." Most FPGA manufacturers provide readback-lock bits to disable this feature. To ensure that nobody can turn off the readback-lock bits by fault injection the FPGA has to be embedded into a secure environment, where the whole configuration is deleted or the FPGA is destroyed if an electromagnetic interference, heating or glitches in power-supply were detected.

- *Cloning of SRAM FPGAs*: "The configuration data is stored externally in non-volatile memory (e.g., PROM) and is transmitted to the FPGA at power up in order to configure the FPGA. An attacker could easily eavesdrop the transmission and get the configuration file." Today's FPGAs provide support for encrypted bitstreams. The bitstream is symmetrically encrypted before storing it in external non-volatile memory and decrypt it on-chip on configuration. The symmetric key is stored on-chip - either in battery backed volatile memory (like in Xilinx Virtex-II using 112 bit 3-DES [AT]) or in one-time programmable non-volatile memory (like in Altera Stratix II using 128 bit AES [Alt]).

- *Reverse-Engineering of Bitstreams*: In order to get the design of proprietary algorithms or the secret-keys, one has to reverse-engineer the bitstream. The condition to launch the attack is not only that the attacker has to get the bitstream, but furthermore the bitstream must not be encrypted.

- *Physical Attacks*: "The aim of a physical attack is to investigate the chip design in order to get information about proprietary algorithms or to determine the secret-keys by probing points inside the chip. Hence, this attack targets parts of the FPGA, which are not available through the normal I/O pins. This can potentially be achieved through visual inspections and by using tools such as optical microscopes and mechanical probes. However, FPGAs are becoming so complex that only with advanced methods, such as Focused Ion Beam (FIB) systems, one can launch such an attack." In [WGP03] the authors analyzed the effort needed to physically attack FPGAs based on SRAM, Anti-fuse and FLASH technology.

- *Side-Channel Attacks*: "Any physical implementation of a cryptographic system might provide a side channel that leaks unwanted information. Examples for side channels include in particular: power consumption, timing behavior, and electromagnetic radiation." While *Simple Power Analysis* (SPA) attacks are feasible on FPGAs *Differential Power Analysis* (DPA) would be harder to implement on an FPGA than on an ASIC as the power consumption of interconnects (60%) is much higher than that of clocking (14%), logic (16%), and others (10%). The numbers in brackets are estimates for a XILINX Virtex-II FPGA as reported in [WGP03].
  "The methods [to prevent side-channel attacks] can generally be divided into software and hardware countermeasures, with the majority of proposals dealing with software countermeasures. "Software" countermeasures refer primarily to algorithmic changes, such as masking of secret-keys with random values, which are

also applicable to implementations in custom hardware or FPGA. Hardware countermeasures often deal either with some form of power trace smoothing or with transistor-level changes of the logic. Neither seem to be easily applicable to FPGAs without support from the manufacturers. However, some proposals such as duplicated architectures might work on todays FPGAs." Also measurements to detect tampering attempts like glitches in power-supply, heating or jitter in the system clock could prevent special side-channel attacks.

2 Cryptographic Fundamentals

This chapter shortly presents the essential cryptographic algorithms and protocols used in this thesis. These and further information can be found in [MVO96, Sch96].

The *Kerckhov Principle* [Ker83] states that the security of any crypto-system should only depend on the secrecy and unpredictability of secret keys whereas the used algorithms should be public. This allows everybody to examine the level of security of the proposed security system.

Table 2.1 shows the notations throughout this thesis.

Notation	Meaning	Section
A_p	Public-key of user A	
A_s	Secret-key of user A	
K_{AB}	Symmetric key K shared between A and B.	
$K[I]$	Symmetric encipherment of information I using the symmetric-key K.	2.3
$A_p[I]$	Asymmetric encipherment of information I using the public-key of A.	2.4.1
$A_s[I]$	Asymmetric encipherment of information I using the secret-key of A.	2.4.1
$K\{I\}$	Information I symmetrically signed with K.	2.2.3
$A\{I\}$	Information I asymmetrically signed with A_s.	2.4.2

Table 2.1: Notation for keys, encryptions and signatures

2.1 Random Numbers

The security of many cryptographic systems depends on the unpredictability of random numbers used for:

- Key generation[1]
- Initialization vectors for modes of operation for symmetric block ciphers (2.3.1)
- Nonces (numbers used once) in cryptographic protocols (2.4.4)

[1]The Kerckhov principle requires the randomness (unpredictability) of keys as described before.

A *random bit generator* (RBG) is a device or an algorithm which outputs a sequence of statistically independent and unbiased[2] binary digits.

A *random number generator* (RNG) produces uniformly distributed numbers in the interval $[0,N]$. It can be constructed out of a RBG by successively taking $\lceil \log_2 N \rceil$ bits of its output and discarding all numbers that are greater than N.

2.1.1 Random Bit Generators (RBG)

(True) random bit generators ((T)RBG) are based on truly random events that are unpredictable.
Hardware-based random bit generators exploit the randomness of physical effects that are quantum mechanically unpredictable. Methods which can be implemented on a chip include:

- Thermal noise from a semiconductor diode or resistor
- The frequency instability of a free running oscillator
- The amount a metal insulator semiconductor capacitor is charged during a fixed period of time

Software-based random bit generators are mostly based on a combination of:

- the system clock
- user input and elapsed time between input events
- operating system values like system load and network statistics

In [MVO96, chapter 5.4] several statistical tests to measure the quality of randomness of PRBGs are presented.

2.1.2 Pseudo-Random Bit Generators (PRBG)

A *pseudo-random bit generator* (PRBG) is a deterministic[3] algorithm that takes a truly random bit sequence of length k (*seed*) as input and outputs a sequence of length $l \gg k$ that "appears" to be random.

Standardized PRBGs are the ANSI X.9.17 PRBG (based on 3-DES [Nat99]) or the FIPS 186 PRBG (based on SHA-1 or DES) described in [MVO96, chapter 5.3].

[2] '0' and '1' occur with same probability.
[3] Given the same initial seed, the generator will always produce the same output sequence.

A *cryptographically secure pseudo-random bit generator* (CSPRBG) is a PRBG that passes the *next-bit test*: There is no polynomial-time algorithm that can predict the $(m+1)^{st}$ bit of the output sequence on input of the first m bits with a probability significantly greater than 0.5.

Examples for CSPRBGs are [MVO96, §5.5]:

- *Blum-Blum-Shub*-PRBG (BBS-PRBG) based on the intractability of the integer factorization problem: $x_i = x_{i-1}^2 \mod N$, where $N = pq$ and p, q are two secret large primes both congruent 3 modulo 4.
- *RSA-PRBG* based on the intractability of the RSA problem: $x_i = x_{i-1}^e \mod N$, where $N = pq$ and p, q are two secret large primes and e a RSA encryption exponent.

The cryptographically secure pseudo-random bit sequence is $(z_i) = ((x_i) \mod 2)$.

2.2 Cryptographic Hash Functions

A *hash function* is a function $\hbar$ that fulfills these properties:

1. *compression* - $\hbar$ maps an input x of arbitrary length to an output $\hbar(x)$ of fixed length.
2. *ease of computation* - given $\hbar$ and an input x, $\hbar(x)$ is easy to compute[4].

A *cryptographic hash function h* - also known as *cryptographic checksum* is a hash function $\hbar$ with the following additional properties:

3. *preimage resistance* - for essentially all pre-specified outputs y, it is computationally infeasible[5] to find a pre-image x such that $h(x) = y$. In other terms - h can not be inverted practically.
4. *2nd-preimage resistance* - it is computationally infeasible to find a second preimage that has the same hash value as a given input, i.e. given x find $x' \neq x : h(x) = h(x')$.
5. *collision resistance* - it is computationally infeasible to find any two distinct inputs x, x' that hash to the same output, i.e. $h(x) = h(x')$.

Collision resistance is the strongest property of all:

$$\text{2nd-preimage resistance} \Leftarrow \text{collision resistance} \Rightarrow^{6} \text{preimage resistance}$$

Given any hash-function of bitlength n the following brute-force attacks exist [Sch96]:

[4] computable in polynomial time

[5] not computable in polynomial time

[6] For cryptographic hash functions where compression factor ≥ 2, i.e. $\#dom(h) \geq 2 \cdot \#codom(h)$.

- *2nd-preimage attack*: a 2nd-preimage for a given value can be found in approximately $0.5 \cdot 2^n = 2^{n-1}$ operations by hashing random values.

- *collision attack* (birthday-attack): a collision can be found in approximately $1.2 \cdot \sqrt{2^n} = 1.2 \cdot 2^{n/2}$ operations by hashing random values and searching for a duplicate.
  This attack is the reason why the cryptographic hash function used in a crypto system must have about twice the size of the used symmetric cipher for equal computational security of both cryptographic primitives. Thus the security layer for the RECONETS uses a 256 bit cryptographic hash function (SHA-256) and a 128 bit symmetric encryption algorithm (AES-128) that are described later in this chapter.

2.2.1 Modification Detection Codes (MDC)

A *modification detection code (MDC)* $H(M)$ is a collision resistant cryptographic hash function h used to ensure the *integrity* of a message M (detect modifications). In contrast to error detection codes (EDC) like CRC-Checksums (cyclic redundancy check) it is however computationally infeasible to find a message that hashes to a given value (preimage resistance of h). The cryptographic hash value of a message M, $H(M)$ is called its *message digest (MD)*.

Examples for MDCs are MD5, SHA-1 and the SHA-2 family of hash-functions (Secure Hash Algorithm). [Bar]

As proposed in [RRS06] MD5 and SHA-1 should no longer be used as attacks on both are known: MD5 because of its too short bit-length of 128 bit where a collision can be found in 2^{64} operations and SHA-1 because of an attack published in Feb 2005 which reduces the effort to find a collision from 2^{80} down to 2^{69} operations.
"While NIST continues to recommend a transition from SHA-1 to the approved SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512), NIST has also decided that it would be prudent in the long-term to develop one or more hash functions through a public competition, similar to the development process for AES." [Nat07, Sch07]

The security layer for the RECONETS implemented in this thesis uses SHA-256 which is thought to be practically collision resistant by now. [RRS06]

2.2.2 Secure Hash Algorithm SHA-256

SHA-256 is a collision resistant cryptographic hash function designed and standardized by the National Institute of Standards and Technology (NIST) [Nat02]. It hashes a message M, having a length of l bits, $0 \leq l < 2^{64}$ to a 256-bit message digest. First the message is padded to a length which is a multiple of 512 bit. After that, each 512-bit block of the message is processed iteratively in 64 rounds starting from a fixed initialization vector (IV) with a length of 256-bit.

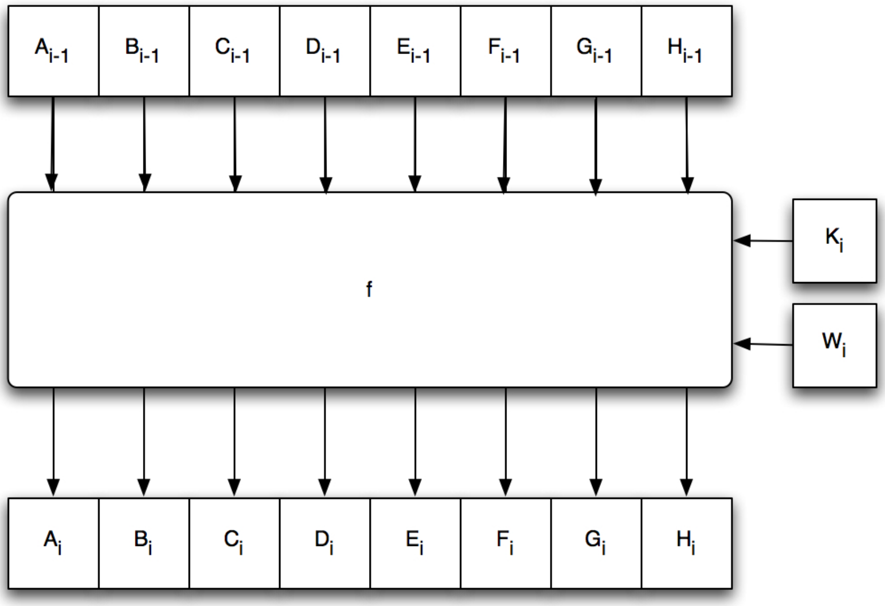

Figure 2.1: *Structure of SHA-256 round i*: The eight 32-bit state registers A ...H are updated by the round function f that depends on the round constant K_i and the next data block to hash W_i.

Fig. 2.1 shows how the eight 32-bit working registers ($A \ldots H$) containing the 256-bit hash value and initially the fixed initialization vector ($A_0 \ldots H_0$) are updated in each of the 64 rounds. The non-linear function f has the old values of the working registers ($A_{i-1} \ldots H_{i-1}$), a round-dependent constant K_i and the data derived from the currently hashed block W_i as inputs and computes the new values of the working registers ($A_i \ldots H_i$). f consists of multiple cyclic shifts, boolean functions (xor, and, not) and modular additions of its input values. ($A_0 \ldots H_0$), f, K_i and W_i are specified in [Nat02].

2.2.3 Message Authentication Codes (MAC)

A *message authentication code* (*MAC*) or *symmetric signature* is a keyed hash function $H(K,M)$ that has two inputs: the data to be hashed (M) and a secret-key (K). A MAC for M can be computed or verified if and only if K is known. Besides the two properties of hash functions *compression* and *ease of computation* a MAC holds this additional property:

3. *computation-resistance* - given zero or more text-MAC pairs $(M_i, H(K,M_i))$, it is computationally infeasible to compute any text-MAC pair $(M, H(K,M))$ for any new input $M \notin \bigcup_i M_i$.

A MAC can be used to ensure both the *authenticity* (correct sender) and the *integrity* (no modification) of a message:

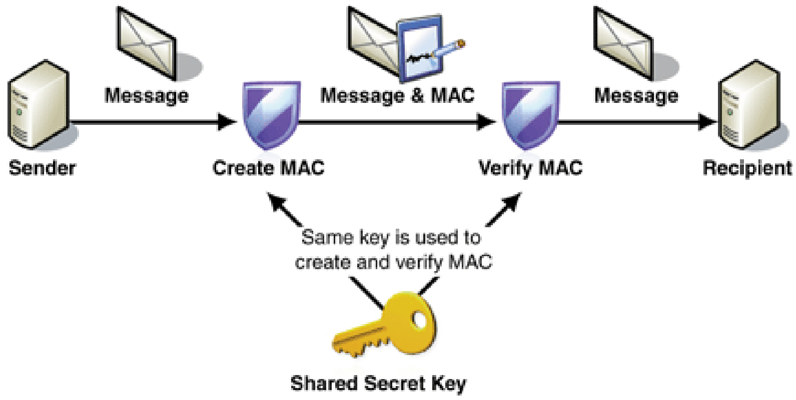

Figure 2.2: *Symmetric-Signature* (from [IMGc]): The sender signs the message symmetrically (MAC) with the shared secret key. The recipient verifies the signature with the same key to detect any modification of the message.

The sender Alice wants to send a message M to the recipient Bob. On receipt, Bob wants to ensure that M was not modified (*integrity*) and really originated from Alice (*authenticity*) as shown in Fig. 2.2:

1. Alice and Bob share a common secret-key K_{AB} known only to them.[7]

2. Alice computes the MAC m of the message M using K_{AB}: $m = H(K_{AB}, M)$ and sends (M, m) to Bob.
   $K_{AB}\{M\} := (M, m) = (M, H(K_{AB}, M))$ denotes such a message M which is symmetrically signed with K_{AB}.

3. Bob receives $(\hat{M}, \hat{m})$, computes $X = H(K_{AB}, \hat{M})$ and compares X to $\hat{m}$. If they are identical, he can be sure, that M originates from Alice and was not modified, as only Alice knows K_{AB} and is able to compute the right MAC of M (computation-resistance).

2.2.4 Hash-MAC (HMAC)

Any MDC $H(M)$ (like SHA-256) can be used to construct a MAC $H(K, M)$ with key K and message M with the following scheme [KBC97]:

$$H(K, M) = H((K \oplus opad)|H((K \oplus ipad)|M))$$

[7] How two parties can securely agree on such a common secret-key will be explained in 2.4.4.

where $\oplus$ denotes the bitwise XOR, $|$ the concatenation of two bitstrings, B the block length of H in bytes (e.g. $512/8 = 64$ for SHA-256 which hashes blocks of 512 bit), opad (outer padding) = 0x5C repeated B times, ipad (inner padding) = 0x36 repeated B times. K should have the same length as the block length B.

2.3 Symmetric Cryptography

A *symmetric-key cipher* is a pair of complementary functions $(K[M], K^{-1}[C])$, where $K[M]$ is the encryption function, $K^{-1}[C]$ the decryption function, K the symmetric-key, M the plaintext message and $C = K[M]$ the encrypted ciphertext. (Fig. 2.3)
The two functions $K[M]$ and $K^{-1}[C]$ are complementary:

$$K^{-1}[K[M]] = M$$

As described in [MVO96, chapter 7] a symmetric-key cipher has to be resistant against several attacks. In general it must be computationally infeasible neither to reconstruct parts of the plaintext M from the ciphertext C (*partial break*) nor to reconstruct the key K out of many ciphertexts C_i (*total break*).
These properties guarantee *confidentiality* of encrypted messages.

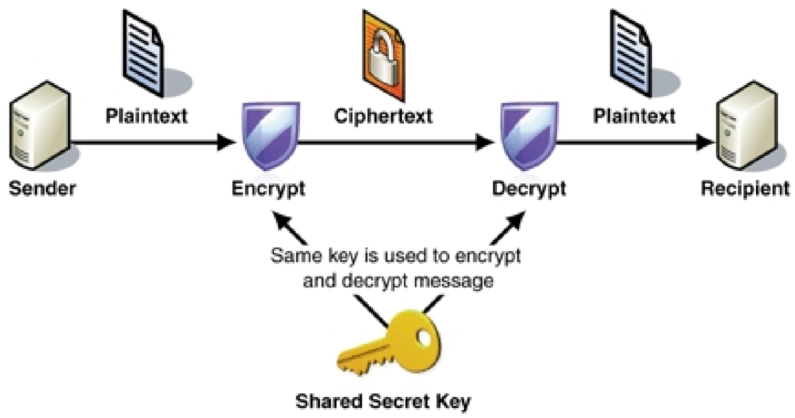

Figure 2.3: *Symmetric-Key Cipher* (from [IMGb]): The sender encrypts the plaintext with the shared secret key and transmits the encrypted ciphertext. The recipient decrypts the ciphertext with the same key to get back the plaintext.

Examples for symmetric-key ciphers are block-ciphers like Triple-DES, IDEA, Twofish, Serpent or AES (Rijndael), and stream-ciphers like RC4. [Fut00]

As AES is standardized and the most widely used symmetric cipher with a reasonable key length it will be used as the symmetric encryption algorithm for the security layer of the RECONETS.

2.3.1 Advanced Encryption Standard (AES)

AES was standardized by the National Institute of Standards and Technology (NIST) as FIPS 197 [Nat01] in 2001 after a 5-year standardization process as successor of DES. The algorithm developed by Joan Daemen and Vincent Rijmen was chosen out of 15 proposed AES candidates. It works on 128-bit Blocks and uses keys of size 128, 192 or 256 bits.
The data to be encrypted is written into a 4x4 matrix of bytes which is then transformed in 10, 12 or 14 rounds depending on the key size. Each round is a substitution-permutation network (SPN) consisting of these four steps shown in Fig. 2.4:

1. *AddRoundKey*: a round-key derived from the key is XOR-ed to the elements of the matrix

2. *SubBytes*: each element of the matrix is substituted by a fixed 8-bit to 8-bit lookup table (S-Box)

3. *ShiftRows*: the rows are rotated by a fixed offset:
$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \Rightarrow \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{pmatrix}$$

4. *MixColumns*: the columns are mixed by a linear transformation:
$$\begin{pmatrix} b_{*,0} \\ b_{*,1} \\ b_{*,2} \\ b_{*,3} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} a_{*,0} \\ a_{*,1} \\ a_{*,2} \\ a_{*,3} \end{pmatrix}$$

Decryption inverts these operations in reverse order.

AES is a so called *block cipher* that enciphers blocks of a fixed length of 128-bit. A *mode of operation* describes, how a block cipher can be used to encipher longer messages by chaining single blocks. Possible modes are ECB , CBC , CFB , OFB and CTR described in [MVO96, chapter 7]. All modes (except ECB) require a pseudo-random *initialization vector* (IV) to avoid that two identical plaintexts are encrypted to the same ciphertext.

There are some theoretical attacks on AES based on its algebraic structure that might be used to break AES in the future [Sch02], however they are impracticable by now.

The main disadvantage of symmetric cryptography is that two parties wishing to communicate confidentially or to ensure authenticity and integrity of messages have to agree on a common secret-key in advance (*key-distribution problem*). Asymmetric cryptography solves this problem.

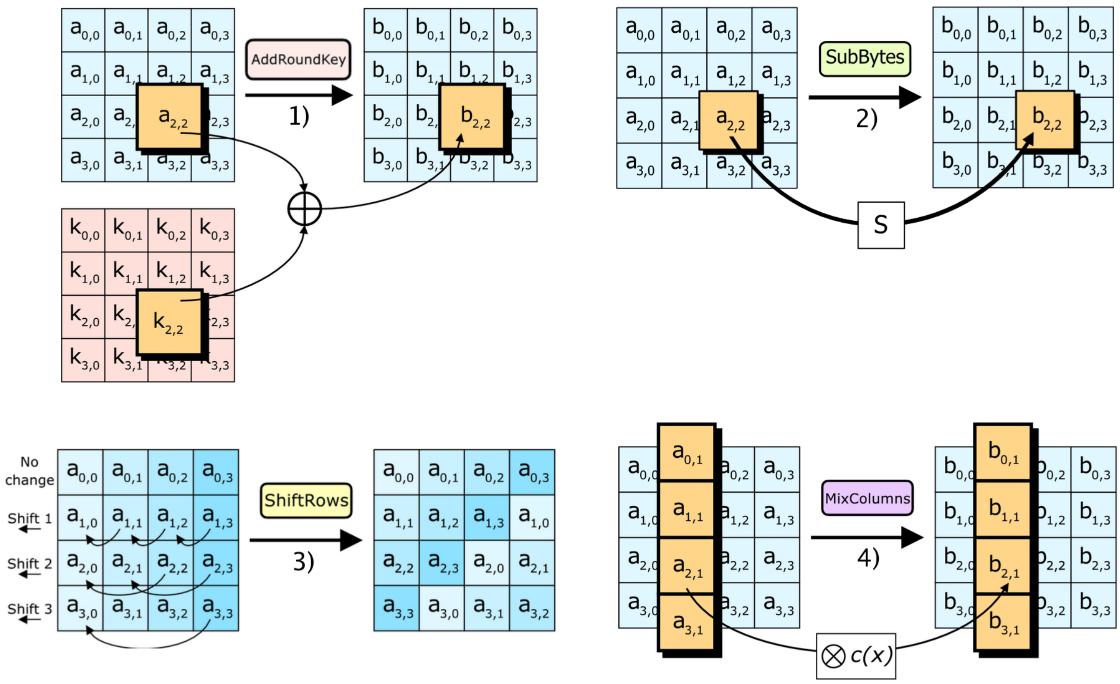

Figure 2.4: *Round transformations of AES* (from [IMGa]): The 512 bit message block is written into a 4x4 matrix of 32 bit values and the transformations AddRoundKey, SubBytes, ShiftRows and MixColumns are applied to it in each round.

2.4 Asymmetric Cryptography

Asymmetric cryptography allows two parties to communicate securely without having a shared common secret (e.g. a key for a symmetric cipher) before. It was invented in the early 1970s by James H. Ellis, Clifford Cocks, and Malcolm J. Williamson of the British Government Communications Headquarters (GCHQ). [Ell70, Coc73, Wil74] It is based on pairs of asymmetric keys - a *public-key* K_p and a *secret-key* K_s also called *private-key*. As in symmetric ciphers, an asymmetric cipher consists of two complementary functions for encryption $K_p[M]$ and decryption $K_s[C]$:

$$K_s[K_p[M]] = M$$

It must also be computationally infeasible to neither gain information about parts of the message M out of its ciphertext $C = K_p[M]$ and K_p, nor to reconstruct K_s out of multiple ciphertexts C_i and K_p.
The drawback of asymmetric-key algorithms is, that they are much slower than symmetric-key algorithms and need longer keys. Thus *hybrid crypto systems* [Den04] working with *temporary-keys* are widely used. At the beginning of a session, an asymmetric algorithm is used to exchange a symmetric temporary-key between the two parties wishing to communicate. This is used afterwards to encrypt messages with a much faster symmetric-key

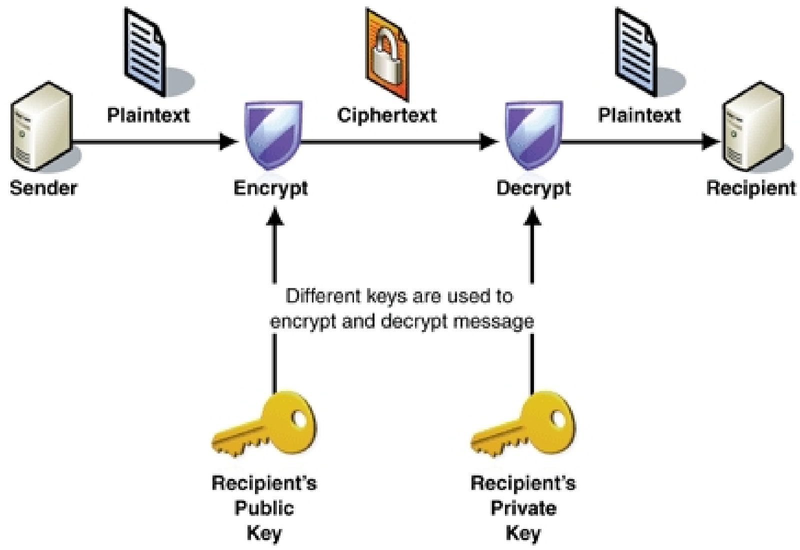

Figure 2.5: *Asymmetric-Key Cipher* (from [IMGb]): The sender encrypts the plaintext with the recipient's public key and sends the encrypted ciphertext. The recipient decrypts the ciphertext with his private key. Two different, corresponding keys are used: a public key for encryption and a private key for decryption.

algorithm like AES to guarantee confidentiality or with a MAC to guarantee integrity and authenticity of messages.

2.4.1 Asymmetric Ciphers - RSA

Asymmetric ciphers are based on hard mathematical problems like discrete logarithms in special groups (ElGamal, Elliptic Curve Cryptography (ECC)) or factorization of large integers (RSA) for which no efficient algorithms are known by now [MVO96].
The upcoming techniques based on elliptic curves provide a shorter key-length and faster execution time than the classical approaches. [Ros98]

RSA was invented in 1977 by Ronald L. Rivest, Adi Shamir and Leonard Adleman [RSA78]. As it is currently the most widely used public-key crypto system this thesis will also use this algorithm for asymmetric cryptography.
RSA is standardized in PKCS-1 (Public Key Cryptography Standard) [RSA02].

Each participant A generates an individual asymmetric-key pair (A_p, A_s):

1. choose two large primes p, q at random

2. compute $N = p \cdot q$, $\phi(N) = (p-1) \cdot (q-1)$. [8]

[8] ϕ is Euler's totient function.

3. choose a public exponent $1 < e < \phi(N)$, with $gcd(e, \phi(N)) = 1$. [9]
4. compute $d = e^{-1} \mod \phi(N)$ with the Extended Euclidean Algorithm
5. $A_s := (d,N), A_p := (e,N)$.

A publishes A_p and keeps A_s secret.

If Bob wants to encrypt a message M for Alice, he does the following:

1. get A's public-key $A_p = (e,N)$[10]
2. transform M into chunks M_i with $|M_i| < N$
3. encrypt M_i with the A's public-key to $C_i = A_p[M_i] := M_i^e \mod N$
4. send C_i to A.

When Alice receives C_i, she can decrypt the message with her secret-key $A_s = (d,N)$:

1. decrypt $M_i = A_s[C_i] := C_i^d \mod N$
2. transform chunks M_i back to message M

RSA is a correct asymmetric encryption scheme as it satisfies:

$$A_s[A_p[M_i]] = (M_i^e \ mod \ N)^d \ mod \ N = M_i^{ed} \ mod \ N \equiv M_i^{1 \ mod \ \phi(N)} \ mod \ N = M_i \qquad (2.1)$$

The security of RSA depends on the unfeasibility to factor N into its prime factors p and q. RSA Security organizes the RSA Factoring Challenge where the largest factorized RSA modulus by 2006 is RSA-640, a 640-bit number [RSA]. RSA and the NIST recommend to use a modulus of 1024 or better 2048-bits to guarantee long-term security [RDS02]. The security layer for the ReCoNets uses asymmetric 1024 bit keys.

2.4.2 Digital Signatures

Digital signatures or *asymmetric signature* are the asymmetric counterpart to the symmetric MACs. They also guarantee message *integrity* and *authenticity* but without prior agreement on a common secret-key.

The basic setting is the same as in the symmetric case however with two different keys as shown in Fig. 2.6.
There are several schemes for digital signatures like DSA, its counterpart ECDSA based on elliptic curves, ElGamal signatures or RSA signatures.

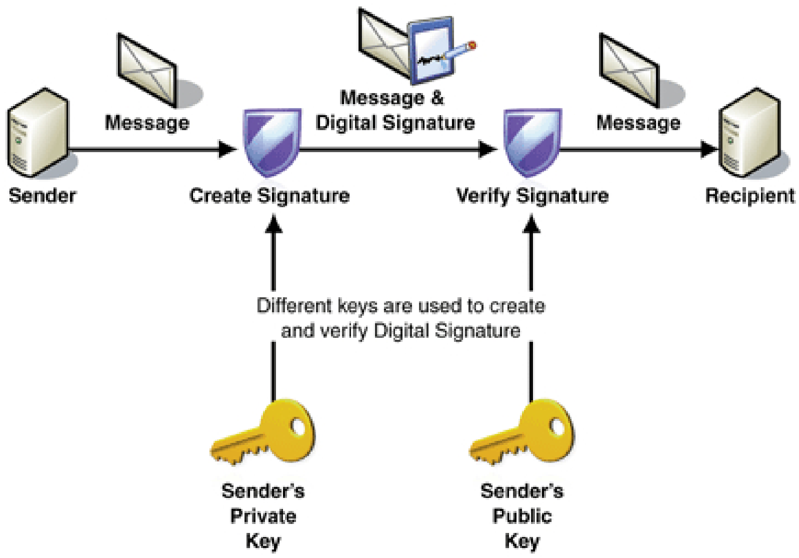

Figure 2.6: *Asymmetric-Signature* (from [IMGc]): The sender signs the message asymmetrically with his private key. The recipient verifies the signature with the sender's public key to detect the sender and any modification of the message. Two different, corresponding keys are used: a private key for signing and a public key for verifying.

In the following RSA digital signatures that are used in our implementation are described:

Alice signs a message M as follows (Fig. 2.7):

- create a message digest m of M using a MDC $H(M)$: $m = H(M)$.
- sign the message digest with her secret-key $A_s = (d, N)$:
  $S = A_s[m] = m^d \mod N$.
- the signed message consists of the message along with the signature (M, S).
  $A\{M\} := (M, S) = (M, A_s[H(M)])$ denotes such a message M that is asymmetrically signed by A.

Bob receives the signed message $(\hat{M}, \hat{S})$ from A and verifies the signature as follows (Fig. 2.8):

- get A's public-key $A_p = (e, N)$[10].
- compute the message digest of $\hat{M}$: $\hat{m} = H(\hat{M})$
- verify the signature with A's public-key: $X = A_p[\hat{S}] = \hat{S}^e \mod N$.

[9] The *greatest common divisor* can be computed efficiently with the Euclidean algorithm.

[10] and verify that the public-key really belongs to Alice e.g. by checking certificates as described in subsection 2.4.3.

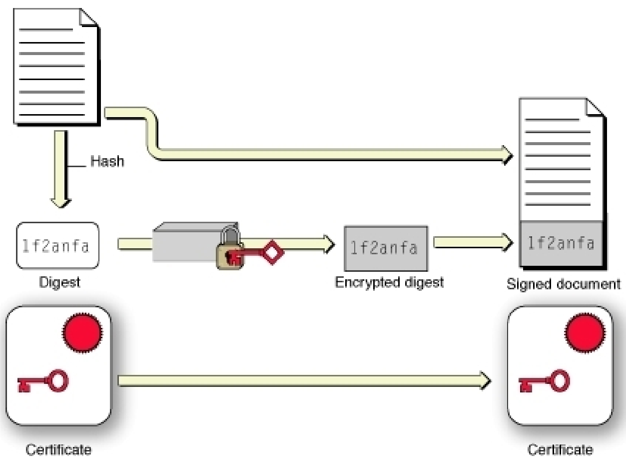

Figure 2.7: *Creation of a digital signature* (from [IMGd]): The document is hashed, the small hash digest is encrypted with the private key of the signer and attached to the document as signature. The public certificate ensures the correct mapping between the signer's name and his public key to verify the signature.

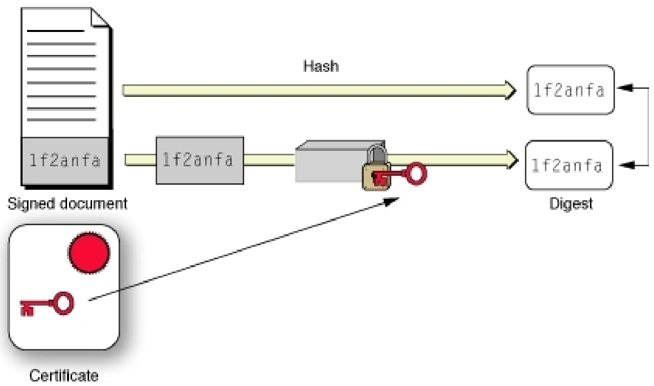

Figure 2.8: *Verification of a digital signature* (from [IMGd]): The signature is decrypted with the public key of the signer and compared to the hash digest of the document.

- if $\hat{m} = X$, he accepts the signature and knows, that the message M really originates from A and has not been altered.

RSA is a correct signature algorithm as it satisfies[11]:

$$A_p[A_s[m]] = (m^d \ mod \ N)^e \ mod \ N = m^{de} \ mod \ N \equiv m^{1 \ mod \ \phi(N)} \ mod \ N = m \qquad (2.2)$$

[11]Note that this is the same as formula 2.4.1 with exchanged public and private keys.

2.4.3 Certificates

For asymmetric cryptography it is essential to know that a public-key really belongs to a specific participant and was not published by an attacker who pretends to be someone else (man in the middle attack).

A *Certificate* binds a public-key to an identity (person, organization or IT system).
A *Certificate Authority (CA)* issues a *certificate* $CA\langle\langle A\rangle\rangle$ to A which guarantees that the CA has verified the correct binding between the public-key A_p of A and its identity:

$$CA\langle\langle A\rangle\rangle := CA\{UCA, UA, A_p\}$$

where UCA is the unique name of the CA, UA the unique name of A and A_p the public-key of A.
A certificate can be compared to a passport that binds a picture (public-key) to a person (identity) and can only be issued by a Passport Service (CA) that guarantees the correct binding by being the only institution to issue correct passports (signatures with CA's secret-key).

In X.509 the format of certificates is standardized by the International Telecommunication Union (ITO-T) [ITU05].

A CA can also issue a certificate to another CA so that *certificate hierarchies* or *certificate trees* can be set up. The root CA of a certificate tree has a self-signed *root certificate* (CA0 in Fig 2.9).

To verify a certificate in a certificate tree the certificate tree is climbed up and each certificate up to the root certificate is verified. If one trusts the root certificate (the public-keys of trusted root certificates are compiled into an application for example) and all certificates in the verified certificate chain are valid, the leaf certificate is valid, too.
To verify the certificate of B in Fig. 2.9, one verifies these certificates in the given order:
$CA0.1.2\langle\langle B\rangle\rangle, CA0.1\langle\langle CA0.1.2\rangle\rangle, CA0\langle\langle CA0.1\rangle\rangle, CA0\langle\langle CA0\rangle\rangle$

Each participant wanting others to be able to verify his certificate stores all certificates from his certificate up to the root certificate and hands them out on request.

2.4.4 Authenticated Key Exchange

In an *authenticated key exchange protocol* two parties authenticate each other (to ensure that each party really communicates with the intended other party and not with an attacker who pretends to be the intended party) and exchange a common secret key.

In a *challenge-response protocol* B authenticates to A by signing a nonce received from A with his secret-key B_s and sending the signature back to A. This is used in the following

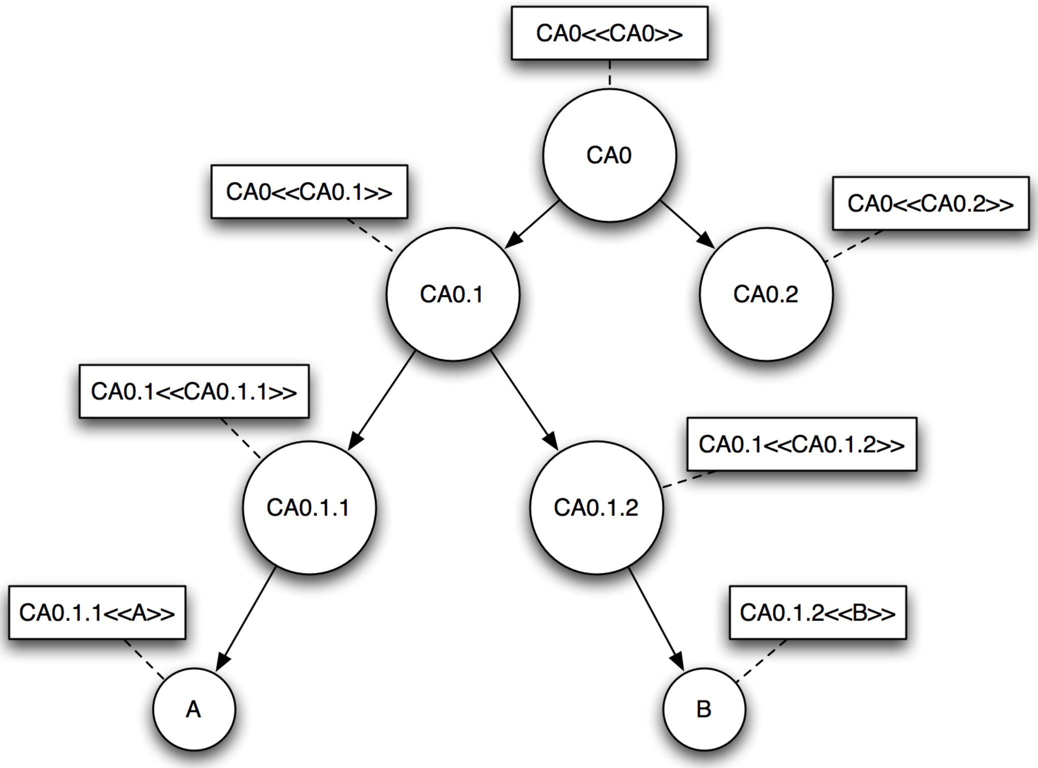

Figure 2.9: *Certificate tree with CA0 as root*: A hierarchic certificate hierarchy can be built by signing sub certificates. A certificate authority (e.g. CA0.1) signs the certificate of a sub certificate authority (e.g. CA0.1.1.2).

protocols.

A *nonce* (number used once) is a non-repeating number, which is used to detect *replay attacks*[12] in cryptographic protocols. To ensure that it is used only once, it should be a strictly increasing sequence, time dependent and/or contain enough pseudo-random bits to ensure a probabilistically insignificant chance of repeating a previously used value.

Two parties can authenticate each other and exchange a common symmetric-key by using two runs of a challenge-response protocol and an asymmetric cipher like in the *three-way authentication protocol* also called *three-way handshake* specified in [ITU05, 18.2.2.3].

The total number of public-key operations needed for authentication and the exchange of one key using this scheme is 8: *A* signs twice, verifies once and encrypts once whereas *B* verifies twice, signs once, and decrypts once.

For the RECONETS security layer a faster but equally computational secure three-way au-

[12] an attacker records a message and replays it later

thenticated key exchange protocol is implemented where the number of expensive public-key operations is reduced to 6: A signs twice and encrypts once whereas B verifies twice and decrypts once:

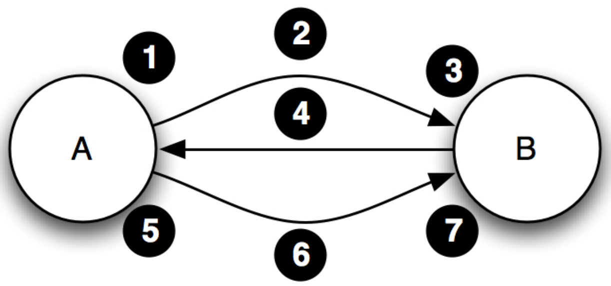

Figure 2.10: *Fast three-way authenticated key exchange protocol*:
1) generate nonce r_A and session key
2) send asymmetrically signed nonce and encrypted session key
3) verify signature, decrypt session key and generate nonce r_B
4) send symmetrically signed nonces
5) verify signature and nonce r_A
6) send asymmetrically signed nonce
7) verify signature and nonce r_B

Alice and Bob want to authenticate each other and exchange a common secret-key to symmetrically sign messages K_{AB} with the *fast three-way authenticated key exchange protocol* shown in Fig. 2.10:

1. A computes:
   - get B's certificate and all certificates needed to verify it (if she doesn't already have them, she asks B for them), verifies them and extracts B's public-key B_p out of B's certificate.
   - generates a random nonce r_A
   - generates a session key $K_{AB} = (E_{AB}, M_{AB})$ consisting of a random symmetric encryption key E_{AB} and a random symmetric MAC key M_{AB}.
2. A sends the following message to B:

$$C_A, A\{r_A, B, B_p[K_{AB}]\}$$

   where C_A is A's certificate and all certificates that B needs to verify this.
3. B computes:

- verifies A's certificate using C_A and extracts A_p out of it
- checks that B itself is the intended recipient
- verifies A's signature
- optionally, checks that r_A has not been replayed
- decrypts $B_p[K_{AB}]$ with his secret-key: $K_{AB} = B_s[B_p[K_{AB}]]$
- generates a random nonce r_B

4. B sends the following message to A:

$$M_{AB}\{r_B, A, r_A\}$$

5. A computes:
   - checks that A itself is the intended recipient
   - verifies B's signature
   - optionally, checks that r_B has not been replayed
   - checks that the received r_A is identical to the r_A sent before

6. A sends the following message to B:

$$A\{r_B, B\}$$

7. B computes:
   - checks that B itself is the intended recipient
   - verifies A's signature
   - checks that the received r_B is identical to the r_B sent before

The Internet Key Exchange protocol (IKE) [HC98] is used in Virtual Private Networks (VPN) as a standard for authenticated key exchange.
It works in two phases: The first phase authenticates the two parties to each other by a three-way authentication protocol and exchanges a common session key for further agreement on temporary-keys. The second phase is periodically scheduled and exchanges temporary-keys for encryption and integrity of the communication by using the session key which was exchanged in the first phase.

3 Conceptual Design of a Security Architecture for a ReCoNet

In order to achieve the security objectives of a ReCoNet as secure task-migration and interprocess communication described in section 1.1, a security architecture has to be designed. As the protection of a single, reconfigurable, embedded system has already been investigated before (see section 1.2), the security architecture for a ReCoNet focusses on the security requirements of a ReCoNet as a distributed system. The security requirements for a single reconfigurable, embedded system are formalized in four security prerequisites. All these prerequisites can be achieved with today's FPGAs as described below.

3.1 Security Prerequisites for the System

These four prerequisites of the reconfigurable, embedded system are assumed:

Pre1 *Secret-Key storage*: A small secret can be stored confidentially, non-cloneably and tamper-resistantly in the system.

Pre2 *Tamper-resistant configuration*: The system can detect modifications of a hardware module on startup.

Pre3 *Secure Hardware*: During operation confidentiality and integrity of the hardware are ensured.

Pre4 *Secure Memory*: During operation confidentiality and integrity of the memory are ensured.

How these prerequisites can be provided by SRAM-based FPGAs supporting bitstream encryption[1] is described in the following. Both FPGA families that are currently used in the ReCoNets project support bitstream encryption: Xilinx Virtex-II provides bitstream encryption with 112 bit 3-DES and Altera with 128 bit AES.

Systems with SRAM-based FPGAs contain on-chip and external memory for data and configuration data with different levels of trust (Fig. 3.1). An attacker can easily read and modify the contents of untrusted components external to the FPGA chip but not of trusted components inside the FPGA.

[1] as described in "Cloning of SRAM FPGAs" in section 1.2

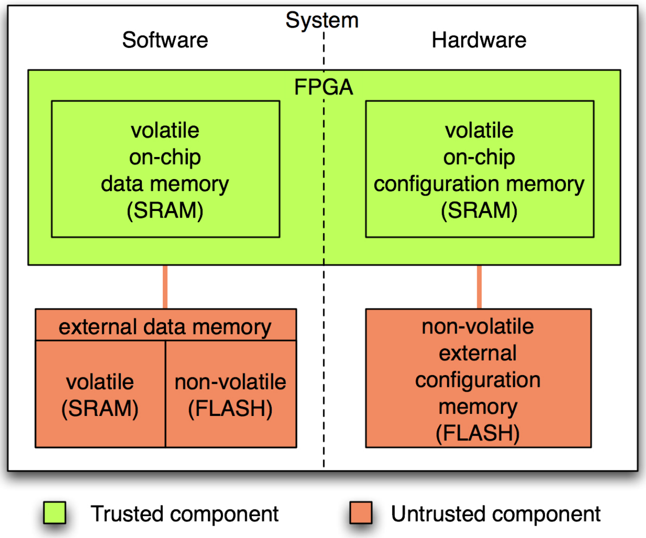

Figure 3.1: *Trust model of SRAM-based FPGA boards*: External data memory (volatile and non-volatile) and external configuration memory (non-volatile) are untrusted components as an attacker can tap into the connection wires (red). The FPGA and on-chip memory are trusted (green).

- If the bitstream is encrypted[1], an attacker has no chance to tap the configuration during transmission from the external configuration memory to the FPGA. In addition, this prevents a reverse engineering of the circuit or any kind of bitstream manipulation. Based on this, Pre1 and Pre2 can be provided.

- In the following it is assumed that an attacker can neither observe nor modify the hardware in the FPGA chip after configuration (trusted component): blackbox , readback , physical and side-channel attacks can be prevented by using countermeasures as described in section 1.2. This provides Pre3.

- All on-chip memory is secure memory as postulated in Pre4. Additionally, external memory can be used as secure memory by signing and encrypting data as described in [SCG+03].

3.2 Security Architecture for the ReCoNet

In order to provide the security objectives described in the beginning of this thesis (Section 1.1), a security architecture for the ReCoNet is designed. The security architecture is based on the previously described security prerequisites and is partitioned into the hard- and software modules shown in (Fig. 3.2):

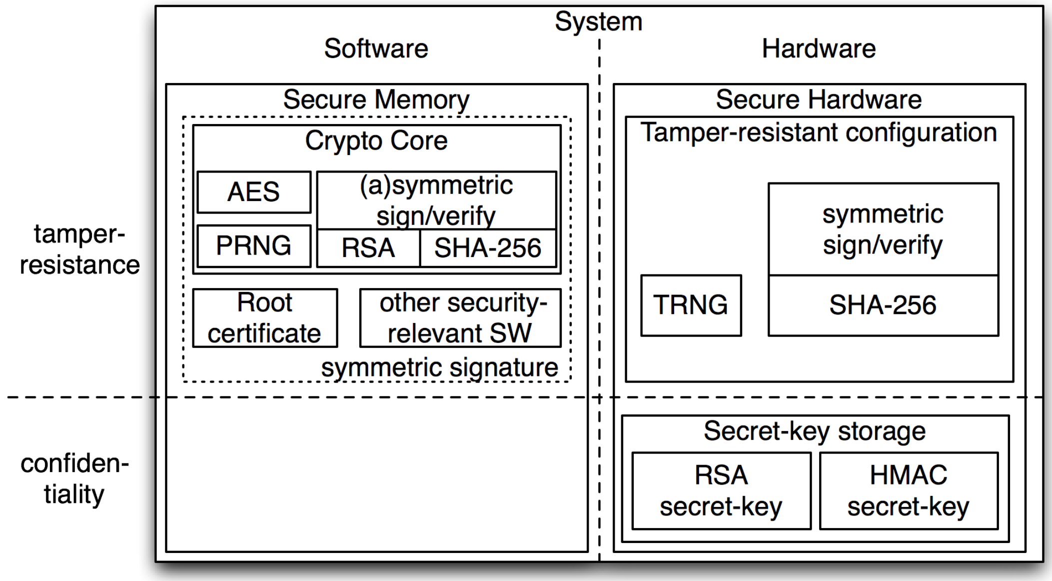

Figure 3.2: Security Architecture for the ReCoNet partitioned into Software (Crypto Core providing cryptographic algorithms, Root certificate to verify digital signatures) and Hardware (Secret-key storage, True Random Number Generator (TRNG) for random numbers that are i.e. needed in cryptographic protocols, and a module to verify the symmetric signatures of software modules (SHA-256))

3.2.1 Hardware Modules

Secret-Key Storage

Each node has an individual asymmetric *RSA secret-key* to identify itself to the rest of the system in a challenge-response protocol (see 2.4.4). Each attacker who learns the secret-key can identify himself as the node whose key was stolen. So the secret-key must be tied to the hardware of the node in a confidential and uncloneable way (Pre1) and must not leave the node.
In addition each node stores an individual *HMAC secret-key* in the Secret-key storage module (see next subsection).

Tamper-Resistant Configuration

These two modules are stored as a tamper-resistant configuration (Pre2):

- A hash module for *SHA-256* together with a corresponding *HMAC*- and a *symmetric signature generation/verification* module can be implemented completely in hardware to verify the symmetric signature of security relevant software modules on system startup by using the HMAC secret-key stored in the secret-key storage module.

- As cryptographic protocols and the padding for asymmetric ciphers (see next subsection) need random numbers, a *true random-number generator* (TRNG) has been implemented in hardware which seeds the *pseudo random-number generator* (PRNG) running in software.

3.2.2 Software Modules

Security relevant modules consisting of *Crypto Core*, *Root certificate* and *other security relevant software* (e.g. operating system kernel, protocol stack, task resolution) are symmetrically signed with the HMAC secret-key stored in the secret-key storage before loading in the system's non-volatile external data memory (FLASH). On power-up, the security relevant modules are transfered from non-volatile memory into secure memory and thereafter the symmetric signature is verified by the verification module implemented in hardware. Only if the signature is valid, the security relevant modules were not tampered and the system continues booting.

Crypto Core

The *Crypto Core* provides cryptographic primitives like *RSA*, *SHA-256*, *asymmetric signature generation/verification*, *AES* for symmetric encryption and a *PRNG* (Pseudo Random Number Generator) in software and is stored in a tamper-resistant way by symmetrically signing it as described before.
The PRNG ensures that the produced random numbers ”look” randomly distributed even if the underlying TRNG (True Random Number Generator) is not perfect and the PRNG is much faster than the TRNG. By combining a PRNG and a TRNG only very few true random numbers are needed.

Root Certificate

The *root certificate* is the root of the certification hierarchy and is required for verification of certificates as described in 2.4.3. Everybody is allowed to read it (as it only contains a public key and some information about the issuer of the root certificate) but it must be ensured, that the root certificate stored in a node cannot be replaced by an attacker. Thus, the root certificate is stored tamper-resistantly.

3.3 Digital Rights and Certificates

The following section describes how *digital rights* (DR) based on *Certificates* can be added to the ReCoNet in order to guarantee that each node can only execute special trustworthy tasks.

Each node is allowed to run special kinds of tasks only. What tasks a node is allowed to run might depend on its connected periphery (e.g. specific sensors or actors), the reliability of its hardware (a node with a high probability of failure won't be allowed to run critical tasks like steer-by-wire), its performance, cost, and more application-specific factors.

3.3.1 Encoding of Digital Rights

Digital rights determine if a task T is allowed to run on a node N. They should be encoded in a generic way to allow arbitrary complex levels of digital rights.

The following scheme encodes digital rights D in conjunctive normal form (CNF) consisting of N clauses C_i that contain L_i literals $V_{i,j}$:

$$D(N) = \bigwedge_{i=1}^{N} \bigvee_{j=1}^{L_i} V_{i,j} \tag{3.1}$$

Each digital right D consists of N *digital right vectors* (DRV) V_i with L_i bits where the j-th bit represents the literal $V_{i,j}$ of the corresponding clause.

In order to check whether digital right D_1 is a subset of D_2 ($D1 \sqsubseteq D_2$), D_1 must have at least as many digital right vectors as D_2, each digital right vector must have the same length and the 1-bits of D_1 must be a subset or equal to the 1-bits of D_2:

$$D_1 \sqsubseteq D_2 \Leftrightarrow (D_1.N \geq D_2.N) \wedge (\forall_{1 \leq i \leq D_2.N}(D_1.L_i = D_2.L_i) \wedge (D_1.V_i \,\&\, \overline{D_2.V_i} \neq 0)) \tag{3.2}$$

Thus, a digital right can be narrowed by adding more clauses (add additional conditions) or switching off some bits in existing clauses (narrow existing conditions).

Practical examples for the encoding of digital rights will be shown in section 3.5.

3.3.2 Certificates

Certificates bind public-keys and digital rights to nodes, tasks and manufacturers. Each certificate contains a unique ID to determine its predecessors in the certification hierarchy as described in 2.4.3 and a name that identifies the owner of the certificate.

Certified Nodes

The *permissions* (digital rights) of a *certified node* specify which groups of tasks the node is allowed to execute. On production of a node, the hardware *manufacturer* generates an asymmetric-key pair (e.g. a RSA key pair as described in 2.4.1) for each node and stores the generated secret-key in the node as described in 3.1. He creates a *node certificate* containing the generated public-key (*PubKey*), the node's *permissions* and the serial-number (S/N) of the node by signing it with the manufacturer's secret-key (Fig. 3.3). The node certificate is public and will be stored in the node, too.

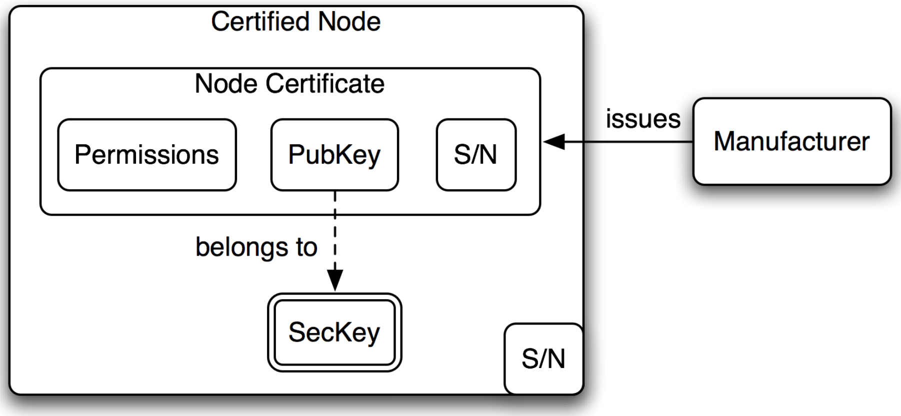

Figure 3.3: Certified Node

Signed Tasks

Security critical tasks that can migrate through the ReCoNet are called *signed tasks*. A software *manufacturer* that has developed, verified and tested a security critical task, attaches the needed *requirements* (digital rights) of the task to it and signs them along with its *binary* with the manufacturer's secret-key (Fig. 3.4).

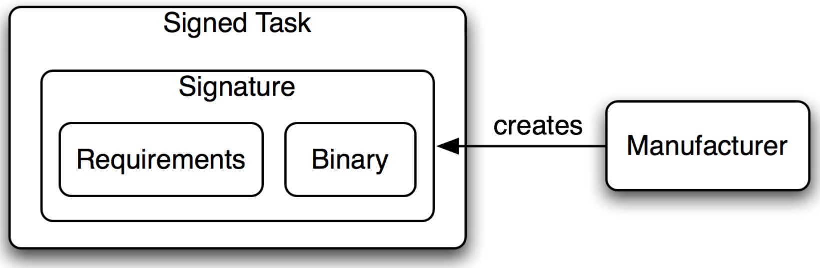

Figure 3.4: Signed Task

Manufacturers

Each *manufacturer* holds besides his secret-key (SecKey) a certificate that binds the corresponding *PubKey* to the manufacturer's name and specifies, what *permissions* (digital rights) the manufacturer has, i.e. what kinds of digital rights for certified nodes/signed tasks the manufacturer is allowed to grant (Fig. 3.5).

Each manufacturer M can delegate a subset of his permissions to sub-manufacturers S by issuing certificates to them. This creates a certification hierarchy of manufacturers as described in 2.4.3.

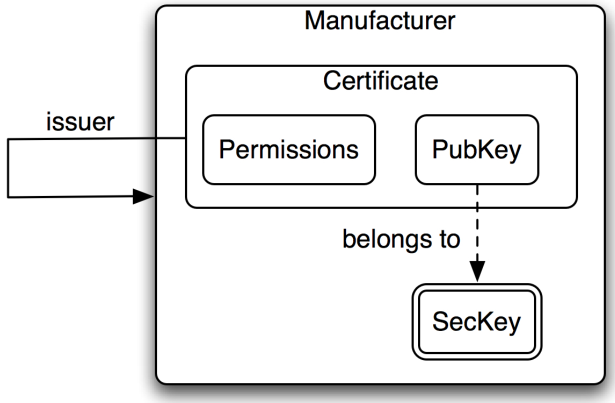

Figure 3.5: Certificate Hierarchy

A sub-manufacturer S must have a subset of the permissions of its issuer M:

$$S.permissions \sqsubseteq M.permissions \tag{3.3}$$

A manufacturer M can only certify nodes N with:

$$N.permissions \sqsubseteq M.permissions \tag{3.4}$$

A manufacturer M can only sign tasks T with:

$$T.requirements \sqsubseteq M.permissions \tag{3.5}$$

3.3.3 Verification of Certificates

The function `checkNodeCert(checkTaskSig)` verifies the certificate(signature) of a certified node (signed task):

- The certificate (signature) of the node (task) and equation 3.4 (3.5) are checked.
- Afterwards the software verifies the chain of manufacturers' certificates to the root certificate that is implicitly trusted by all participants of the ReCoNet. Besides verifying the manufacturers' signatures in the certificates it must be checked, that the digital rights of each sub-manufacturer S don't exceed the permissions of its issuer M (equation 3.3).

On power-up of a node, the node does a self-check (checkSelf) of its certificate and its stored secret-key to ensure, that the certificate is valid, not modified (checkNodeCert) and matches the secret-key (`checkKey`).

These certificates are the basis for a secure task migration.

3.4 Secure Task Migration

A task T is allowed to run on a node N only if the following conditions hold:

1. The requirements t of T must match the permissions n of N:
   Each of their Digital Right Vectors must have at least one common bit.

$$match(t,n) \Leftrightarrow (t.N \leq n.N) \wedge (\forall_{1 \leq i \leq t.N}(t.L_i = n.L_i) \wedge (t.V_i \,\&\, n.V_i \neq 0)) \quad (3.6)$$

2. T's signature must be valid.

3. N's certificate must be valid.

4. The certificates of N and T must have a common predecessor P in the certification hierarchy that has a certificate with exactly $t.N$ Digital Right Vectors. This ensures semantic integrity of N's and T's Digital Right Vectors.

Before a task T is started on node N, the node checks whether T is allowed to run on N. The verification of the task's signature also ensures that the task was not modified during submission - neither incidentally by a bit-failure nor intentionally by an attacker.

Before a node A migrates a task T to another node B of the ReCoNet A performs these actions:

1. retrieve B's certificate [2]

2. authenticate B by a three-way handshake (2.4.4) [2]

3. check whether B is allowed to run T.

A will migrate T to B only if B was authenticated correctly and is allowed to run T.

This scheme also allows the secure update in-field of the system:

- In an *offline-update scenario*, a signed task is deployed to a node of the ReCoNet by a data medium[3]. Each node of the ReCoNet can verify the authenticity and integrity of the signed task as described in chapter 3.3.3.

- In an *online-update scenario*, an update server wants to deliver a signed task confidentially to a node of the ReCoNet via a public network like the internet:
  First the update server and the node authenticate each other and exchange a symmetric session key K by an authenticated key exchange as described in chapter 2.4.4. The update server does not necessarily need a certificate that the node can verify - if he has none, the three-way handshake is replaced by a two-way handshake where only the node signs a nonce. After that, the server encrypts the signed task symmetrically with K and sends it to the node. This decrypts the task with K and verifies the task's authenticity and integrity by checking the task's signature (`checkTask`).

[2] if not already done before
[3] e.g. USB Stick or Compact Flash Card

3.5 Practical Examples for Digital Rights

The proposed scheme for digital rights can be used to express many conditions as digital rights. Some examples that appear often in practical contexts are shown in the following subsections. Every concept uses just one DRV and they can be combined by composing the corresponding DRVs to a digital right D. All conditions of D must be fulfilled.

3.5.1 Classes of Hardware Requirements

A task might require special hardware of one type. Different types t_i $(1 \leq i \leq n)$ of related hardware are grouped together to a class $C = \bigcup_{i=1}^{n} t_i$.
The corresponding DRV $V = v_1 v_2 \ldots v_n v_{n+1}$ where $v_i = 1$ $(1 \leq i \leq n)$ means that a node provides t_i respectively a task requires t_i. v_{n+1} is used to ignore this hardware class as follows:

All nodes have a DRV V with v_i $(1 \leq i \leq n)$ set corresponding to their hardware equipment and $v_{n+1} = 1$.
A node that provides none of the hardware in that class has $V = 0^n 1$.

A task that requires any of the types has a DRV V with v_i $(1 \leq i \leq n)$ set corresponding to the hardware he can use and $v_{n+1} = 0$.
A task that does not require any of the hardware has $V = 1^n 1$.

As an example let the class C be cameras, t_1 is a black-and-white camera and t_2 is a color camera. Task T_0 requires any camera, T_1 requires camera t_1, T_2 camera t_2 and T_3 does not require any camera at all. Node N_0 provides no camera, N_1 camera t_1 only, N_2 camera t_2 only and N_3 both cameras.

$N.V \& T.V \rightarrow$ can run?	$\mathbf{N_0 : 001}$ no camera	$\mathbf{N_1 : 101}$ camera t_1	$\mathbf{N_2 : 011}$ camera t_2	$\mathbf{N_3 : 111}$ both cameras
$\mathbf{T_0 : 110}$ require any camera	000 → *no*	100 → *yes*	010 → *yes*	110 → *yes*
$\mathbf{T_1 : 100}$ require camera t_1	000 → *no*	100 → *yes*	000 → *no*	100 → *yes*
$\mathbf{T_2 : 010}$ require camera t_2	000 → *no*	000 → *no*	010 → *yes*	010 → *yes*
$\mathbf{T_3 : 111}$ ignore camera	001 → *yes*	101 → *yes*	011 → *yes*	111 → *yes*

Table 3.1: *DRV example - Classes of Hardware Requirements*: A task can only run on a node, if the hardware requirements are fulfilled, i.e. the AND of their corresponding digital right vectors (DRV) is not zero.

Table 3.1 shows the corresponding DRVs of the tasks and whether they are allowed to run on the given nodes.

A hardware manufacturer that is allowed to produce nodes that provide specific hardware types has a DRV V with the corresponding v_i $(1 \leq i \leq n)$ set to 1 and $v_{n+1} = 1$. If he is only allowed to produce nodes that do not provide any hardware of this class, he has $V = 0^n 1$.

If the tasks of a software manufacturer must use any special hardware, he gets v_i $(1 \leq i \leq n)$ set accordingly and $v_{n+1} = 0$. If the software manufacturer is allowed to sign tasks that ignore the class, he gets $V = 1^n 1$.

3.5.2 Reliability Level

Tasks might depend on a certain level of reliability of the hardware. For example the task that controls the airbag in an automobile must only be run on very reliable hardware. If there are n levels of reliability with 1 being the lowest and n being the highest level of reliability the DRV is composed as described in table 3.2:

Description	**DRV clause** V	**Example**
A manufacturer M that is allowed to create nodes or tasks up to reliability level K $(1 \leq K \leq n)$ has the permissions	$1^K 0^{n-K}$	111110
M can allow a sub-manufacturer M' to create nodes or tasks of a smaller reliability level $K' \leq K$ by issuing a manufacturer certificate with permissions	$1^{K'} 0^{n-K'}$	111100
A node N with a reliability level of k that is issued by M' $(1 \leq k \leq K')$ has the permissions	$1^k 0^{n-k}$	111000
A task T that depends on a reliability level of at most k' and is signed by manufacturer M' $(1 \leq k' \leq K')$ is allowed to run on N if $k' \leq k$. It has the requirements	$0^{k'-1} 1 0^{n-k'}$	010000
T is allowed to run on N as $T.V \& N.V \neq 0$	$0^{k'-1} 1 0^{n-k'}$	010000

Table 3.2: *DRV example - Reliability Level*: A thermometer code is used to encode different levels of reliability into a digital right vector (DRV).

3.6 Secure Interprocess Communication

To allow *secure interprocess communication*, the existing communication stack of the ReCoNets [KSD+06] is transparently extended with a security-layer similar to IPsec [DH03].

After two nodes A and B have exchanged a symmetric session key K_{AB} with an authenticated key exchange protocol (2.4.4) they can symmetrically sign messages M with a HMAC (2.2.4) to ensure their integrity and authenticity: $K_{AB}\{M\}$ is sent from A to B.

The ReCoNets communication architecture for intertask communication as described in [Dit05, Chapter 3] is designed according to the OSI layer model [Zim80] and consists of several layers that are shown in Fig. 3.6 and described in Table 3.3.

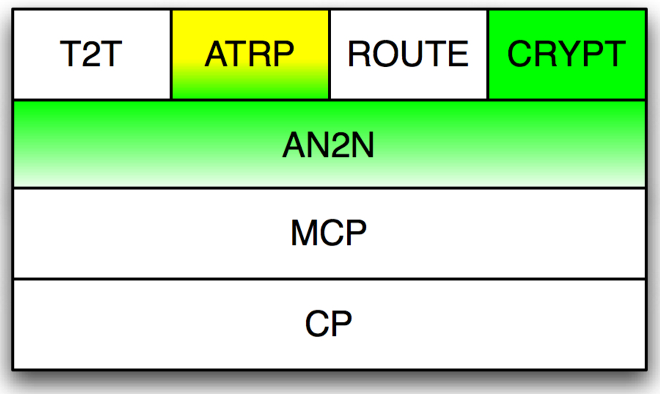

Figure 3.6: *Security extensions of the* ReCoNets *protocol stack*: AN2N signs and verifies messages symmetrically (extended), CRYPT exchanges a symmetrical signature key (completely implemented) and ATRP verifies the matching between node certificates and task signatures (basic functionality implemented). The remaining protocol layers are unchanged.

Layer	Description
CP (Cell Protocol) [Dit05, 3.3]	Transfer data cells of fixed size between adjacent nodes.
MCP (Multi Cell Protocol) [Dit05, 3.6]	Transfer packets of variable size between adjacent nodes.
N2N (Node To Node Protocol) [Dit05, 3.7]	Transfer messages of variable size between two nodes including an acknowledge mechanism to resend defect packets (reliable transport protocol).
ROUTE (Route Protocol) [Dit05, 3.8]	Synchronize routing tables used in Dijkstra's algorithm to determine routes.
TRP (Task Resolution Protocol) [Dit05, 3.9]	Determine mapping of tasks to nodes and task migration.
T2T (Task To Task Protocol) [Dit05, 3.10]	Send messages between tasks independent on which node it is currently executed (inter process communication).

Table 3.3: Layers of the RECONETS communication architecture

This communication architecture can be extended with the required security aspects as shown in Table 3.4:

Layer	Description
CRYPT (Crypto Protocol)	Protocol for authenticated key exchange between two nodes as described in section 2.4.4.
AN2N (Authenticated N2N)	Extension of the N2N protocol to sign and verify all messages with the symmetric key exchanged by CRYPT as described in 2.2.3.
ATRP (Authenticated TRP)	Extension of TRP to verify the task requirements against the node permissions before task migration as described in 3.4.

Table 3.4: Modified layers of the secure RECONETS communication architecture

All protocols that are based on the AN2N protocol - particularly T2T for task to task communication - will transparently be signed and thus protected against intentional modifications.

4 Implementation and Integration of the Security Layer into the ReCoNet

This chapter describes how the existing RECONETS demonstrator was extended and how the hard- and software modules of the security architecture for the ReCoNet described in Section 3.2 are implemented in detail.

4.1 Hardware Modules

The following Hardware modules were implemented for the current RECONETS hardware architecture consisting of Altera Cyclone EP1C20F400C7 FPGAs and integrated into the existing Altera Quartus II Project.

4.1.1 True Random Number Generator (TRNG)

As the RECONETS will be migrated from Altera to XILINX FPGAs, a generic TRNG is provided. It is written in VHDL and independent of manufacturer and FPGA families.
In [KG04] the authors describe how to extract true randomness out of the jitter of two free running oscillators. As free running oscillators they use a circuit that is hard-wired into one Configurable Logic Block (CLB) of a Xilinx Virtex XCV1000 FPGA. The technique for extraction of randomness described in [KG04] is generalized to be used on any FPGA by implementing generic oscillators that do not depend on special assumptions on logic blocks:

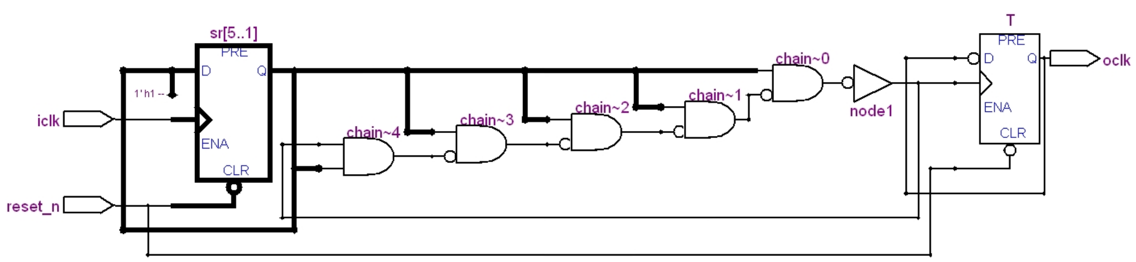

Figure 4.1: *Generic oscillator with 5 gates delay*: A shift register is used in order to prevent optimization of the chain of 5 NAND gates.

As shown in Fig. 4.1 we use a chain of 2-input NAND gates where one input of each NAND gate is connected to a shift-register. The shift-register initially contains zeroes and is filled with a one on each clock cycle after reset. This avoids that the chain of inverters is optimized away from a synthesis tool. The number of delay gates must be odd and can be set as a VHDL generic.

Our complete TRNG (crypt_trng_0) with a delay of 11 gates needs 182 Logic Cells on the current RECONETS system. It includes a van Neumann corrector to ensure that the output of the TRNG is unbiased as described in [KG04]. A van Neumann corrector takes two successive output bits i_{2k} and i_{2k+1} of the possibly biased (it might produce more zeroes as ones or the other way around) TRNG and outputs an unbiased value o_k as shown in Tab. 4.1:

i_{2k}	i_{2k+1}	o_k
0	0	nothing
0	1	0
1	0	1
1	1	nothing

Table 4.1: *Output of a van Neuman corrector*: discard successive equal values in order to unbias the input.

The TRNG is connected to the Nios II-CPU as a memory mapped input that returns a new 32 bit random number or 0 if the new random number is not yet available.

4.1.2 Secret Key Storage

The secret key storage is implemented as a hardware ROM that contains up to four secret keys. As the current RECONETS demonstrator consists of four nodes, each node can use the same version of the secret key storage hardware module. The corresponding secret key for the node is selected depending of the node ID of the node in software. The secret key storage (crypt_secrom_0) for 4 kByte of secret ROM needs 2648 logic cells for four keys.
In real scenarios every node must have only one secret-key of course. With a more compressed data format for the secret key, its storage could be reduced to 0.5 kByte per node which would result in about 350 logic cells.
The secret key ROM is directly mapped into the memory of the Nios II-CPU (read-only).

4.1.3 SHA-256

The hardware SHA-256 module can be used to verify the symmetric signature of the crypto core, the root certificate and possibly the whole OS on system startup as described in 3.2.1. As a proof-of-concept the SHA-256 module was implemented and tested on the Altera FPGAs of the ReCoNets. A synthesis of the module for the Xilinx based Erlangen Slot Machine (ESM) [BMA+05] resulted in 1668 slices. This module could be integrated into the new ESM-based ReCoNets demonstrator in further works.

4.2 Software Modules

Most of the functionality of the security architecture for the ReCoNets is implemented in software within the crypto core. The software of the ReCoNets demonstrator consists of C and C++ code that runs on a Nios II softcore CPU implemented on the FPGA and uC/OS-II as the underlying operating system.

4.2.1 Crypto Core

The needed cryptographic standard algorithms for the software crypto core are ported from *Libgcrypt*, a standard cryptographic software library [Gnu] and were ported to the Nios II CPU and uC/OS-II for use in the ReCoNets as part of this thesis. A detailed description of the libgcrypt API can be found in [KS05].
In particular, the following cryptographic primitives whose functionality has already been introduced in Chapter 2, are now available and tested in the extended ReCoNets demonstrator:

- *cryptographic random number generator*: Pseudo Random Number Generator - (PRNG) with RIPEMD-160 as hash function based on the hardware True Random Number Generator (TRNG) described in sections 4.1.1 and 3.2.2.
  API described in [KS05, Chapter 9: Random Numbers].
  Random numbers are needed during authenticated key exchange (section 2.4.4) and for the padding of asymmetric ciphers (see last item).

- *cryptographic hash function*: SHA-256
  API described in [KS05, Chapter 6: Hashing, GCRY_MD_SHA256].
  SHA-256 is described in section 2.2.2 and used as the cryptographic hash function in digital signatures (section 2.4.2) needed for certificates and task signatures.

- *symmetric signatures*: HMAC-SHA-256
  API described in
  [KS05, Chapter 6: Hashing, GCRY_MD_SHA256 and GCRY_MD_FLAG_HMAC].
  HMAC-SHA-256 is described in section 2.2.4 and used for symmetric signatures of messages (section 3.6).

- *symmetric cipher*: AES
  API described in
  [KS05, Chapter 5: Symmetric cryptography, GCRY_CIPHER_AES128].
  AES is described in section 2.3.1 and was ported for further works that might use symmetric encryption like encrypted messages or a two-phase authenticated key protocol (section 4.2.3).

- *asymmetric cipher and asymmetric signatures*: RSA with PKCS #1 padding
  API described in [KS05, Chapter 7: Public Key cryptography (I)].
  RSA is described in section 2.4.1 and used for digital signatures (section 2.4.2) in certificates, task signatures and authenticated key exchange (section 2.4.4). The authenticated key exchange protocol also uses asymmetric encryption with RSA (section 2.4.1).
  PKCS #1 [RSA02] is a standard that describes how to use RSA for asymmetric encryption of messages with arbitrary length and in particular how to fill the last block of a message with random numbers (padding).

4.2.2 Certificates and Digital Rights

Certificates and digital rights are stored as *S-expressions* (Symbolic expressions), a human readable data format in which asymmetric keys are stored in libgcrypt ([KS05, Chapter 10: S-expressions]). Libgcrypt contains functions for creation, manipulation and extraction of S-expressions (gcry_sexp_*).
The verification of certificates is implemented as described in section 3.3.3 and contains a *caching mechanism* that ensures that each certificate and signature is verified only once as the underlying digital signature operations are very expensive.
Appendix A.2 contains an example for manufacturer certificates, node certificates and task signatures.

Tools to create (create_*), check (check_*), and display (sexp) manufacturer certificates (*manufacturer) , node certificates (*node) and task signatures (*task) on a UNIX based host system (i.e. Linux, Solaris, MAC OS) are available as described in Appendix A.1 and the documentation (-help).

4.2.3 Authenticated Key Exchange

After two nodes in the ReCoNets have discovered each other (triggered by the routing protocol ROUTE of the ReCoNets protocol stack), they exchange a common symmetric key with the CRYPT protocol. This protocol implements the *fast three-way authenticated key exchange protocol* described in subsection 2.4.4. It currently has only one phase and uses the exchanged key directly to sign the messages. An example trace of the exchanged messages is contained in Appendix A.2.

The implemented CRYPT-protocol ensures that if one node fails (i.e. because of a system reboot) a new key is exchanged when the two nodes discover each other again whereas on link failures both nodes keep the previously established key to resume as fast as possible.

The protocol can easily be extended to a *two-phase key-exchange protocol* as described in subsection 2.4.4 to improve the security of a ReCoNets with long up-times (all needed algorithms are already included in the crypto core):
In the *first phase* a session key consisting of a 128 bit AES key K_{AB}^{AES} and a 256 bit

HMAC-SHA-256 key K_{AB}^{HMAC} is exchanged with the implemented *fast authenticated key exchange protocol*.
The periodically scheduled *second phase* exchanges the temporary key to sign the messages as follows: A generates a random temporary key, symmetrically encrypts it with K_{AB}^{AES}, signs the message with K_{AB}^{HMAC} symmetrically, and sends this signed and encrypted key to Bob. Bob verifies the signature with K_{AB}^{HMAC} and on success decrypts the temporary key with K_{AB}^{AES}.

4.2.4 Secure Interprocess Communication

After two nodes have exchanged a common secret key by using the CRYPT-protocol all messages are signed symmetrically in the modified AN2N-layer. Thus all protocols that are above the AN2N-protocol (except ROUTE and CRYPT) are secured - including T2T, the protocol for - now secure - task to task communication.

For every N2N-message the sender computes the symmetric signature value over all fields in the N2N-header and the N2N-message with the common key in software and appends it to the N2N-header.
The receiver of a N2N-message verifies the symmetric signature and asks the sender to resend the packet if it was intentionally modified.

To reach full computational security all 32 bytes of the signature value have to be appended to the N2N-header which increases the size of each N2N-message dramatically (the rest of the N2N-header consists of only 10 bytes). If a smaller security level suffices for the system, only the first N bytes of the signature can be transmitted and verified.
The security parameter N is currently global and is fixed on compile-time (constant CRYPT_SIGBYTES).

The N2N layer also ensures that *transmission errors* of N2N-messages are detected - in this case the sender is asked to resend the packet. As transmission errors are more likely than intentional attacks, the previously included 2 byte CRC checksum is kept in the header of the N2N-messages as the CRC verification is much cheaper than the verification of the symmetric signature[1]:
The *sender* first computes the symmetric signature of the message as described before, appends it to the header and then appends the CRC checksum of the whole N2N-message (including the symmetric signature).
The *receiver* first verifies the CRC checksum. If it is wrong, a bit failure was detected and the message has to be resent. Otherwise he verifies the symmetric signature to detect intentional attacks on the message.

[1]Note that the symmetric signature would be able to detect transmission errors, too.

4.2.5 Secure Task Migration

The matching of node certificates and task signatures is implemented as described in subsection 3.4 to determine whether a task T can run on a node N: check_run(T,N).

On system startup all node certificates and task signatures that are statically compiled into the system are loaded and verified. Thereafter a matching table is printed that shows what task can run on a node or its transposition what tasks a node is allowed to execute (see Appendix A.3).

This matching functionality could easily be integrated into the task resolution protocol (TRP) to allow only the execution and migration of signed tasks that match to the node certificate as described in subsection 3.4. This would result in an authenticated task resolution protocol (ATRP).

As the current RECONETS demonstrator only allows tasks that are known to the system at compile-time the secure off- and online update functionality can currently not be demonstrated. It could have been added, if the system had an MMU (Memory Management Unit) and the operating system would support dynamic loading of processes that are not known on compile time ([Dit05, Chapter 4]).

4.2.6 Total Costs of the Implemented ReCoNets Security Layer

The costs for the implemented security layer are shown in Table 4.2 and Table 4.3:

HW module	absolute size	relative size
RECONETS demonstrator (Audio-node) including security layer modules	13195 LUTs	100.0%
True Random Number Generator	182 LUTs	1.4%
4 kByte secret KEYROM	2648 LUTs	20.1%

Table 4.2: *Hardware costs of the* RECONETS *security layer*: The costs for the KEYROM can be reduced to approximately 331 LUTs (3.0%).

SW system	absolute size	relative size
RECONETS with no crypto support (CRYPT=0)	287 kByte	100%
RECONETS with crypto support (CRYPT=1)	525 kByte	183%

Table 4.3: Software costs of the RECONETS security layer

As described in section 4.1.2 about 0.5 kByte would suffice for the storage of one secret key on a node. This would decrease the size of the RECONETS demonstrator to approximately 10900 LUTs and the 0.5 kByte KEYROM to approximately 331 LUTs (3.0%).

This would reduce the hardware overhead for the security layer to a negligible amount of less than 5 % of the complete system.

The large increase in software cost of 83% is mainly caused by additional system functions that are needed by the ported libgcrypt library (like scanf to read in the keys stored in a text format) and can not be reduced.

A detailed trace including the needed execution time of every operation of the security layer can be found in Appendix A.3. The system bootup process for two nodes including a test of all functions of the crypto library (9s), verification of certificates (2s) and one key-exchange (7s) takes about 18s. The test of the correct functionality of the crypto library could be omitted which would speed up the boot process to 9s.

5 Outlook

The security architecture for a ReCoNets proposed in this thesis can be extended into many directions and further theses in the area of *Security in Reconfigurable, Distributed Embedded Systems* can base on the modules implemented within this thesis:

- *Authenticated Task Resolution Protocol ATRP* as suggested in 4.2.5:
  This protocol could combine the existing rules for task binding of the ReCoNets with the cryptographic aspects of secure task migration presented in this thesis to support secure on- and offline update scenarios of the ReCoNets.

- *Extension of the fast authenticated key-exchange protocol* as proposed in 4.2.3:
  By using two-phases to exchange a session-key and in regular intervals a temporary-key for the symmetric signature of messages, the long-term security of the exchanged session-key is improved as the temporary-key expires and is instantly renewed during runtime.

- *Secure external memory* in FPGAs as assumed in 3.1:
  Today's FPGA have very few internal memory and use cheaper on-board but off-chip external memory instead. The wires between the FPGA and the SRAM chip however are subject to data modification or eavesdropping.
  Techniques like "Efficient Memory Integrity Verification and Encryption for Secure Processors" [SCG+03] could be applied to FPGAs to secure external memory against modification and eavesdropping.

- *Verification of symmetric signatures in hardware* of the software modules on system boot as described in 3.2.2:
  On system bootup, a HMAC-SHA-256, implemented completely in hardware, checks the symmetric signature of the confidential software modules (crypto core, root certificate, OS) and the CPU is only allowed to start, if this signature is correct. Afterwards the hardware HMAC-SHA-256 module is no longer needed, as messages can efficiently signed symmetrically in software. The hardware HMAC-SHA-256 module can now be displaced by a hardware Montgomery multiplication module by dynamic reconfiguration of the FPGA. The Montgomery multiplier would speed up the very computation intensive asymmetric key operations that are currently implemented in software only and are needed to verify the certificates

and task signatures during the initialization of the crypto core and the following authenticated key exchanges of the nodes. The functionality of dynamic reconfiguration is supported by the XILINX FPGAs that are used in the ESM version of the ReCoNets demonstrator that has been developed during the last months by Thomas Walther and Dirk Koch [Wal07]. This would be a useful and practical example for the application of dynamic reconfiguration to speed up system bootup while keeping the needed FPGA area of the ReCoNet constant.

6 Conclusion

This thesis investigates how cryptography can be used to secure a reconfigurable, distributed, embedded system. In particular a security architecture that allows secure task migration and secure interprocess communication in a ReCoNet is designed.

The possible intentional attacks against a ReCoNet that have to be prevented are explained in chapter 1. The intended security objectives of a ReCoNet concern the whole distributed system of connected reconfigurable, embedded systems. The section on attacks and countermeasures on FPGAs shows what methods already exist to protect a single FPGA-based reconfigurable, embedded system against known attacks.

These reconfigurable, embedded security measurements are extended to achieve the security objectives of the distributed, reconfigurable, embedded system. The used cryptographic primitives are described in chapter 2: Many encryption algorithms rely on cryptographic secure random numbers as a unpredictable source of entropy. For secure interprocess communication, symmetric ciphers and signatures are needed. Asymmetric ciphers and signatures are the fundamental concepts for certificates and authenticated key-exchange protocols.

The security architecture for the RECONETS project proposed in chapter 3 partitions the security layer into hard- and software parts and describes how the security objectives can be achieved. A general scheme for digital rights is proposed and implemented which is powerful enough to describe almost any relations between tasks and nodes for secure task migration in a ReCoNet. Two practical examples for the usage of this scheme are explained. A manufacturer hierarchy allows each manufacturer to give parts of his permissions to sub manufacturers. He issues a sub manufacturer certificate to each which allows them to produce in his name.

The existing RECONETS demonstrator was extended by implementing most of the modules of the security architecture as explained in chapter 4. The needed cryptographic algorithms were taken from a standard cryptographic software library (libgcrypt) which was ported to the demonstrator system. The implemented true random number generator is written in generic VHDL to be manufacturer independent and to allow an easy port to other architectures. Currently, the RECONETS demonstrator supports secure interprocess communication and has all cryptographic prerequisites for secure task migration implemented. Finally the costs for the security layer of the RECONETS demonstrator were shown to be negligible in hardware- but huge in relative software costs (space and runtime) as the complete functionality is currently implemented in software.

A proposal for further research in the area of ”Security in Reconfigurable, Distributed Embedded Systems” is described in chapter 5.

In the appendix, a complete demonstration of the new features of the extended ReCoNets demonstrator is provided. The host tools to generate, check, and show certificates and signatures are documented in A.1. An example for the setup of the demonstrator in a real scenario is provided in A.2. Finally, the trace of a simple ReCoNet consisting of two nodes in A.3 demonstrates the implemented, fast authenticated key exchange, certificate and signature verification, secure interprocess communication, a check of all ported functions of the cryptographic software library and the needed execution time for each cryptographic operation.

Within the next years, security aspects in reconfigurable, distributed embedded systems like those that are already embedded or currently developed for use in automobiles or aircrafts will play an increasing role. Attacks on these systems - either for personal advantage like chip-tuning or intentional attacks to bully a rival manufacturer or even with terroristic background can already be prevented in today’s systems as shown in this thesis.

Bibliography

[Alt] Altera Corporation: *Design Security in Stratix II and Stratix II GX Devices*. http://www.altera.com/products/devices/stratix2/features/security/st2-security.html

[Alt06] *Altera Corporation*. http://www.altera.com. Version: 2006

[AT] ANIL TELIKEPALLI, Xilinx I.: *Is Your FPGA Design Secure?* http://www.xilinx.com/publications/xcellonline/xcell_47/xc_pdf/xc_secure47.pdf

[Bar] BARRETO, Paulo: *The Hash Function Lounge*. http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html

[BMA+05] BOBDA, Christophe ; MAJER, Mateusz ; AHMADINIA, Ali ; HALLER, Thomas ; LINARTH, André ; TEICH, Jürgen: Increasing the Flexibility in FPGA-Based Reconfigurable Platforms: The Erlangen Slot Machine. In: *IEEE 2005 Conference on Field-Programmable Technology (FPT)*. Singapore, Dec 2005, 37-42

[Coc73] COCKS, C C.: *A Note On 'Non-Secret Encryption'*. Government Communications Headquarters (GCHQ), 1973 http://www.cesg.gov.uk/site/publications/media/notense.pdf

[Den04] DENT, Alexander: *Hybrid Cryptography*. http://eprint.iacr.org/2004/210.pdf. Version: Aug 30 2004

[DH03] DORASWAMY, Naganand ; HARKINS, Dan: *IPSec, The New Security Standard for the Internet, Intranets, and Virtual Private Networks*. Second Edition. Prentice Hall, 2003

[Dit05] DITTRICH, Steffen: *Konzeption und Implementierung einer Infrastruktur für Betriebssystemdienste wie auch deren Analyse und Umsetzung*, Lehrstuhl Hardware-Software-Co-Design, Universität Erlangen-Nürnberg, Diplomarbeit, Sep 2005

[Ell70] ELLIS, J. H.: *The Possibility of Secure Non-Secret Digital Encryption*. Government Communications Headquarters (GCHQ), 1970 http://www.cesg.gov.uk/site/publications/media/possnse.pdf

[Fut00] FUTURE SYSTEMS INC.: *Symmetric Ciphers*. http://cnscenter.future.co.kr/crypto/algorithm/block.html. Version: 2000

[Gnu] *GnuPG - cryptographic library libgcrypt*. http://gnupg.org

[HC98] HARKINS, D. ; CARREL, D.: *The Internet Key Exchange (IKE)*. IETF - Network Working Group, 1998 http://tools.ietf.org/html/rfc2409

[HKT04] HAUBELT, Ch. ; KOCH, D. ; TEICH, J.: Basic OS Support for Distributed Reconfigurable Hardware. In: PIMENTEL, A. (ed.) ; VASSILIADIS, S. (ed.): *Computer Systems: Architectures, Modeling, and Simulation* Vol. 3133. Springer, Berlin, July 2004 (Lecture Notes in Computer Science (LNCS)), p. 30–38

[IMGa] *Advanced Encryption Standard*. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[IMGb] *Data Confidentiality*. http://msdn2.microsoft.com/en-us/library/aa480570.aspx

[IMGc] *Data Origin Authentication*. http://msdn2.microsoft.com/en-us/library/aa480571.aspx

[IMGd] *Digital Signatures*. http://developer.apple.com/documentation/Security/Conceptual/Security_Overview/Concepts/chapter_3_section_6.html

[ITU05] ITU-T: *The Directory: Public-key and attribute certificate frameworks - X.509*. http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-200508-I!!PDF-E. Version: Aug 2005

[KBC97] KRAWCZYK, H. ; BELLARE, M. ; CANETTI, R.: *HMAC: Keyed-Hashing for Message Authentication*. IETF - Network Working Group, 1997 http://tools.ietf.org/html/rfc2104

[Ker83] KERCKHOFF, Auguste: La Cryptographie Militaire. In: *Journal des sciences militaires* (1883). http://www.petitcolas.net/fabien/kerckhoffs/crypto_militaire_1.pdf

[KG04] KOHLBRENNER, Paul ; GAJ, Kris: An Embedded True Random Number Generator for FPGAs. In: *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–829–6, p. 71–78

[KS05] KOCH, Werner ; SCHULTE, Moritz: *The Libgcrypt Reference Manual*. Version 1.2.2, July 2005. http://www.cse.psu.edu/~cg497c/gcrypt.pdf

[KSD+06] KOCH, Dirk ; STREICHERT, Thilo ; DITTRICH, Steffen ; STRENGERT, Christian ; HAUBELT, Christian ; TEICH, Jürgen: An Operating System Infrastructure for Fault-Tolerant Reconfigurable Networks. In: *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS 2006), Frankfurt / Main, Germany*. Frankfurt, Germany : Springer, März 2006, p. 202–216

[MVO96] MENEZES, Alfred J. ; VANSTONE, Scott A. ; OORSCHOT, Paul C. V.: *Handbook of Applied Cryptography*. Boca Raton, FL, USA : CRC Press, Inc., 1996 http://www.cacr.math.uwaterloo.ca/hac/. – ISBN 0849385237

[Nat99] NATIONAL INSTITUTE OF STANDARDS & TECHNOLOGY: *FIPS 46-3 - Data Encryption Standard (DES)*. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf. Version: Oct 1999

[Nat01] NATIONAL INSTITUTE OF STANDARDS & TECHNOLOGY: *FIPS 197 - Announcing the Advanced Encryption Standard (AES)*. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Version: Nov 2001

[Nat02] NATIONAL INSTITUTE OF STANDARDS & TECHNOLOGY: *FIPS 180-2 - Announcing the Secure Hash Standard*. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf. Version: Aug 2002

[Nat07] NATIONAL INSTITUTE OF STANDARDS & TECHNOLOGY: *Tentative Timeline of the Development of New Hash Functions*. http://www.csrc.nist.gov/pki/HashWorkshop/AHS_Timeline_022307.pdf. Version: Feb 23 2007

[RDS02] ROBERT D. SILVERMAN, RSA L.: *Has the RSA algorithm been compromised as a result of Bernstein's Paper?* http://www.rsasecurity.com/rsalabs/node.asp?id=2007. Version: Apr 2002

[ReC] *ReCoNets*. http://www12.informatik.uni-erlangen.de/research/reconets/

[Ros98] ROSING, Michael: *Implementing Elliptic Curve Cryptography*. Manning Publications, 1998

[RRS06] RECHBERGER, C. ; RIJMEN, V. ; SKLAVOS, N.: The NIST Cryptographic Workshop on Hash Functions. In: *Security and Privacy Magazine, IEEE* 4 (2006), Jan-Feb, Nr. 1, 54-56. http://ieeexplore.ieee.org/iel5/8013/33481/01588827.pdf

[RSA] *The RSA Challenge Numbers*. http://www.rsasecurity.com/rsalabs/node.asp?id=2093

[RSA78] RIVEST, R.L. ; SHAMIR, A. ; ADLEMAN, L.: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, Vol. 21 (2), pp-120-126. http://theory.lcs.mit.edu/~rivest/rsapaper.pdf. Version: 1978

[RSA02] RSA LABORATORIES: *PKCS #1 v2.1: RSA Cryptography Standard*. RSA Laboratories, 2002 ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf

[SCG+03] SUH, G. E. ; CLARKE, Dwaine ; GASSEND, Blaise ; DIJK, Marten van ; DEVADAS, Srinivas: Efficient Memory Integrity Verification and Encryption for Secure Processors. In: *Proceedings of the 36th International Symposium on Microarchitecture* (2003). http://www.microarch.org/micro36/html/pdf/suh-EfficMemory.pdf

[Sch96] SCHNEIER, Bruce: *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York, NY, USA : John Wiley & Sons, Inc., 1996. – ISBN 0471117099

[Sch02] SCHNEIER, Bruce: *Crypto-Gram Newsletter - AES News*. http://www.schneier.com/crypto-gram-0209.html#1. Version: Sep 15, 2002

[Sch07] SCHNEIER, Bruce: *Crypto-Gram Newsletter - A New Secure Hash Standard*. http://www.schneier.com/crypto-gram-0702.html#11. Version: Feb 15, 2007

[SKHT06] STREICHERT, Thilo ; KOCH, Dirk ; HAUBELT, Christian ; TEICH, Jürgen: Modeling and Design of Fault-Tolerant and Self-Adaptive Reconfigurable Networked Embedded Systems. In: *EURASIP Journal on Embedded Systems* (2006)

[Wal07] WALTHER, Thomas: *Concept and Implementation of an FPGA-Platform Independent Softcore RISC-CPU*. Lehrstuhl Hardware-Software-Co-Design, Universität Erlangen-Nürnberg, Master's Thesis, 2007

[WGP03] WOLLINGER, Thomas ; GUAJARDO, Jorge ; PAAR, Christof: Cryptography on FPGAs: State of the Art Implementations and Attacks. In: *ACM Special Issue Security and Embedded Systems* (2003), March. http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/wollingeretal_acmtransembeddedsysfpgacryptooverview_final.pdf

[Wil74] WILLIAMSON, M H.: *Non-secret encryption using a finite field*. Government Communications Headquarters (GCHQ), 1974 http://www.cesg.gov.uk/site/publications/media/secenc.pdf

[Zim80] ZIMMERMANN, Hubert: OSI Reference Model - The ISO Model of Architecture of Open Systems Interconnection. In: *IEEE Transactions on Communications* Vol. 28, 1980, 425 - 432

A Documentation and Demonstration

A.1 Host Tools

The following tools can be used to *create*, *check* or show (*sexp*) certificates and signatures on a host computer running Linux, UNIX, OS X or an equivalent OS. This data can be *extract*-ed and *install*-ed into the extended RECONETS demonstrator's hard- and software.

A.1.1 create_manufacturer

```
SYNTAX: create_manufacturer {-m MANID} {-n NAME} {-d RIGHTFILE}
                            {-M MANCERTDIR} {-K MANKEYDIR}
                            {-q} {-h | -help}

Issues certificate for a manufacturer.

-m MANID: Id of the new manufacturer. (ask otherwise) -
          e.g. 0 or 0.1
-n NAME: Name of the new manufacturer. (ask otherwise) -
         e.g. "Manufacturer 12"
-d RIGHTFILE: File containing digital rights for the certificate to issue.
              (ask otherwise)
-M MANCERTDIR: Directory containing manufacturers' certificates.
               (Default: "data/MCERTS")
-K MANKEYDIR: Directory containing manufacturers' private-keys.
              (Default: "data/MKEYS")
-q: Be quiet.

-h | -help: show this help
```

A.1.2 create_node

```
SYNTAX: create_node {-i NODEID} {-i NAME} {-d RIGHTFILE} {-c MANCERTDIR}
                    {-k MANKEYDIR} {-C NODECERTDIR} {-K NODEKEYDIR}
                    {-q} {-h | -help}
```

```
Issues certificate for a node.

-i NODEID: Id of the node. (ask otherwise) - e.g. 0 or 0.1
-n NAME: Name of the node. (ask otherwise) - e.g. "Root Manufacturer"
-d RIGHTFILE: File containing digital rights for the node certificate
              to issue. (ask otherwise)
-c MANCERTDIR: Directory containing manufacturers' certificates.
               (Default: "data/MCERTS")
-k MANKEYDIR: Directory containing manufacturers' private-keys.
              (Default: "data/MKEYS")
-C NODECERTDIR: Directory containing nodes' certificates.
                (Default: "data/NCERTS")
-K NODEKEYDIR: Directory containing nodes' private-keys.
               (Default: "data/NKEYS")
-q: Be quiet.

-h | -help: show this help
```

A.1.3 create_task

```
SYNTAX: create_task -b BINARYFILE {-t TASKID} {-n NAME} {-d RIGHTFILE}
                    {-M MANCERTDIR} {-K MANKEYDIR} {-T TASKSIGDIR}
                    {-q} {-h | -help}

Signs a task.

-b BINARYFILE: Binary of task to sign. (must be given)
-t TASKID: Id of the task. (ask otherwise) - e.g. 0.1 or 0.1.3
-n NAME: Name of the task. (ask otherwise) -
         e.g. "Drive-by-wire controller"
-d RIGHTFILE: File containing digital rights for the task signature
              to issue. (ask otherwise)
-M MANCERTDIR: Directory containing manufacturers' certificates.
               (Default: "data/MCERTS")
-K MANKEYDIR: Directory containing manufacturers' private-keys.
              (Default: "data/MKEYS")
-T TASKSIGDIR: Directory containing tasks' signatures.
               (Default: "data/TSIGS")
-q: Be quiet.

-h | -help: show this help
```

A.1.4 sexp

sexp can be used to show certificates or keys in *CRYPT/host/data/* in a human readable format (-a).

```
The program ‘sexp’ reads, parses, and prints out S-expressions.
 INPUT:
   Input is normally taken from stdin, but this can be changed:
      -i filename      -- takes input from file instead.
      -p               -- prompts user for console input
   Input is normally parsed, but this can be changed:
      -s               -- treat input up to EOF as a single string
 CONTROL LOOP:
   The main routine typically reads one S-expression, prints it out
   again, and stops.  This may be modified:
      -x               -- execute main loop repeatedly until EOF
 OUTPUT:
   Output is normally written to stdout, but this can be changed:
      -o filename      -- write output to file instead
   The output format is normally canonical, but this can be changed:
      -a               -- write output in advanced transport format
      -b               -- write output in base-64 output format
      -c               -- write output in canonical format
      -l               -- suppress linefeeds after output
   More than one output format can be requested at once.
 There is normally a line-width of 75 on output, but:
      -w width         -- changes line width to specified width.
                          (0 implies no line-width constraint)
 The default switches are: -p -a -b -c -x
 Typical usage: cat certificate-file | sexp -a -x
```

A.1.5 check_run

```
SYNTAX: check_run {-t TASKID} {-n NODEID} {-T TASKSIGDIR}
                  {-N NODECERTDIR} {-q} {-h | -help}

Check whether task TASKID can run on node NODEID.

-t TASKID: Id of the task to check - e.g. 0.1 or 0.2.1
           (if omitted: all tasks in TASKSIGDIR)
-n NODEID: Id of the node to check - e.g. 0.2 or 0.1.3
           (if omitted: all nodes in NODECERTDIR)
-T TASKSIGDIR: Directory containing tasks’ signatures.
               (Default: "data/TSIGS")
-N NODE: Directory containing nodes’ certificates.
```

```
         (Default: "data/NCERTS")
-q: Be quiet.

-h | -help: show this help
```

A.1.6 check_manufacturer

```
SYNTAX: check_manufacturer (-m MANID | -a) {-M MANCERTDIR}
                           {-q} {-h | -help}

Checks certificate of a manufacturer.

-m MANID: Id of the manufacturer to check - e.g. 0 or 0.1
-a: check all manufacturers' certificates in MANCERTDIR
-M MANCERTDIR: Directory containing manufacturers' certificates.
               (Default: "data/MCERTS")
-q: Be quiet.

-h | -help: show this help
```

A.1.7 check_node

```
SYNTAX: check_node (-n NODEID | -a) {-M MANCERTDIR} {-N NODECERTDIR}
                   {-q} {-h | -help}

Check certificate of a node.

-n NODEID: Id of the node to check - e.g. 0.1 or 0.2.1
-a: check all nodes' certificates in NODECERTDIR
-M MANCERTDIR: Directory containing manufacturers' certificates.
               (Default: "data/MCERTS")
-N NODECERTDIR: Directory containing nodes' certificates.
                (Default: "data/NCERTS")
-q: Be quiet.

-h | -help: show this help
```

A.1.8 check_task

```
SYNTAX: check_task (-t TASKID | -a) {-M MANCERTDIR} {-T TASKSIGDIR}
                   {-q} {-h | -help}

Check signature of a task.
```

```
-t TASKID: Id of the task to check - e.g. 0.1 or 0.2.1
-a: check all tasks' certificates in TASKSIGDIR
-M MANCERTDIR: Directory containing manufacturers' certificates.
               (Default: "data/MCERTS")
-T TASKSIGDIR: Directory containing tasks' signatures.
               (Default: "data/TSIGS")
-q: Be quiet.

-h | -help: show this help
```

A.1.9 extract_certs.sh

Export manufacturer certificates, node certificates and task signatures from *data/* into *include/certs.h*.

A.1.10 install_certs.sh

Install *include/certs.h* in U300. Recompile U300 afterwards.

A.1.11 extract_keys.sh

Extract secret keys from *data/NKEYS* to *allkeys* and generate SECROM in *crypt_secrom_rom.vhd*.

A.1.12 install_keys.sh

Install *crypt_secrom_rom.vhd* in *../../EPLD_audio/v0.0/*. Re-synthesize hardware afterwards.

A.2 Example

This is an example for a complete setup of the extended ReCoNets demonstrator describing how to use the previously described host tools, the digital right examples and all used data formats in a real context of a ReCoNet.

The following commands must be entered in a Linux-, UNIX-, Solaris-, MAC OS- or equivalent shell.

A.2.1 Manufacturer Certificates

In our example we have six manufacturers with different permissions in the following hierarchy:

- Manufacturer 0 is the root manufacturer.
  - Manufacturer 0.1 is allowed to build high reliable nodes with no camera connected.
  - Manufacturer 0.2 is allowed to build low reliable nodes with any camera connected.
  - Manufacturer 0.3 is allowed to create software of any reliability level with any camera support.
    - Manufacturer 0.3.1 is allowed to build low reliable software with needed color camera support.
    - Manufacturer 0.3.2 is allowed to build high reliable software with any camera support.

We design the DRs as described in Section 3.5:

- The first DRV encodes the reliability (2 levels).
- The second DRV encodes the connected camera type:
  t_1=b/w camera, t_2=color camera

We first clean all data (certificates and signatures):

```
> cd CRYPT/host/data; make clean; cd ../../../
```

Then we create the corresponding manufacturer certificates:

```
> cd CRYPT/host
> echo "(digitalrights #11# #0111#)" > R;
> ./bin/create_manufacturer -m 0 -n "Root Manufacturer" -d R -q;
> echo "(digitalrights #11# #0001#)" > R;
> ./bin/create_manufacturer -m 0.1 -n "HW Manufacturer 0.1" -d R -q;
> echo "(digitalrights #10# #0111#)" > R;
```

```
> ./bin/create_manufacturer -m 0.2 -n "HW Manufacturer 0.2" -d R -q;
> echo "(digitalrights #11# #0111#)" > R;
> ./bin/create_manufacturer -m 0.3 -n "SW Manufacturer 0.3" -d R -q;
> echo "(digitalrights #10# #0010#)" > R;
> ./bin/create_manufacturer -m 0.3.1 -n "SW Manufacturer 0.3.1" -d R -q;
> echo "(digitalrights #11# #0111#)" > R;
> ./bin/create_manufacturer -m 0.3.2 -n "SW Manufacturer 0.3.2" -d R -q;
> rm R;
```

Have a look at the certificate of Manufacturer 0.3.1 (rsa public-key and sig-val values will differ!):

```
> ./bin/sexp -i data/MCERTS/0.3.1 -a
(manufacturer
 (signature
  (signed
   (id ID_0.3.1)
   (name "SW Manufacturer 0.3.1")
   (digitalrights #10# #0010#)
   (public-key
    (rsa
     (n
      #00D0ADB2558F5E5A6197ABEDBDE83FC6A3ADA411E020BA6F664BD146707BC46
      F24398EB63E049CAED99E3CE6B6BC0D7B50F2A59CA8CFD4B42277435346086DA
      A9A3B74568FA35532A9472BC1A5F6D86565EE2AC6EF6DE5372A53A14B6CE999B
      501189A1D3A9579F8AA1F5B053E0E55F0872AB4B6A76584E75E229CAACD89B86
      26F#)
     (e #010001#))))
  (sig-val
   (rsa
    (s
     #74F0C185CFB9DDA20F556D1DD6623BA5A2912E8A75C50CD1A688FD5CADE6CA6E
     FA3ECE7323286986B10E64730D93CBFE35EACBFFAA0CF54D6B6A21F4D5E23519F
     1CA64E16CA95ACF03F1F7AC17A7DE24E6112EC565AE0BA3AE311FF07DCACFBB95
     09B9112ADAAA6DBAB076E829DB30508F44326F8CBEA7ECD52BCC2F72896E72#)
    ))))
```

The corresponding secret key is (rsa private-key values will differ!):

```
> ./bin/sexp -i data/MKEYS/0.3.1 -a
(private-key
 (rsa
  (n
   #00D0ADB2558F5E5A6197ABEDBDE83FC6A3ADA411E020BA6F664BD146707BC46F24
   398EB63E049CAED99E3CE6B6BC0D7B50F2A59CA8CFD4B42277435346086DAA9A3B7
```

```
  4568FA35532A9472BC1A5F6D86565EE2AC6EF6DE5372A53A14B6CE999B501189A1D
  3A9579F8AA1F5B053E0E55F0872AB4B6A76584E75E229CAACD89B8626F#)
 (e #010001#)
 (d
  #60D0AD63D7B3C13FF4F3CDC5A54A6D78C3D75279C7056828B03544366C9D9AA8D6
  515948D5AFF1C9420A6449D45E76DF7BEC0D0E1EFA42A698E971E9948078BCE5B68
  79E8B0D4E242D3CAFC6C690768DD576CAAF394758543754A0298C45EB71E5ABCBB2
  7122FC63958EAB281D1E1966B3C3C578AB6DA572ED727A596609789D#)
 (p
  #00D842DCB91D520A2E2DD838EE3D157A49665778683D3D391A693C30F04E5ADFC7
  489A1C81EFEBB180A35A6CAE4ED32F147899F06C5721F7BA71DEF8E545514013#)
 (q
  #00F7062347C813D3C15A3DF57A2E2255FFC3DD30A5A4D0D048E06810200360E244
  EBCCB763AEB332889ABE7E0F050432850FC0432D4FE9BAC5C7198FE907B637B5#)
 (u
  #70FEC442BCFFF1DC8A5665B5E2C7AD0CBB1353ED8759914EB05789E6F9FF5F6246
  4CB1AC0727610F37FD93E95E92ECE3EF0EF126AE69F658A684F44AA27D4EDA#)))
```

A.2.2 Node Certificates

We take two nodes with different hardware support:

- Node 0.1.1 (Alice) - a high reliable node from Manufacturer 0.1 with no camera
- Node 0.2.1 (Bob) - a low reliable node from Manufacturer 0.2 with a b/w camera connected

Let's create the corresponding node certificates:

```
> echo "(digitalrights #11# #0001#)" > R;
> ./bin/create_node -i 0.1.1 -n "Alice" -d R -q;
> echo "(digitalrights #10# #0101#)" > R;
> ./bin/create_node -i 0.2.1 -n "Bob" -d R -q;
> rm R;
```

The node certificate of Node 0.2.1 can be shown by (rsa public-key and sig-val values will differ!):

```
> ./bin/sexp -i data/NCERTS/0.2.1 -a
(node
 (signature
  (signed
   (id ID_0.2.1)
   (name Bob)
   (digitalrights #10# #0101#)
   (public-key
```

```
    (rsa
     (n
      #00E97135BE65D4669A4F9EE8A535C9EA2D8E286A57B3819A5EF9159B041AEF6
      3F37E8B58CCBBF42FC730A647EEBC55AEE1069F85C3722900EEF35E74F7CF74F
      FB14647864A6271BFE194539498D996DFA765FFEC34B4B95E072281C809EB20A
      E20691B20E025ECD2928A203904635116FBADE31E8F1311362B2929E056172D7
      07F#)
     (e #010001#))))
  (sig-val
   (rsa
    (s
     #0083C1B50BB647919CD025F268F7E9BA4329A9941197D6BBA77400BBC5490773
     08F38BD563B863D0B6971FCF3CA830D03B343FF97761E1E0F4ED19AEE870D44FA
     89DC4F9842D3C7028E0D0D19D68D7B94FC2810E439E87A2AEBE87D2B1127F6D2E
     F55B77D803148DCF22633FECEB4B7B2DB4C729F6B9CEA43979EF6ABD101F40C9#
     )))))
```

The corresponding key is (rsa private-key values will differ!):

```
> ./bin/sexp -i data/NKEYS/0.2.1 -a
(private-key
 (rsa
  (n
   #00E97135BE65D4669A4F9EE8A535C9EA2D8E286A57B3819A5EF9159B041AEF63F3
   7E8B58CCBBF42FC730A647EEBC55AEE1069F85C3722900EEF35E74F7CF74FFB1464
   7864A6271BFE194539498D996DFA765FFEC34B4B95E072281C809EB20AE20691B20
   E025ECD2928A203904635116FBADE31E8F1311362B2929E056172D707F#)
  (e #010001#)
  (d
   #2B8A3D836B1BC01D54EF6725F54FD93930F411CD94C1FE086BBDDF615722C24A36
   9687F3FBB4723ADD348E63154687ED199EA444CD649F7371F9F2A80BCE1F2856A18
   B23C737B2F3E4BE138BA6B22240F450E1EF00FB645AD614AEA052845DC39FF5664D
   57571AE8874639C7A5B99E924ED67C9EC32FD8C84A5898746C3C3701#)
  (p
   #00EE0CAB40E637699CF1022E18573A5E40FC5150A6558F058327B4B163A9275B29
   5C2B95FED221FA0FC61EFD3EA1D14D3165514C22176BEF473DCBB170A6EFF701#)
  (q
   #00FB0B9A18812BC472DDBAB7D1E6717881FE5F47A0DDCD116080CE64A1B0676003
   4C346E0EA54346584F50688A974DAA3034AD8533E29DBA3E31521FC0FE40E77F#)
  (u
   #380BFBF8C5942EA21ACC845AF78B3D2A2C9D78663211A8F35FA5D8B51070F7D639
   BCE0388028273FA148D26507E5D4DF0DAAAC394B8A1F21BADDB1179BE7BB1A#)))
```

A.2.3 Task Signatures

Suppose we have these tasks:

- Signed by Manufacturer 0.3:
  - Task 0.3.1: high reliable task with no camera needed
  - Signed by Manufacturer 0.3.1:
    - Task 0.3.1.1: low reliable task with color camera needed
  - Signed by Manufacturer 0.3.2:
    - Task 0.3.2.1: low reliable task with b/w camera needed
    - Task 0.3.2.2: low reliable task with any camera needed

We now create the task signatures with dummy binaries in Bi (replace them with real binaries of the corresponding task).

```
> echo "binary1code" > B1; echo "(digitalrights #01# #0111#)" > R;
> ./bin/create_task -b B1 -t 0.3.1 -n "Task 1" -d R -q;
> echo "binary2code" > B2; echo "(digitalrights #10# #0010#)" > R;
> ./bin/create_task -b B2 -t 0.3.1.1 -n "Task 2" -d R -q;
> echo "binary3code" > B3; echo "(digitalrights #10# #0100#)" > R;
> ./bin/create_task -b B3 -t 0.3.2.1 -n "Task 3" -d R -q;
> echo "binary4code" > B4; echo "(digitalrights #10# #0110#)" > R;
> ./bin/create_task -b B4 -t 0.3.2.2 -n "Task 4" -d R -q;
> rm R; rm B1 B2 B3 B4;
```

We look at the signature for Task 0.3.2.2 (rsa sig-val value will differ!):

```
> ./bin/sexp -i data/TSIGS/0.3.2.2 -a
(task
 (signature
  (signed
   (id ID_0.3.2.2)
   (name "Task 3")
   (digitalrights #10# #0110#)
   (len #000000000000000C#)
   (hash #23621EFEF48705C4BEA28D90451A455307A21DB70906E466E413B0C9405D
    3544#))
  (sig-val
   (rsa
    (s
     #0096DEC5048EC63E2590B696A0597B511609B565E897FC5398A25E28B4254D21
     997692099B6596F3891828CD24580BA6E39FFF7049B52129847F67919AE244E72
     E5C6C2A79A0DB1B2673770DC1AFED6B375F7CBDDF873ED0D76CFA232967BB86D5
     A62EAA0E2F698B985FC39C1AABAA8F349A050ED514C49A0246B38CF1CCA60A0F#
     )))))
```

A.2.4 Allowed Binding between Nodes and Tasks

We can examine which task is allowed to run on which node:

```
> ./bin/check_run -q
Task 0.3.1 is allowed to run on nodes:
0.1.1: YES
0.2.1: NO

Task 0.3.1.1 is allowed to run on nodes:
0.1.1: NO
0.2.1: NO

Task 0.3.2.1 is allowed to run on nodes:
0.1.1: NO
0.2.1: YES

Task 0.3.2.2 is allowed to run on nodes:
0.1.1: NO
0.2.1: YES
```

A.2.5 Prepare and run Demonstrator

Next we extract and install the secret node keys into the hardware:

```
> ./bin/extract_keys.sh
Hex to VHDL ROM converter by Daniel Wallner. Version 0244
Reading binary file
Keys written to crypt_secrom_rom.vhd
> ./bin/install_keys.sh
Installing secret keys
Please synthesize hardware now.
```

Afterwards we synthesize the hardware in Quartus and load it on the two connected FPGA nodes.

We extract the certificates and task signatures into the software branch U300 of the ReCoNets demonstrator:

```
> ./bin/extract_certs.sh
> ./bin/install_certs.sh
Installing certs
Please make U300 now and U400 now.
```

Afterwards we make U300 and U400 in the Altera Nios2-Command-Shell and download the software on both nodes:

```
[SOPC Builder] cd PATH_TO_SWPROJECT/U300/V0.0/; make clean; make; make
[SOPC Builder] cd PATH_TO_SWPROJECT/U400/V0.0/; make
[SOPC Builder] nios2-download U400.elf
-- Plug JTAG connector to other Altera Board now --
[SOPC Builder]  nios2-download U400.elf
```

Finally we start our test system and see, that the allowed task assignment is correctly verified on system boot.

- Node 1 (Alice): connected via COM1, nad=0x42
- Node 2 (Bob) : connected via COM2, nad=0x43

Start two nios-run consoles in two Altera Nios2-Command-Shells to view the nodes' debugging outputs:

Alice:

```
[SOPC Builder] nr -t -p COM1
```

Bob:

```
[SOPC Builder] nr -t -p COM2
```

After the two start buttons on both nodes are pressed a runtime trace like the one in the following section will appear in the two consoles.

A.3 Demonstrator Traces

The following traces show the initialization of the crypto layer on system startup of two connected RECONETS nodes, the fast authenticated key-exchange and signed messages exchanged between them.

- Alice: nad 42 (0x29) with node certificate 0.1.1
- Bob: nad 41 (0x2A) with node certificate 0.2.1

The following traces of Alice respectively Bob consist of these phases whose beginnings are marked in the traces:

- - 1 - - Every node checks whether all ported functions of libgcrypt work correctly (runtime $\sim 9s$).

- - 2 - - All manufacturer and node certificates and task signatures are verified and the secret key of the node is loaded. The possible bindings between tasks and nodes are determined. ($\sim 2s$)

- - 3 - - The Task Resolution Protocol is initialized and the ROUTE protocol starts to establish routes.

- - 4 - - When the route between the two connected nodes is set up, the CRYPT protocol authenticates the nodes to each other and exchanges a secret key ($\sim 7s$). All messages sent before the completion of the key exchange are not signed symmetrically (”Unsigned”). The signature of incoming packets is not verified (”Unverified.”)

- - 5 - - After the key exchange, Alice ”pings” Bob. The every outgoing message (the ping message as well as its reply) is ”Signed” and the signature of all incoming messages is verified (”Signature verification: OK.”). Signing and verifying a ping message needs $< 4ms$ each.

- - 6 - - Finally Bob ”pings” Alice. The messages are also signed and verified correctly with the previously exchanged symmetric key.

The time measurements have a resolution of the system clock’s frequency of $\frac{1}{50MHz} = 20ns$.

A.3.1 Trace of Node "Alice"

```
nios-run: Entering terminal mode over COM1 at 115200 bps

nios-run: Terminal mode (Control-C exits)
-----------------------------------------
type 'help' for help
7Segment opened correctly
Inf12_NET opened correctly
Initializing CRYPT layer:
-- 1 --
Testing libgcrypt...
  Checking SECMEM
  Checking TRNG
  Checking AES
AES-128:
Time for encrypting one AES Block: 2770 us
Time for decrypting one AES Block: 893 us
AES-192:
Time for encrypting one AES Block: 768 us
Time for decrypting one AES Block: 974 us
AES-256:
Time for encrypting one AES Block: 592 us
Time for decrypting one AES Block: 1033 us
  Checking SHA
Time for hashing 3 bytes: 790 us
Time for hashing 56 bytes: 1119 us
Time for hashing 1000000 bytes: 1608077 us
  Checking HMAC
Time for HMAC of 8 bytes: 2436 us
Time for HMAC of 28 bytes: 2472 us
Time for HMAC of 50 bytes: 2355 us
Time for HMAC of 50 bytes: 2524 us
Time for HMAC of 20 bytes: 2527 us
Time for HMAC of 54 bytes: 3375 us
Time for HMAC of 152 bytes: 3628 us
  Checking Random
  Checking Pubkey
  Checking RSA sign/verify
Time to create one signature: 1996889 us
Time to verify one signature: 66848 us
Time to create one signature: 2771 us
Time to create one signature: 3925 us
Time to create one signature: 1404446 us
```

```
Time to verify one signature: 66706 us
Time to create one signature: 1403462 us
Time to verify one signature: 66771 us
Time to create one signature: 2798 us
Time to create one signature: 3817 us
  Checking RSA enc/dec
Time for asymetric encryption: 72412 us
Time for asymetric decryption: 2201369 us
Test libgcrypt successful.
Time for test_libgcrypt(): 9203565 us
-- 2 --
Loading manufacturer cert 0: OK.
Time to check manufacturer certificate: 73875 us
Loading manufacturer cert 0.1: OK.
Time to check manufacturer certificate: 72447 us
Loading manufacturer cert 0.2: OK.
Time to check manufacturer certificate: 72304 us
Loading manufacturer cert 0.3: OK.
Time to check manufacturer certificate: 72505 us
Loading manufacturer cert 0.3.1: OK.
Time to check manufacturer certificate: 72485 us
Loading manufacturer cert 0.3.2: OK.
Time to check manufacturer certificate: 72605 us
Loading node cert 0.1.1: OK.
Time to check node certificate: 216039 us
Loading node cert 0.2.1: OK.
Time to check node certificate: 216840 us
Loading task signature 0.3.1: OK.
Time to check task signature: 142149 us
Loading task signature 0.3.1.1: OK.
Time to check task signature: 213979 us
Loading task signature 0.3.2.1: OK.
Time to check task signature: 218423 us
Loading task signature 0.3.2.2: OK.
Time to check task signature: 215165 us
Assigning secret key 1 to nad 42.
corresponding nodecert id is 0.1.1
These tasks are allowed to run on nodes:
  Node 0.1.1: 0.3.1
  Node 0.2.1: 0.3.1.1, 0.3.2.2
These nodes are allowed to execute tasks:
  Task 0.3.1: 0.1.1, 0.2.1
  Task 0.3.1.1:
  Task 0.3.2.1: 0.2.1
```

```
  Task 0.3.2.2: 0.2.1
Time for crypt_init(): 2018133 us
Time for complete crypto initialization: 11223606 us
crypt start
-- 3 --
PortUp(00000004)
send LINK @2
2a -> 2 (1) |82| Unsigned
TRP:Neighbour(2):Write:SET 1@2A
2a -> 2 (2) |50| Unsigned
TRP:Neighbour(2):Write:SET 2@2A
2a -> 2 (2) |50| Unsigned
TRP:Neighbour(2):Write:SET 3@2A
2a -> 2 (2) |50| Unsigned
ack for 003FFE4C/1 to 29/2
2a <- 29 @2 (1) |82| Unverified.
CNode::ROUTE_Read(*, 02, 29) 1
LINK @2 from 29
link up: 2a <-> 29
send MATRIX @2
2a -> 2 (1) |174| Unsigned
-- 4 --
Asking 29 for new key.
A_newrequest
2a->29 MessageNo=0
[open]
  [data="newrequest"]
[close]
2a -> 29 (10) |58| Unsigned
---
ack for 003FFB8C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 0040060C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FF88C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FDF8C from 29
(0) acked
ack for 003FF48C/1 to 29/2
2a <- 29 @2 (1) |174| Unverified.
CNode::ROUTE_Read(*, 02, 29) 2
MATRIX @2 from 29
2a <- 29 @42 (10) |55| Unverified.
---
```

```
A0
B_id: 0.2.1
A_nonce = 9a40a85c346cf346
Key = c6db132be20bfd82d272d3b29538a22ccd915db9066b2dd947edd5a2bae1cea3
B_id: 0.2.1
2a->29 MessageNo=0
[open]
  [data="signature"]
  [open]
    [data="signed"]
    [open]
      [open]
        [data="rA"]
        [data="\x9a@\xa8\4l\xf3F"]
      [close]
      [open]
        [data="A"]
        [data="0.1.1"]
      [close]
      [open]
        [data="B"]
        [data="0.2.1"]
      [close]
      [open]
        [data="key-enc"]
        [data="(7:enc-val(3:rsa(1:a128:r\xb18\xe7jW\x82\x11F8\x94F\xf0
          u\xf8\xda$y\x8f\x94\x09\n\xd3N\xf9\xa4}\x9eh\x06\xf2\xdfY\x9
          4\xd3\xed[u\xcb\x1dN_\x07\xd9\xf0x\x1b\x901\x09\xda\xcf\x93f
          \xbd\xba\xe6\x9b\xcf\xe2\xf7\x8a\xd4\x1e\x09-c\x02\x1f\xc3\x
          ce\x98\xc1\xeeB \xfd4\xde\xcb\xcbt'p7\x19h\xc1\xe2v\x9c \x0f
          eT\xdf\xe6\xf7\xd4\xd70\xab\xc6\x84\x17\xa6\xaa\0ft#*\x9eYh\
          xbf#\xe2<i`\x88\xda:,\x8d=)))"]
      [close]
    [close]
  [close]
  [data="\0"]
  [open]
    [data="sig-val"]
    [open]
      [data="rsa"]
      [open]
        [data="s"]
        [data="C\xcb\xb8\x01rN$\xd8\x16\xc8?\xae\x8b]\xe1\xb7\xb3\x14p
          TLC5\xe6\x88vsO\x061\xee,h\xd0,L\xd0s\xea\xa4\xd11\xecC\x8bF
```

```
          \x03\xbd\xf8J\xec\x0f\x1ee.J\x17\xee\x99t+\xa5\xa0Fj\xda\xce
          \xeb\x9f\xcb\xf5O\xf7T\x1aZ\xefzO)\xe87\x07\xb2\xb2V\xfa\x0f
          \x07\xb4!\x11i\xf8\x88i\x22\xa2\x99v\x981$\x90\v}\xa45\x13@\
          x9d\xf7~\n[f\x9eK\xe5U*\xc4\xb7\xea\xe4\xd3\xfde"]
      [close]
    [close]
  [close]
  [data="\0"]
[close]
2a -> 29 (10) |439| Unsigned
Node processing time: 2228917 us
---
ack for 003FFCCC from 29
(1) acked
ack for 004004CC from 29
(2) acked
ack for 003FFFCC from 29
(3) acked
ack for 0040098C from 29
(4) acked
ack for 003FFC0C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FF7CC/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FD54C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FFC0C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FF7CC/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FD54C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FFC0C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FF7CC/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
ack for 003FD54C/2 to 29/2
2a <- 29 @2 (2) |50| Unverified.
2a <- 29 @42 (10) |96| Unverified.
---
A1
[open]
  [data="signed"]
  [open]
```

```
    [open]
      [data="B"]
      [data="0.2.1"]
    [close]
    [open]
      [data="rB"]
      [data="\xf3\xb7\x12\xb8\x8fky$"]
    [close]
  [close]
[close]
2a->29 MessageNo=1
[open]
  [data="signature"]
  [open]
    [data="signed"]
    [open]
      [open]
        [data="B"]
        [data="0.2.1"]
      [close]
      [open]
        [data="rB"]
        [data="\xf3\xb7\x12\xb8\x8fky$"]
      [close]
    [close]
  [close]
  [data="\0"]
  [open]
    [data="sig-val"]
    [open]
      [data="rsa"]
      [open]
        [data="s"]
        [data="O\xe5\xd4\x8b&0\x04v\xf0)^\xb5\xe2h\x7f\xbc\xc5w\xa7X\x
          18Z$\x84\x1bC4\xb9\xd1\xd0\n\xb1- 5\xa0\xca\x91\x22hWh\x87\x
          c3\x84r^\xebc\xaa$B\xce\xcdspL\x9dB\xfb\x86\xe4\x06p\x17\xfa
          \xf0\x82H\x87\x06\xf9N\xebAY\xda\x86\nm\x84\xb7\xe7\x9f\xa6-
          O\xb1\xc7\x12\x93+\xdd\x88\xa7\x87\x1c\xaf:\xbd\xae3l\xcb\x9
          0k\b~0\xd2Ejb\xb8\x13\xfa\xe7\xeb\x14Q\xe7\xc4\xa1\0O\xdfS\x
          d1"]
      [close]
    [close]
  [close]
  [data="\0"]
```

```
[close]
2a -> 29 (10) |257| Unsigned
Authenticated key exchange with 29 successful.
Total time since CRYPT initialization: 18139359 us
Total time since start of Key initialization: 6896754 us
Node processing time: 2157245 us
-- 5 --
> ping 2A 1

Request Ping to node 2A, 1 times
ping(1/1) 2A...
29 -> 2a (7) |44| Signed
Time to sign a packet of length 44: 2992 us
ack for 004013F0 from 2A
(0) acked
ack for 004037B0/8 to 2A/29
29 <- 2a @41 (8) |44| Signature verification: OK.
Time to verify a packet of length 44: 3676 us
ping(1/1) ok from 2A
ping 2A: 1 of 1 ok
-- 6 --
ack for 00401270/7 to 2A/29
29 <- 2a @41 (7) |44| Signature verification: OK.
Time to verify a packet of length 44: 3409 us
PingReq from 2a. Sending reply.
29 -> 2a (8) |44| Signed
Time to sign a packet of length 44: 3005 us
ack for 004029F0 from 2A
(0) acked
---

nios-run: exiting.
```

A.3.2 Trace of Node "Bob"

```
nios-run: Entering terminal mode over COM2: at 115200 bps

nios-run: Terminal mode (Control-C exits)
-----------------------------------------
type 'help' for help
7Segment opened correctly
Inf12_NET opened correctly
Initializing CRYPT layer:
-- 1 --
Testing libgcrypt...
  Checking SECMEM
  Checking TRNG
  Checking AES
AES-128:
Time for encrypting one AES Block: 2706 us
Time for decrypting one AES Block: 890 us
AES-192:
Time for encrypting one AES Block: 770 us
Time for decrypting one AES Block: 965 us
AES-256:
Time for encrypting one AES Block: 604 us
Time for decrypting one AES Block: 1039 us
  Checking SHA
Time for hashing 3 bytes: 788 us
Time for hashing 56 bytes: 1134 us
Time for hashing 1000000 bytes: 1608203 us
  Checking HMAC
Time for HMAC of 8 bytes: 2442 us
Time for HMAC of 28 bytes: 2477 us
Time for HMAC of 50 bytes: 2435 us
Time for HMAC of 50 bytes: 2523 us
Time for HMAC of 20 bytes: 2522 us
Time for HMAC of 54 bytes: 3381 us
Time for HMAC of 152 bytes: 3611 us
  Checking Random
  Checking Pubkey
  Checking RSA sign/verify
Time to create one signature: 1996898 us
Time to verify one signature: 66840 us
Time to create one signature: 2774 us
Time to create one signature: 3947 us
Time to create one signature: 1404489 us
```

```
Time to verify one signature: 66736 us
Time to create one signature: 1403469 us
Time to verify one signature: 66804 us
Time to create one signature: 2798 us
Time to create one signature: 3812 us
  Checking RSA enc/dec
Time for asymetric encryption: 72409 us
Time for asymetric decryption: 2201602 us
Test libgcrypt successful.
Time for test_libgcrypt(): 9203874 us
-- 2 --
Loading manufacturer cert 0: OK.
Time to check manufacturer certificate: 73869 us
Loading manufacturer cert 0.1: OK.
Time to check manufacturer certificate: 72450 us
Loading manufacturer cert 0.2: OK.
Time to check manufacturer certificate: 72524 us
Loading manufacturer cert 0.3: OK.
Time to check manufacturer certificate: 72483 us
Loading manufacturer cert 0.3.1: OK.
Time to check manufacturer certificate: 73213 us
Loading manufacturer cert 0.3.2: OK.
Time to check manufacturer certificate: 73742 us
Loading node cert 0.1.1: OK.
Time to check node certificate: 216027 us
Loading node cert 0.2.1: OK.
Time to check node certificate: 216881 us
Loading task signature 0.3.1: OK.
Time to check task signature: 142193 us
Loading task signature 0.3.1.1: OK.
Time to check task signature: 213999 us
Loading task signature 0.3.2.1: OK.
Time to check task signature: 218448 us
Loading task signature 0.3.2.2: OK.
Time to check task signature: 215168 us
Assigning secret key 2 to nad 41.
corresponding nodecert id is 0.2.1
These tasks are allowed to run on nodes:
  Node 0.1.1: 0.3.1
  Node 0.2.1: 0.3.2.1, 0.3.2.2
These nodes are allowed to execute tasks:
  Task 0.3.1: 0.1.1
  Task 0.3.1.1:
  Task 0.3.2.1: 0.2.1
```

```
  Task 0.3.2.2: 0.2.1
Time for crypt_init(): 2017768 us
Time for complete crypto initialization: 11223512 us
crypt start
-- 3 --
PortUp(00000000)
PortUp(00000004)
send LINK @2
29 -> 2 (1) |82| Unsigned
TRP:Neighbour(2):Write:SET 1@29
29 -> 2 (2) |50| Unsigned
TRP:Neighbour(2):Write:SET 2@29
29 -> 2 (2) |50| Unsigned
TRP:Neighbour(2):Write:SET 3@29
29 -> 2 (2) |50| Unsigned
ack for 003FDF8C/1 to 2A/2
29 <- 2a @2 (1) |82| Unverified.
CNode::ROUTE_Read(*, 02, 2A) 1
LINK @2 from 2A
link up: 29 <-> 2a
send MATRIX @2
29 -> 2 (1) |174| Unsigned
-- 4 --
Initiating keyexchange with 2a.
B0
29->2a MessageNo=0
[open]
  [data="B"]
  [data="0.2.1"]
[close]
29 -> 2a (10) |55| Unsigned
---
ack for 003FFCCC/2 to 2A/2
29 <- 2a @2 (2) |50| Unverified.
ack for 004004CC/2 to 2A/2
29 <- 2a @2 (2) |50| Unverified.
ack for 003FFFCC/2 to 2A/2
29 <- 2a @2 (2) |50| Unverified.
ack for 003FFE4C from 2A
(0) acked
ack for 0040098C/1 to 2A/2
29 <- 2a @2 (1) |174| Unverified.
CNode::ROUTE_Read(*, 02, 2A) 2
MATRIX @2 from 2A
```

```
29 <- 2a @41 (10) |58| Unverified.
---
B1
Node processing time: 471 us
---
ack for 003FFB8C from 2A
(1) acked
ack for 0040060C from 2A
(2) acked
ack for 003FF88C from 2A
(3) acked
ack for 003FF48C from 2A
(4) acked
29 -> 2 (2) |50| Unsigned
29 -> 2 (2) |50| Unsigned
29 -> 2 (2) |50| Unsigned
29 <- 2a @41 (10) |439| Unverified.
---
B1
A_id: 0.1.1
B_id: 0.2.1
Key = c6db132be20bfd82d272d3b29538a22ccd915db9066b2dd947edd5a2bae1cea3
A_nonce = 9a40a85c346cf346
B_nonce = f3b712b88f6b7924
CRYPT: 29->2a Packet 1
c416ff8d08d691e47cf76b0e5e4568609aec49ad31eebdc4756335ebbf69c8259a40a8
  5c346cf346f3b712b88f6b7924302e312e3300
29 -> 2a (10) |96| Unsigned
Node processing time: 2489863 us
---
ack for 003FFC0C from 2A
ack for 003FF7CC from 2A
ack for 003FD54C from 2A
ack for 003FFC0C from 2A
ack for 003FF7CC from 2A
ack for 003FD54C from 2A
ack for 003FFC0C from 2A
ack for 003FF7CC from 2A
ack for 003FD54C from 2A
29 <- 2a @41 (10) |257| Unverified.
---
B2
[open]
  [data="signature"]
```

```
  [open]
    [data="signed"]
    [open]
      [open]
        [data="B"]
        [data="0.2.1"]
      [close]
      [open]
        [data="rB"]
        [data="\xf3\xb7\x12\xb8\x8fky$"]
      [close]
    [close]
  [close]
  [data="\0"]
  [open]
    [data="sig-val"]
    [open]
      [data="rsa"]
      [open]
        [data="s"]
        [data="O\xe5\xd4\x8b&0\x04v\xf0)^\xe2h\x7f\xc5w\xa7\x18Z$\x84\
          x1bC4\xb9\xd1\xd0\n\xb1- 5\xa0\xca\x91\x22hWh\x87\xc3\x84r^\
          xebc\xaa$B\xce\xcdspL\x9dB\xfb\x86\xe4\x06p\x17\xfa\xf0\x82H
          \x87\x06\xf9N\xebAY\xda\x86\nm\x84\xb7\xe7\x9f\xa6-O\xb1\xc7
          \x12\x93+\xdd\x88\xa7\x87\x1c\xaf:\xbd\xae3l\xcb\x90k\b~0\xd
          2Ejb\xb8\x13\xfa\xe7\xeb\x14Q\xe7\xc4\xa1\0O\xdfS\xd1"]
      [close]
    [close]
  [close]
  [data="\0"]
[close]
B_nonce = f3b712b88f6b7924
Authenticated key exchange with 2a successful.
Total time since CRYPT initialization: 18320639 us
Total time since start of Key initialization: 7058272 us
Node processing time: 159215 us
-- 5 --
ack for 004013F0/7 to 29/2A
2a <- 29 @42 (7) |44| Signature verification: OK.
Time to verify a packet of length 44: 3154 us
PingReq from 29. Sending reply.
2a -> 29 (8) |44| Signed
Time to sign a packet of length 44: 3657 us
ack for 004037B0 from 29
```

```
(0) acked
-- 6 --
> ping 29 1

Request Ping to node 29, 1 times
ping(1/1) 29...
2a -> 29 (7) |44| Signed
Time to sign a packet of length 44: 2836 us
ack for 00401270 from 29
(0) acked
ack for 004029F0/8 to 29/2A
2a <- 29 @42 (8) |44| Signature verification: OK.
Time to verify a packet of length 44: 3745 us
ping(1/1) ok from 29
ping 29: 1 of 1 ok
---

nios-run: exiting.
```

B Lists and Index

List of Tables

List of Figures

List of Abbreviations

3-DES Triple - DES

AES Advanced Encryption Standard

AN2N Authenticated Node To Node Protocol

ANSI American National Standards Institute

API Application Programming Interface

ASIC Application Specific Integrated Circuit

ATRP Authenticated Task Resolution Protocol

CA Certificate Authority

CBC Cipher-Block Chaining mode of symmetric block ciphers

CFB Cipher FeedBack mode of symmetric block ciphers

CLB Configurable Logic Block

CP Cell Protocol

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CRYPT Crypto Protocol

CSPRBG cryptographically secure pseudo-random bit generator

CTR CounTeR mode of symmetric block ciphers

DES Data Encryption Standard

DPA Differential Power Analysis

DR Digital Right - consists of several Digital Right Vectors (DRV) that all have to be fulfilled

DRV Digital Right Vector

DSA Digital Signature Algorithm

ECB Electronic Code Book mode of symmetric block ciphers

ECC Elliptic curve cryptography

ECDSA Elliptic Curve Digital Signature Algorithm

EDC Error Detection Codes

ElGamal public-key system based on Diffie-Hellman key agreement and described by Taher Elgamal in 1984

ESM Erlangen Slot Machine

FIB Focused Ion Beam

FIPS Federal Information Processing Standard of the US Institute of Computer Sciences and Technology (ICST)

FLASH non-volatile computer memory that can be electrically erased and reprogrammed

FPGA Field Programmable Gate Array

GCHQ Government Communications Headquarters - British intelligence agency

HW Hardware

IDEA International Data Encryption Algorithm

IKE Internet Key Exchange protocol

IPsec Internet Protocol security

IV Initialization Vector

MAC Message Authentication Code

MCP Multi Cell Protocol

MD Message Digest

MD5 Message-Digest algorithm 5 - cryptographic hash function

MDC Modification Detection Code

MMU Memory Management Unit

N2N Node To Node Protocol

NIST National Institute of Standards and Technology

OFB Output FeedBack mode of symmetric block ciphers

PKCS Public Key Cryptography Standard

PRBG Pseudo-random bit generator

PROM Programmable read-only memory

RBG Random Bit Generator

ReCoNet Reconfigurable Network consisting of Reconfigurable, Distributed, Embedded Systems

ReCoNets Research project run by the University of Erlangen-Nuremberg - Department of Computer Science - Hardware-Software-Co-Design [ReC]

RNG Random Number Generator

ROM Read-Only Memory

ROUTE Route Protocol

RSA public-key system described by Ronald L. Rivest, Adi Shamir and Leonard Adleman in 1977

SHA Secure Hash Algorithm - cryptographic hash function

SPA Simple Power Analysis

SPN Substitution permutation network

SRAM Static Random Access Memory

SW Software

T2T Task To Task Protocol

TRBG True Random Bit Generator

TRNG True Random Number Generator

TRP Task Resolution Protocol

USB Universal Serial Bus

VHDL Very High Speed Integrated Circuit Hardware Description Language

Index

Erklärung

Ich versichere, dass ich die vorliegende wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, wurden unter Angabe der Quelle als Entlehnung deutlich gemacht. Das Gleiche gilt auch für beigegebene Skizzen und Darstellungen. Diese Arbeit hat in gleicher oder ähnlicher Form meines Wissens nach noch keiner Prüfungsbehörde vorgelegen.

Erlangen,

Thomas Schneider