Diploma work of

Christian Scheurer and

Niklaus Schild

HTI Biel/Bienne

December 12, 2003

1 TABLE OF CONTENTS

2 PREFACE

This page gives you an impression of our paper.

Today we often hear about security and about how bad security is implemented in our IT environments, even if host and network security for Personal Computers and servers have improved steadily over the last decade. However systems tend to break at their weakest link. So IT security is needed wherever communication and data exchange is involved.
There are many embedded devices, which are connected over the Internet and corporate networks. Nowadays, networking security in this area is still as rare as it used to be in the PC environment.

Our diploma work is an attempt to prove that it must not be expensive or difficult to bring network security into embedded devices. We have chosen the Internet Protocol Security (IPsec) suite of protocols to provide standard-compliant, true end-to-end security for networked embedded devices. Benefits from this approach are interoperability with existing systems, strong encryption and authentication based on well-known, trusted algorithms.

In this document, we describe our implementation of basic IPsec support (tunnel-mode with manual keying) for a 16-bit microcontroller. RFC compliance, interoperability with other IPsec solutions and performance constraints – mainly imposed by the used cipher and hash functions - are discussed.

A "Developer's Manual" shows step-by-step how to add our IPsec library to the lwIP TCP/IP stack. This can serve as a guideline for porting the embedded IPsec library to different TCP/IP stacks.

Finally, we look ahead at further steps our implementation would require in order to be a really successful product, ready to be used by an interested community.

If you are unfamiliar with the IPsec terms used heavily in this document, then have a look at the glossary at the end of this paper.

3 BACKGROUND

In this chapter, we explain why we started thinking about implementing IPsec on a 16-bit microcontroller and which good reasons we found to justify such a project.

We realized that there are various IPsec solutions available for high-end 32-bit microcontrollers, but nothing to protect mass-market consumer electronic devices or networked sensors and controllers on network level. Proprietary mechanisms, which are neither compatible nor trustworthy, are often used to connect small devices in a "safe" way.

Adding IPsec (Security Architecture for the Internet Protocol) capabilities to small devices allow sensors to report tamper-proof results or machines to get firmware updates directly from the manufacturers Internet server, hidden from competitors eyes.
An inexpensive 16-bit microcontroller could report a households electric power and water consumption over a common Internet connection, that is already present, or can be temporary established in many buildings.
Due to the IPsec standard compliance, interoperability with existing network infrastructure is provided.

The higher code density on 16-bit microcontrollers helps saving memory and with this it reduces package size. In many non-interactive applications, it is not important if it takes 50ms or 2 minutes to transmit a certain information. It is even possible to add IPsec security to already existing and deployed products, if the remaining resources permit this.

Other reasons for using a 16-bit design could be the already gained experience of the staff with a simple controller and the price difference of development tools.

Calculating the HASH-MACs of the same buffer showed that MD5 should be preferred because it is more than twice as fast as SHA1 (see Table 2).

Algorithm	28 bytes	128 bytes	512 bytes	1500 bytes
MD5	6ms	8ms	14ms	32ms
SHA1	16ms	18ms	40ms	86ms

Table 2: HASH-MAC performance comparison

After studying the times needed to encrypt and authenticate big IP packets (1500 bytes), we concluded that in the worst case, an implementation on our 16-bit processor should be able to provide a round-trip time below one second (see Table 3).

Task	Required time
Encrypt and decrypt 1500 bytes with 3DES	2*222ms
Check and calculate 1500 bytes with HMAC with MD5	2*32ms
Packet processing (pure estimation, probably far too much)	500ms
Total packet processing	1000ms

Table 3: Estimated packet round-trip time

A more realistic example or a more optimized situation (i.e. using DES) would have a much better round-trip time.

What we did not sufficiently analyze in the semester project were the memory requirements when both TCP/IP stack and additional IPsec functionality are present in the microcontrollers limited memory. Since lwIP, DES and HASH code ran smoothly on its own, we hoped to successfully combine both codes in one program. We expected that we would have to optimize all parts oft the software to get everything to run at the same time.

During the implementation studies, we noticed that IPsec includes several nontrivial protocols such as IKE, OAKLEY and ISAKMP.

As soon as key generation and public key cryptography needed to be done on the microcontroller, it again will slow down the system (see *[STEF02]*) and significantly complicate the implementation.

4 PRELIMINARY WORK

Prior to beginning the diploma work, we did some investigation to locate the critical aspects of an IPsec implementation for 16-bit microcontrollers.

As project for the final semester of our computer science studies, we explored the possibility of an IPsec implementation for 16-bit microcontrollers.
We divided the project into the following two parts.
In order to improve the general understanding of networking, security and embedded systems and learn about IPsec, the semester project was split up in studying the IPsec standard and testing critical parts on the target system.
It was necessary to get familiar with IPsec. This implied reading through various RFCs since there exists no single document covering the whole standard. The "Big Book of IPsec RFCs" *[LOSH02]* is a collection of the relevant documents. The key aspects of a basic IPsec implementation include *[RFC]*:

- RFC 2401: Security Architecture for the Internet Protocol (IPsec)
- RFC 2402: IP Authentication Header (AH)
- RFC 2406: IP Encapsulating Security Payload (ESP)
- Various RFCs: Algorithm descriptions for MD5, SHA1 and DES

More sophisticated parts of IPsec such as the Internet Key Exchange (IKE) Protocol, the OAKLEY Key Determination Protocol or the Internet Security Association and Key Management Protocol (ISAKMP) where not discussed in this semester work.

After having become familiar with the IPsec standard, a breakdown of the whole system was necessary. We needed to identify the different modules out of the IPsec architecture so that we were able to characterize the following attributes of the modules:

- Priority
- Dependencies
- Performance sensibility

This breakdown leads us to carefully study the performance of some modules. We already knew that an implementation of IPsec is a feasible task (not to mention the amount of work needed). Implementations for 32-bit systems proved this. A report by Andreas Steffen *[STEF02]* showed that RSA public key operations require from a few seconds to several minutes to accomplish on a 20MHz Intel-compatible 80186 processor.
Nevertheless, we could not find in-depth information on the performance of IPsec-relevant cryptographic algorithms on a 16-bit microcontroller. We decided to run our own tests using a Keil C166 compiler and a Infineon C167CS microcontroller.

An important part of our semester work was to find a suitable IP-stack that is able to carry our IPsec implementation. The lwIP TCP/IP Stack by Adam Dunkels *[DUNK]* of the Swedish Institute of Computer Science has all the desired features: modular design, active community and free BSD-style license.

4.1 RESULTS

Our tests (see *[CSNS03]* for details) focused on the performance of the basic encryption and authentication algorithms. A useful IPsec implementation needs to support at least the DES encryption algorithm and the MD5 or SHA1 HASH-MAC algorithm. The OpenSSL Project's libssl library implements all required algorithms. We decided to extract and port parts of the OpenSSL code to reach our goal. This allowed us to study the performance of the different algorithms running on our slow hardware.

A glance at the runtime needed to encrypt a memory block of a certain size shows that using 3DES on a microcontroller is very costly. For many applications, the security of single DES could be sufficient. Encrypting buffers from different sizes gives an idea how DES performs (see Table 1).

Algorithm	**64 bytes**	**256 bytes**	**1024 bytes**	**1500 bytes**
DES	4ms	15ms	56ms	83ms
3DES	11ms	39ms	152ms	222ms

Table 1: Encryption performance comparison

We decided that an implementation with manual keys would be sufficient to prove our concept and the only feasible way to come up with a result after eight weeks of work.

5 TASK DESCRIPTION

In this chapter, we describe more detailed the assigned problem that needed to be solved during our diploma work.

The goal of our diploma work was to implement the basic functionality of the IPsec standard for the C167 microcontroller. The implementation must be able to tunnel IP traffic with the AH and ESP security protocol. Interoperability tests show that we worked according to the standard and that our product will work with other implementations.
The project did not only consist of coding and testing. A very important part of the project covered planning, organizing, verifying and documenting the actions.

The following steps lead us to a simple but working IPsec implementation:

- Setting up work environment
- Setting up a project plan
- Defining milestones
- Preparation of a test environment for the microcontroller and networking part
- Establishing coding and documentation guidelines
- Preparation of the implementation
- Composing of a test-framework with test cases for all IPsec functions
- Start coding and testing
- Documenting

Date	Tasks done	Next tasks
04.11.2003	Planning and preparation completed	Start of IPsec implementation
26.11.2003	Basic IPsec implementation done	Start with studying of ISAKMP and completion of documentation.
09.12.2003	Code and content of documentation completed	Ready to close down the project.

Table 4: The following milestones were defined

6 DEVELOPMENT ENVIRONMENT

This chapter describes our project environment and the hardware that was needed.

Small networked sensors usually don't require the resources provided by 32-bit microcontrollers but demand reliability, various I/O ports, small package size and low price. IPsec and the user application itself need certain calculation power. A 16-bit microcontroller can fulfill all these requirements and still has some resources left for future enhancements of the application. It is important to know that sending and receiving IPsec packets claims considerable resources of the ALU typically over several milliseconds.

We wanted to use a well-known and stable development environment that let us concentrate on implementing IPsec and did not require endless adjustment and configuration of the environment itself.

6.1 KEIL C166 COMPILER AND UVISION2 IDE

Preliminary experience with the Keil C166 compiler and the uVision2 IDE *[KEIL]* made us confident that this environment is suitable for our task. The maturity of the compiler, its integration in the uVision2 IDE and the uVision2 Simulator proved to be very helpful. The code can be written, compiled, simulated and debugged on the target within one environment.
All code had been tested in the simulator, which was very useful to debug the cryptographic routine and the packet handling. Unfortunately, the CS8900 Ethernet controller could not be simulated. As workaround, we wrote a dummy network driver, which worked on previously dumped network packets represented in arrays of characters.
The forum and support database on Keil's homepage turned out to be particularly useful for questions related to memory layout settings of the linker.

6.2 PACKETYZER – PACKET ANALYZER FOR WINDOWS

Network Chemistry's Packetyzer *[NCP]* is a cleanly designed Windows frontend for the Ethereal packet capture and dissection library and can decode approximately 400 network protocols. It uses the winpcap driver to capture and inject network packets. Captured sessions can be saved. Packetyzer is GPL licensed and thus free to use and available in source code. We found this tool very helpful and recommend having a look at it.

6.3 LINUX PC RUNNING FREES/WAN AND IPSEC_TUNNEL

A Debian GNU/Linux 3.0 (Woody) *[DEB]* PC running Kernel 2.4.18 and FreeS/WAN 1.96 *[FS]* in combination with tcpdump is used to debug the network traffic and operate as peer partner of the microcontroller. Debians apt-get utility eases installation and removal of a large range of stable utilities.
The detailed logging output of FreeS/WAN helps analyzing the dataflow more in-depth than it would be possible with a network analyzer. The ipsec_tunnel *[TRS]* IPsec implementation can also be compiled and installed. Since the source code for both IPsec implementations is available, it is possible to understand how IPsec is processed on the remote side.

6.4 INFINEON C167CS MCU

The Infineon C166 series of microcontrollers originate from the 1990's. Its near-RISC CPU core is tightly coupled with the peripherals. The maturity of the design makes the C166 suitable where a reliable microcontroller with a balance between peripherals, calculation power and price is needed *[HTX]*.
The C167CS-L40M *[INF]* derivate shares the 4-stage instruction pipeline with the traditional versions but operates at 40MHz with an instruction cycle time of 50ms. Plenty of on-chip peripherals such as 2 CAN modules, 16-bit timers, PWM, USART ports, a 16-prioritylevel interrupt system with 56 sources, A/D and D/A converters exist. The internal RAM consists of 3kB fast dual-port integer RAM and 8kB of on-chip XRAM, thus we needed to add external RAM (up to 16MB is possible).

Infineon introduced an improved version with the XC167 series and attested a factor two increase in CPU power (which could be very helpful for IPsec). Unfortunately, we didn't have the possibility to verify these claims.

6.5 CRYSTAL LAN CS8900A ETHERNET CONTROLLER

This rather old 10BASE-T Ethernet controller by Cirrus Logic *[CS89]* is used for many embedded designs due to its low price and uncomplicated use. It has 4kB of internal RAM to assemble fragmented packets and off-load the microcontroller. Erroneous packets are automatically rejected and automatically re-transmitted on collision. The 16-bit controller can be configured to automatically handle padding and CRC calculation. The CS8900A controller eliminates the need for external analog or digital circuits to handle IEEE 802.3 Ethernet. Communication can be done using a DMA modus or over 16 bit or 8 bit ports. An external EEPROM can store the controllers configuration which is then automatically read form there after power up or reset. The lack of 100Mbit compatibility is a drawback of this device and forces other networking devices to support the obsolete 10Mbit Ethernet.

6.6 KEIL MCB167-NET BOARD

The evaluation board of Keil features a C167CS microcontroller with 1MB external RAM and a CS8900A Ethernet controller. We used the Keil PK166 compiler v4.27 and uVision2 IDE v2.39 to develop and debug the IPsec stack. During debugging, a target monitor is loaded into a reserved region of the available RAM (note that the socket for the Flash ROM is empty and thus not used). After having loaded the monitor, the Keil uVision2 IDE loads the application into a free memory location and initiates the debug session using the pre-loaded monitor.
The advantage of this board is the seamless integration in the Keil tool chain and allows even a less experienced developer to get started within less than an hour. The Ethernet controller can be accessed as memory mapped device.

6.7 PHYTEC PHYCORE167-HS/E

Phytec provides an evaluation board with the same microcontroller and Ethernet controller as Keil but comes with more memory (AMD 29F800T 1MB Flash-ROM

and two Alliance AS7C4098-12TC 512kB RAM modules). This board can also be integrated with the Keil tool chain.
The phyCORE167-HS/E board is built on a compact 60x53 mm board and is designed as subassembly for OEM products in small- to medium volume series.

7 APPROACH

In this chapter, we explain how we tackled the implementation of the IPsec standard.

The implementation can be divided into the following parts.

First of all there needed to be a mechanism, which allowed us to add IPsec functionality to any other TCP/IP stack. We call this the traffic interception. Traffic interception is somewhat stack dependent because it is the interface from our IPsec library to the TCP/IP stack. Even tough its implementation is different on each TCP/IP stack, the presented concept of data flow should be applicable to other TCP/IP stacks.

The IPsec engine is the central part, which does the whole standard conform processing of the incoming and outgoing IP traffic. It uses a set of databases (SPD and SAD) to determine the flow of the IP packets. The main processing is then done in the AH and ESP module.

A small cryptographic library contains all the functionality used to encrypt, decrypt or to authenticate the packets.

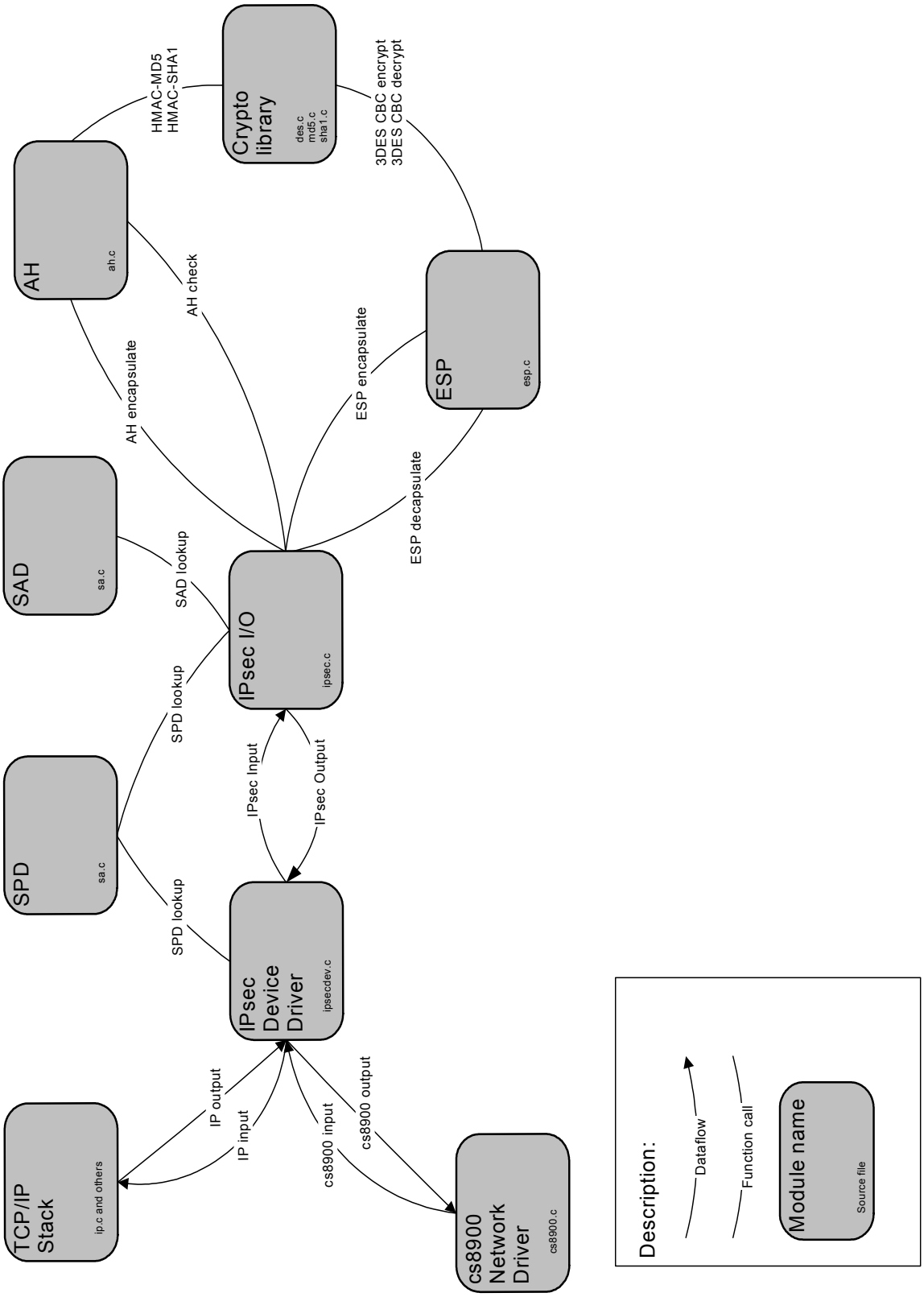


Figure 1: The whole IPsec system with the dependencies

7.1 TRAFFIC INTERCEPTION

IPsec needs to control or modify all incoming and outgoing IP packets. For this reason, we must provide a traffic interception mechanism, implemented as "IPsec device driver". This part is closely connected to the TCP/IP stack and the transport device driver.

Any inbound data is forwarded to ipsecdev's ipsecdev_input() function. Depending on the protocol field in the packet header, the entire packet is forwarded to the IP protocol stack. If the packet could be identified as belonging to the suit of IPsec protocols, it is transferred to the IPsec library. Pure IPsec specific processing, such as applying ESP de-/encapsulation or AH de-/encapsulation is done within the IPsec library. After these steps, the original IP packet is rebuilt by applying new offsets and packet length to the pbuf structure (see section 7.3.2 on page 27). Then the clear-text packet is passed up to the ip_input() function.

For outbound packets, all IP based protocols forward their data to ipsecdev_output(). Here the decision is made whether the packet needs IPsec processing or not. Depending on the appropriate Security Association, AH or ESP functionality will encapsulate the packet. After these steps, the packet is forwarded to the appropriate physical device driver (we use a CS8900 driver) and sent over the wire.

A design goal is to put most TCP/IP stack specific code into the ipsecdev and keep it as fast and simple as possible. All IPsec specific processing is concentrated in the ipsec module.

The Security Policy Database (SPD) can be accessed from the ipsecdev and ipsec module. This database contains all rules required to decide how to handle packets, which have security associations but also how to handle non-IP traffic. There are several possibilities: any non-IPsec packet can be forwarded to the default protocol handler (in order for connections from non-IPsec clients are accepted) or any non-IPsec packet can be dropped immediately without wasting CPU time on further analysis.

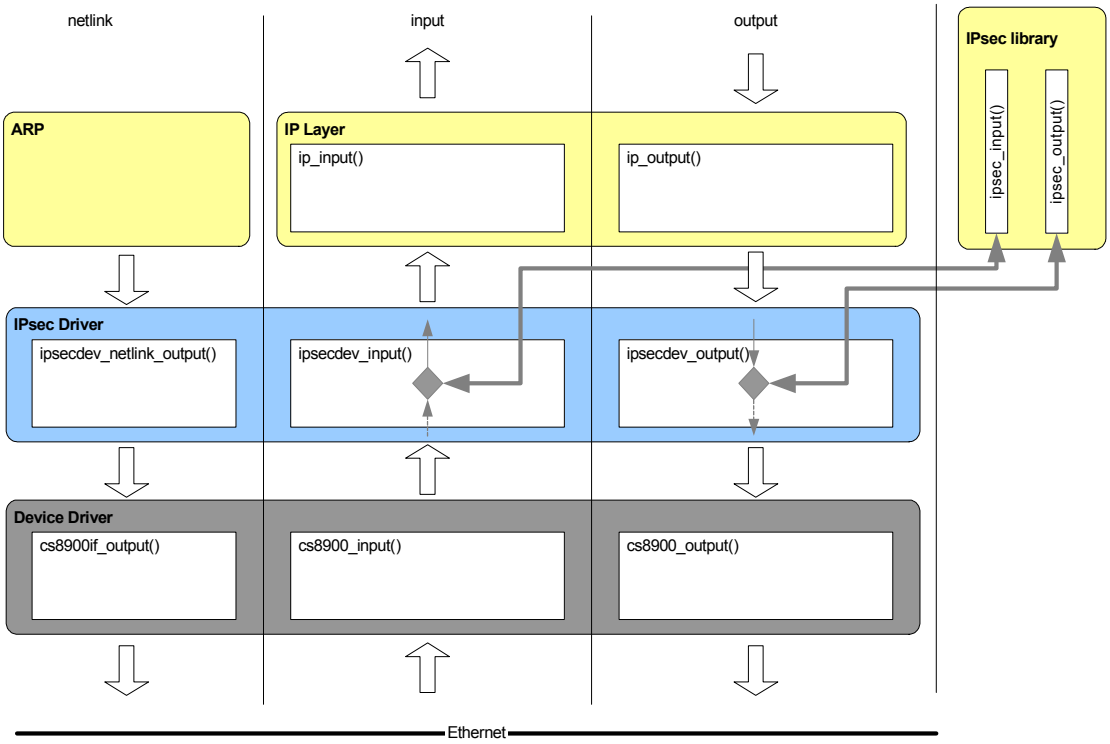


Figure 2: IPsec trafic interception

In order to modify the lwIP's default data flow, functionality was needed.
Since our goal was not to modify lwIP source code, we decided, as already mentioned, to implement a virtual device driver. This device driver has the same interface as the real hardware network driver (see Figure 2 on page 18).
When lwIP installs its device drivers, our IPsec device also has to be installed with the function netif_add(). The parameters of this function are:

- IP address of the device
- Netmask
- IP address of the gateway
- Pointer to the device initialization call-back function
- Pointer to the output-function: the function to which the packet is passed after processing

Section 10.6.1 on page 70 shows, how netif_add() function can be used to install the IPsec driver after the real network device driver, so that all IP packets are routed through the IPsec device.

These two lines do the following:
The CS8900 driver is installed. Its initialization function is cs8900if_init().

Whenever the cs8900 driver receives data from the Ethernet, it should forward the packets to the function ipsecdev_input(). This is the input function of the IPsec device.
Then the IPsec driver is installed. Its initialization function is ipsecdev_init(). Because the input dataflow already has been setup by the previous netif_add() function, the output dataflow must be established. This is done in the IPsec device initialization function, where the output function of the CS8900 driver is first saved and then overwritten. This is done by overwriting the output function in the network interface structure with the output function of the IPsec device. So whenever data comes down the IP stack, data is passed to the output function stored in the network interface structure of the CS8900 driver, where the address of IPsec device output function is stored.
After IPsec processing has been done, the saved and overwritten output function of the CS8900 driver is called to forward the packet to the Ethernet.

7.2 Security Policy and Association Management

IPsec needs one database to control the flow of the IP packets. This database is called Security Policy Database. It simply describes which traffic requires IPsec processing and which traffic does not.

The other database, the Security Association Database, holds data about each configured connection and also defines how the traffic must be processed if the policy in the SPD defines the APPLY rule for a certain packet.

The SPD can be seen as a persistent database while the SAD is only temporary for each connection. In our simplified environment the SAD could also be static because a dynamic standard conform way to add SA's is not implemented.

7.2.1 Basic concept of Security Associations

IPsec needs the Security Policy Database and the Security Association Database to process packets correctly.

The SPD defines the packets, to which IPsec needs to be applied. To guarantee that each packet is processed the right way, each IP packet leaving or entering

the system must be checked against the SPD. We call this action the SPD lookup. This lookup does nothing except compare the selectors from the database with the ones from the packet. The SPD lookup delivers back the following results:

- BYPASS: this packet is forwarded directly to the next protocol layer without applying IPsec
- DISCARD: this packet is discarded, it will be dropped
- APPLY: this packet requires IPsec processing

If the result of a SPD lookup is BYPASS, the unmodified packet is forwarded to the next protocol layer. This is particularly useful if certain protocols such as ICMP should not be protected by IPsec or communication with non-IPsec hosts must be concurrently possible.
The DISCARD rule is returned when the intention is not to process this packet. If this is the case, the packet will be dropped. This means that we simply delete the packet (free its allocated memory) instead of passing it to the next protocol layer. It is possible to use this feature to build a primitive firewall.
IPsec processing is only needed if the result of the SPD lookup is APPLY. Whenever a packet matches an SPD entry whose policy says APPLY, then there must also be an SA that describes exactly how the packet has to be processed.

A successful SPD lookup provides us with a pointer to the SP over which we can access the SA using a pointer stored in the SP structure.

In a dynamic environment this SA can be created using IKE. As soon as the SPD finds out that there is no current SA available, it will trigger an IKE function which is responsible for the negotiation of the required parameters. The packet can be processed only after Security Association parameters are successfully negotiated.

In a more static environment, where IKE functionality is missing, an SA cannot be set-up on the fly. In such a case, the SA needs to be created at system start-up so that IPsec is ready to process traffic.

Source Address	Source Netmask	Destination Address	Destination Netmask	Protocol	Source Port	Destination Port	Policy	SA
192.168.1.5	255.255.255.255	192.168.1.3	255.255.255.255	UDP	0	500	BYPASS	
192.168.1.2	255.255.255.255	192.168.1.3	255.255.255.255	TCP	0	80	APPLY	0
192.168.1.0	255.255.255.0	192.168.1.3	255.255.255.255	ICMP	0	0	APPLY	1

Table 5: Example of an SPD table, with SA pointers to Table 6

One important feature of the SAD/SPD is that they must be defined per network interface and also per stream direction. This signifies that each network interface and each flow direction has its own set of SPD and SAD. It is important to mention this because the IPsec layer must be able to determine the network interface where the packet came from.

Index	Destination Address	Destination Netmask	SPI	Protocol	Mode	Encr.	Auth.
0	192.168.1.3	255.255.255.255	0x1012	ESP	TUNNEL	3DES	HMAC-MD5
1	192.168.1.3	255.255.255.255	0x1013	AH	TUNNEL	0	HMAC-MD5
2							

Table 6: Example of SAD table, referenced by Table 5

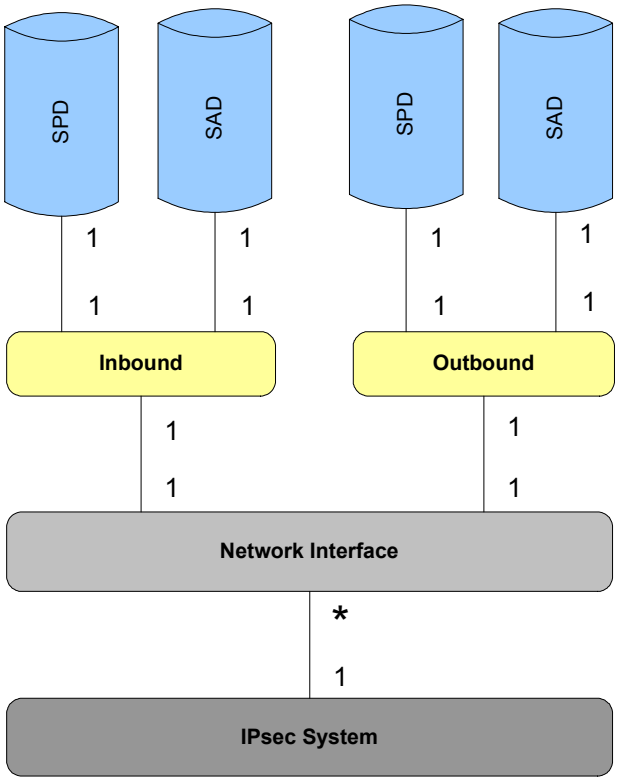


Figure 3: Relations of IPsec components in a IPsec system

7.2.2 IMPLEMENTATION

At system startup, before the initialization of the network interfaces, an initial configuration for the SPD must be set. We did this in a configuration file where other general settings for the IPsec system are defined. The configuration fields were not only those for the SPD since we also needed to set-up the appropriate SA. One configuration entry contained both the configuration for the SPD and the one for the corresponding SA (thus all SA's are stored in the SAD). A possible simplification was to merge these databases, since the RFC[1] does not define how the databases have to be implemented. Such a solution had been possible, but in our opinion the distinction between SPD and SAD made sense, because a later IKE implementation was expected to be much easier if there were at least two physically different databases. Adding and removing SPD entries and SAD entries had to be possible in a dynamic way, so that an SA can be added and removed during runtime. This is required for a later IKE implementation.

[1] RFC 4201, Section 4

The next two paragraphs contain more details on the processing of inbound and outbound packets. Section 7.7 on page 36 describes how the databases can be maintained in a more dynamic way using IKE.

7.2.3 OUTBOUND PROCESSING

1. When a packet leaves an IPsec configured system, the very first step is an SPD lookup, a determination of how the packet must be processed. When the policy says APPLY, the IPsec process continues. Otherwise the function passes the packet to the device driver or returns to the TCP/IP stack without doing anything.
2. Now (in case of an APPLY policy) the packet must be processed according to the SA that was given back by the SPD lookup. When no SA is available, IKE functionality would be invoked. If no IKE is available or the IKE negotiation fails, the packet must be discarded.
3. In case of a valid SA being available the packet is encapsulated either in an AH- or ESP-header.
4. After the new IPsec packet has been built, it must be sent out on the appropriate device.

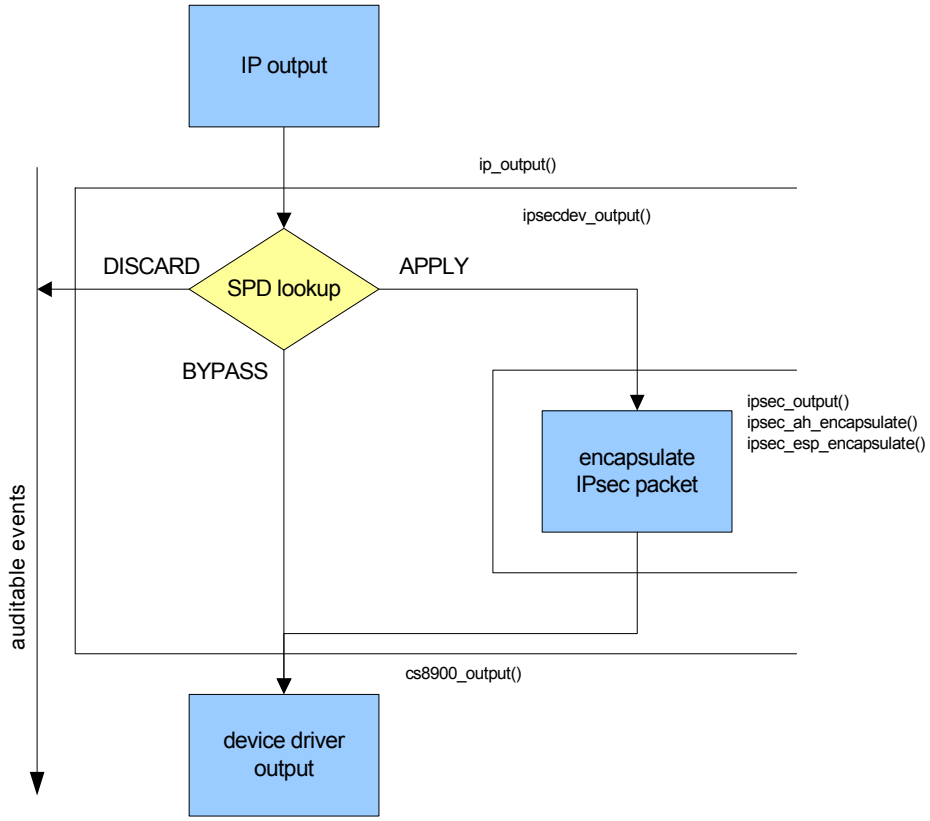


Figure 4: SPD outbound processing

7.2.4 INBOUND PROCESSING

1. Inbound processing is somewhat different because an incoming IPsec packet already has an SPI, which allows a direct lookup in the SAD table. The reason for using the SPI is straightforward. The incoming IPsec packet may be encrypted and so the SPD lookup, which must be performed on the inner packet data, cannot be performed. The SAD lookup would directly give back an SA if one was found. If no SA is found, then the packet must be discarded. Note: at this stage it is not allowed to trigger some IKE functionality to negotiate a new SA with the communicating peer. The communicating peer should have done that already.
2. With the valid SA we are now able to process the packet properly. In inbound processing this corresponds to decapsulation in case of ESP or integrity checking in case of AH.
3. After decapsulation, we have a clear-text or authenticated packet. To be sure that the right SA was applied to the packet, an SPD lookup has to be performed now on the clear text packet. This check will confirm that there was a valid SPD entry for the SA, which was used. This must be done

because a packet could have been sent with a fake SPI to force the proper processing of the packet. If the SPD lookup fails or points to a different SA, the packet must be dropped.

4. After the IPsec packet has been decapsulated, it can be passed on to the TCP/IP stack.

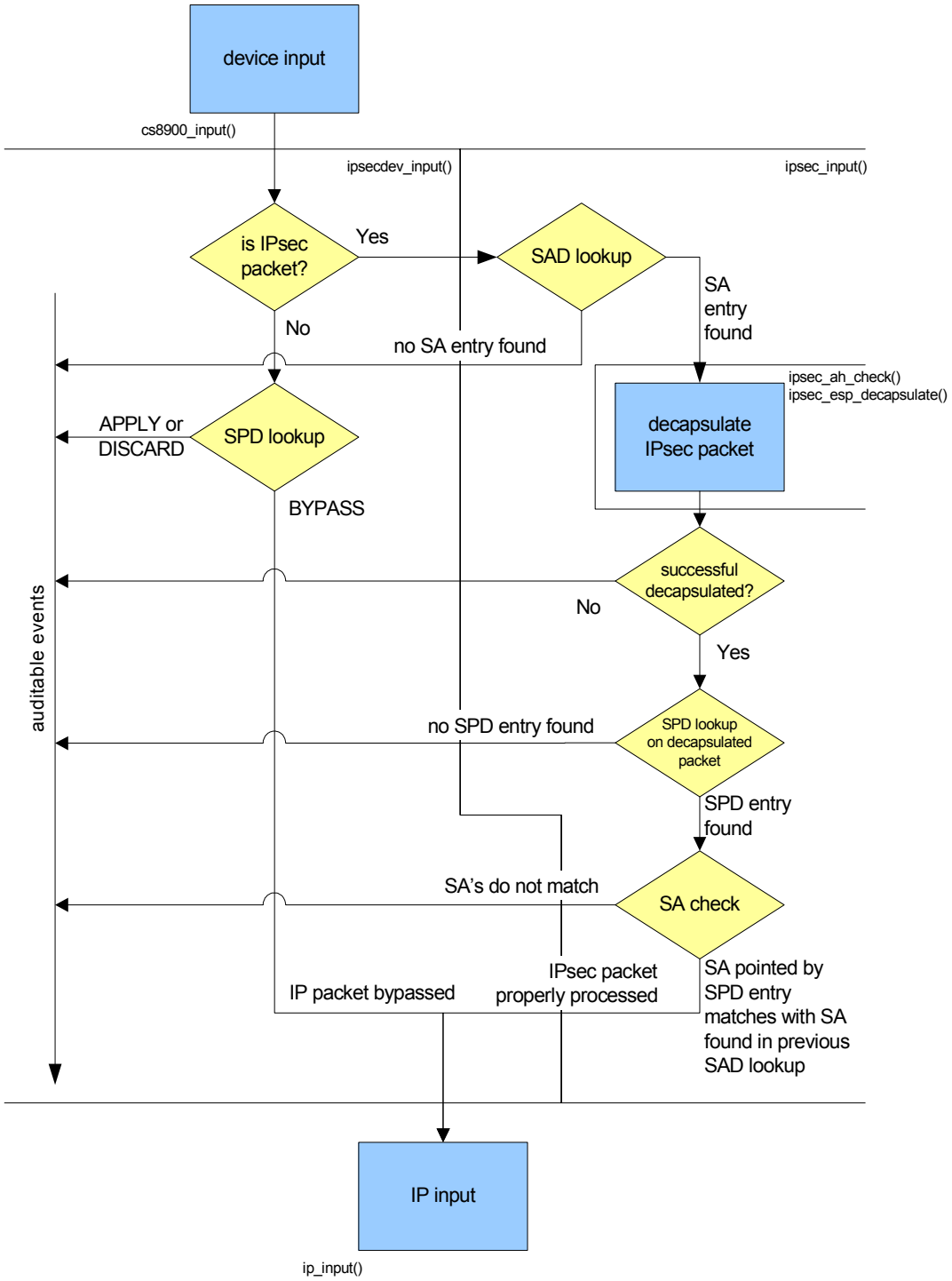


Figure 5: Inbound SPD/SAD processing

Whenever negative decisions are made in the processes above, a log message is generated. This corresponds to the auditable event mentioned in the RFC 2401.

7.3 MEMORY MANAGEMENT

IPsec processing modifies the content and size of both inbound and outbound traffic, thus a facility to manage memory is required.

7.3.1 MEMORY MANAGEMENT STRATEGY

An efficient memory management strategy is necessary to avoid further reduction of performance on embedded devices:

- no copying / moving of memory blocks (zero copy maxim)
- packets are passed by reference (using pointers)
- cryptographic operations are performed in-place
- packet size can be adjusted dynamically

Most operating systems and TCP/IP stacks already contain memory managers. By separating memory allocation calls from the IPsec core routines, the replacement of the memory manager does not affect the IPsec core. Calls to malloc()/free() are exclusively allowed in the platform-dependent "ipsecdev" device driver. Inside the IPsec library, packets are represented as simple char arrays.
All packets must be stored in a contiguous block of RAM memory, with spare room both in front and behind the packet. Contiguity guaranties simple and fast access for cryptographic algorithms when applied on multiple blocks. Additional room is needed to dynamically adjust the packet size: that is adding AH or ESP headers and the new (outer) IP header for the tunnel mode in front of the packet or appending padding bytes after the inner packet.

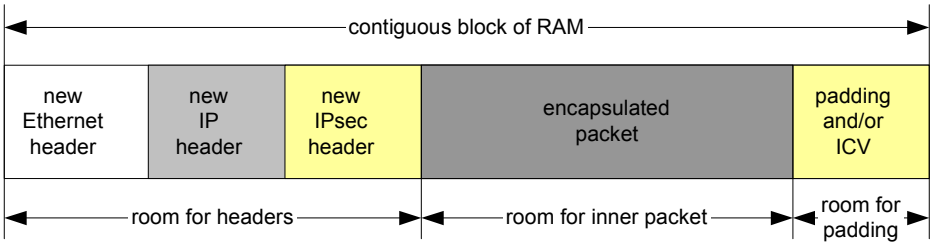


Figure 6: A contiguous block of RAM

It is important that the size of the contiguous block of RAM does not exceed the network's Maximum Transmission Unit (MTU), usually up to 1500 bytes for IP over Ethernet[2].

With inbound packets, the outer IP header, AH/ESP header and padding bytes become obsolete after IPsec processing. A pointer to the start of the inner packet can be passed to higher protocol layers and copying of data can be avoided. After successfully processing the packet, the whole contiguous block of RAM must be freed.

When outbound packets are assembled by the TCP/IP stack, room for the Ethernet header, outer IP header, AH/ESP header must be reserved right in front of the original (inner) packet. Since the implemented cryptographic algorithms can only be applied on contiguous, fixed-size data blocks, room for padding needs to be left after the original (inner) packet. After encapsulating the packet using the IPsec library routines, a pointer to the start of the outer IP header is passed to the physical device driver. When the packet becomes outdated, the complete contiguous block of RAM must be freed.

7.3.2 WORKING WITH LWIP PACKET BUFFERS

Fortunately, the lwIP TCP/IP stack is already equipped with a memory structure that is particularly suitable for working with dynamically resizable buffers: the pbuf packet buffer structure.

[2] RFC 894

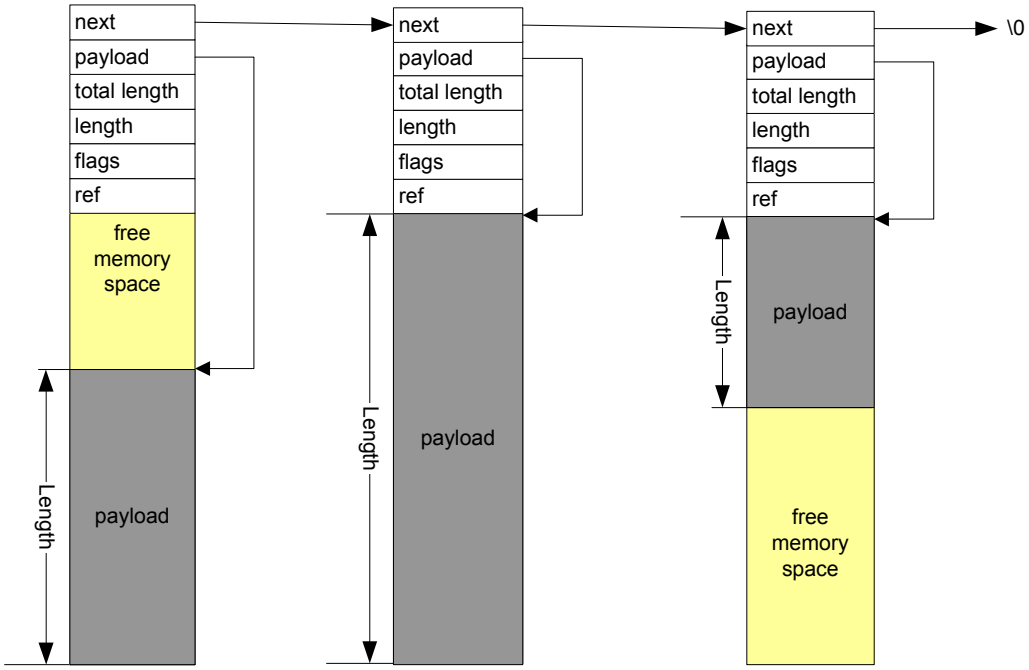


Figure 7: Packet spanning over chained pbufs must be avoided

A singly linked list of pbuf data structures holds all packets on their way up or down the stack. A new pbuf is allocated when data arrives on the physical network interface or an application starts sending out data. The pbuf allocation routine mainly initializes an empty pbuf header by setting the payload pointer to a free part of memory and setting up all other fields of the structure. With a one-line-modification in the pbuf_alloc() routine (see section 10.5.2 on page 68), we ensure that the payload size of the pbuf is as large as the MTU of the system. The returned payload pointer is initialized with an offset of about 100 bytes (IPsec header, outer IP header and Ethernet header). To add an AH/ESP header, the payload pointer can be decremented by the AH/ESP headers size and will still point to valid memory. Adding the padding bytes can be done by simple appending these to the inner packet (also increasing the length field), as without leaving the allocated memory space.

It is obvious that memory is wasted due to over dimensioned rooms in front and behind the inner packet (see Figure 6 on page 26), especially for small packets such as the ICMP ping service. Thus the amount of concurrently held packets is below the theoretical maximum.

The advantages of this scheme are simplicity, robustness and speed. There is certainly room for further improvements, i.g. by using different pbuf sizes for different protocols.

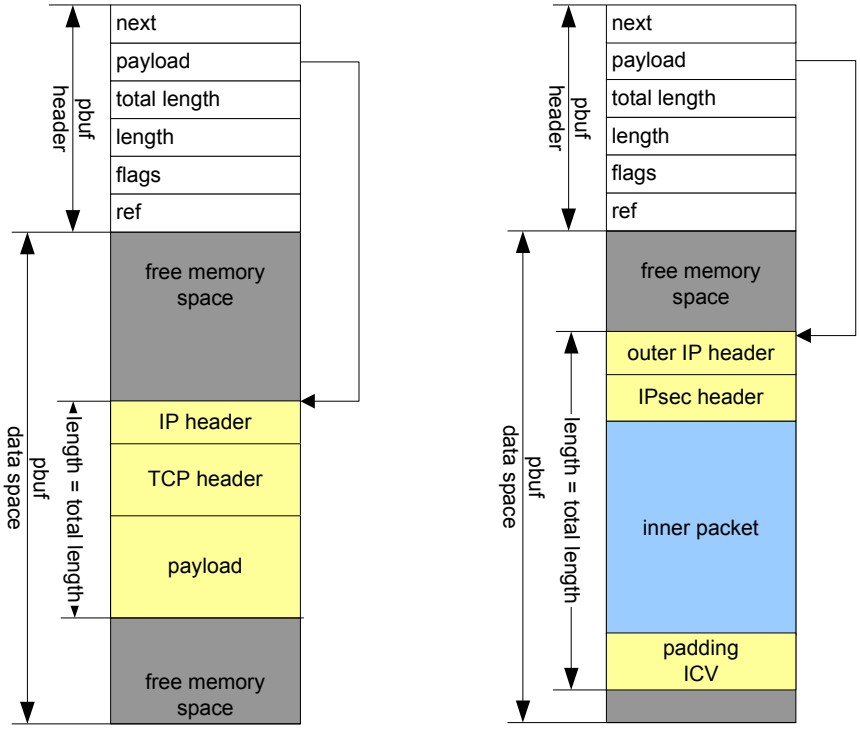


Figure 8: Pbufs before and after processing

7.4 ANTI-REPLAY PROTECTION

Limited sequence integrity is provided by the IPsec anti-replay protection[3]. It helps detect packet reordering and replay of previously dumped traffic. This check of the packet sequence number must only take place in combination with authentication in order to avoid maleficent manipulation of the sequence number. A window size of up to 32 is available in embedded IPsec. If a new, unseen packet arrives, the corresponding bit in the window is set. The bit can only be set once, or else the packet is recognized as already seen and therefore rejected. The sequence number of the last packet is also stored to check if the sequence numbers are monotonically increasing.

The following functions check and update the anti-replay window:

- **ipsec_check_replay_window():** makes a quick check of the sequence number without updating the window. This function is called prior to authentication. Packets that are obviously out of order do not get a further check, thus CPU time is saved and invalid packets cannot alter the anti-replay window.

[3] RFC 2401, Appendix C

- **ipsec_update_replay_window():** after authentication, this function must be called to verify and update the anti-replay window. If the packet has not yet been seen, its corresponding bit is set in the window. In case it is a new, unseen packet with a sequence number right next to the window, the anti-replay window is shifted (updated).

7.5 AH PROCESSING

The IP Authentication Header (AH) provides data origin authentication and integrity for IP packets. Authentication is done by the well-known HASH-MAC[4] MD5[5] and HASH-MAC SHA1[6] algorithms. These are the algorithms requested by the standard. As seen in section 4.1 on page 7, MD5 should be preferred because its performance is much better than that of SHA1.
In the SA configuration, the user has the possibility to choose the algorithm that shall be selected for authentication. Because the integrity of an AH packet can always be verified, the anti-replay check is performed on each packet.
If one considers that ESP also supports integrity and authentication, one may think that there is no need for AH. This is not true because the authentication and integrity check of AH is a bit more sophisticated. Authentication in AH covers more fields of the packet than ESP does.

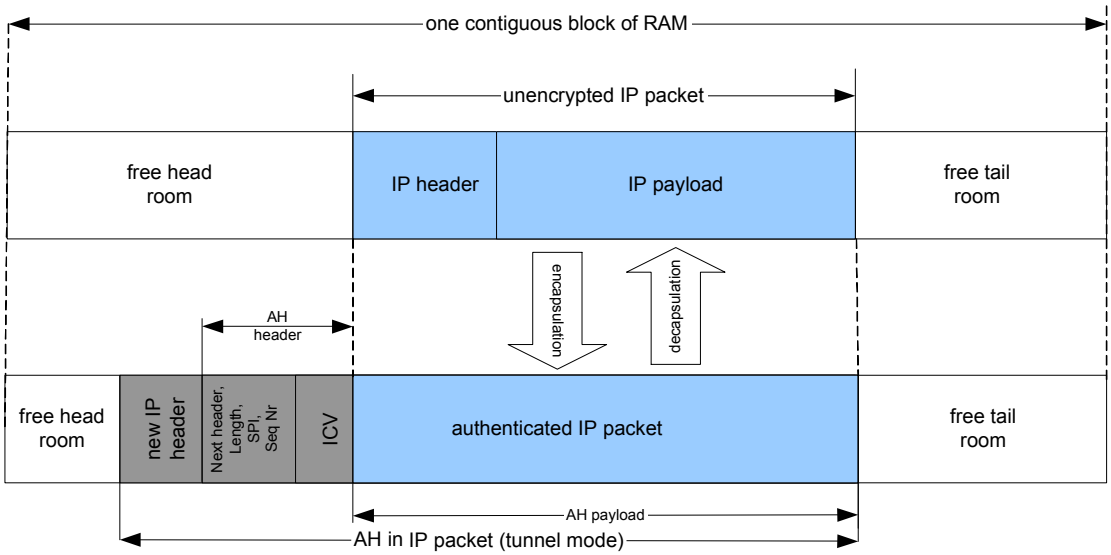


Figure 9: A clear text packet and a packet encapsulated in AH

[4] RFC 2104
[5] RFC 1321
[6] RFC 3174

AH processing can be split up into inbound and outbound processing. These two parts are implemented in:

- **ipsec_ah_check():** verifies the integrity of the AH packet by applying a HMAC with given key and performs an anti-replay check.
- **ispec_ah_encapsulate()**: sets up a new AH and IP header in front of the inner packet and calculates its integrity check value.

The next two paragraphs describe more detailed how AH processing was implemented.

7.5.1 AH INBOUND PROCESSING

AH inbound processing was implemented with the function ipsec_ah_check().
This function gets the following input parameters:

- Pointer to the IP packet that must be verified
- Pointer to the SA that describes how the packet must be processed

After AH processing is done, two variables are passed back:

- Offset to the decapsulated IP packet (relative to the address of the input IP packet)
- Length of the inner IP packet

The processing itself described step-by-step:

1. In order to check the integrity and the authentication of the packet, the ICV must be calculated. The ICV calculation in AH also covers the outer IP header. In this header there are so-called mutable fields, which change their value while they are sent across the network. Those fields (Type of Service, Offset, TTL and checksum) must first be set to zero. The ICV fields in the AH header must be backed up and zeroed, so that later comparing remains possible. It becomes clear that AH authentication also covers the source and destination address of the outer IP packet.
2. The packet is now ready to be verified and the integrity check value can be calculated over the whole packet. The SA determines the appropriate algorithm and key.
3. The calculated ICV can be compared with the one saved in the first step. Processing continues only if the calculated ICV matches the original one.
4. The authentication of the packet is now verified and the anti-replay check can be performed. If it is successful, the sequence number

(stored in the SA) is incremented. Finally, the offset and packet length are passed back.

7.5.2 AH OUTBOUND PROCESSING

AH outbound processing was implemented with the ipsec_ah_encapsulate() function. This function gets the following parameters:

- Pointer to the IP packet, which must be encapsulated.
- Pointer to the SA, which defines how the packet must be encapsulated.
- Source IP address, describing the tunnels source address
- Destination IP address, describing the tunnels destination address

After AH processing has been completed, two variables are passed back:

- Offset to the encapsulated IP packet (relative to the address of the input packet)
- Length of the encapsulated IP packet

The processing itself described step-by-step:

1. First of all a new AH header is placed in front of the IP packet, leaving a gap between the inner IP header and the AH header. This gap is later used to place the ICV. The AH header fields: next header, length, SPI and sequence number are added.
2. After the outer IP header has been constructed, only the source and destination address, version, header length and total length are set. The other fields are set to zero as a preparation for the ICV calculation. Padding is not required because the packet is already aligned[7].
3. The integrity check value can now be calculated and placed into the gab between AH header and inner IP header.
4. After the ICV has been calculated, the zeroed fields are now filled with the appropriate values.
5. Finally, the offset and the packet length are passed back.

7.6 ESP PROCESSING

An Encapsulating Security Payload (ESP) header is designed to provide a mix of security services for IP packets. In our ESP implementation we support both

[7] RFC2402, section 3.3.3.2.1

encryption and authentication. Encryption is done by the widely used 3DES algorithm, which is applied in CBC mode. Pure DES is also implemented but there was not enough time to fully test it. For authentication we use HASH-MAC MD5 and HASH-MAC SHA1.
With the SA configuration the user has the possibility to configure the security features that are to be applied to ESP processing.
When the user also selects authentication, the anti-replay service can guarantee that resent IP packets, or packets entering the system out of the replay-window are discard.

An IPsec packet, which was built with ESP (in tunnel mode), looks the following way:

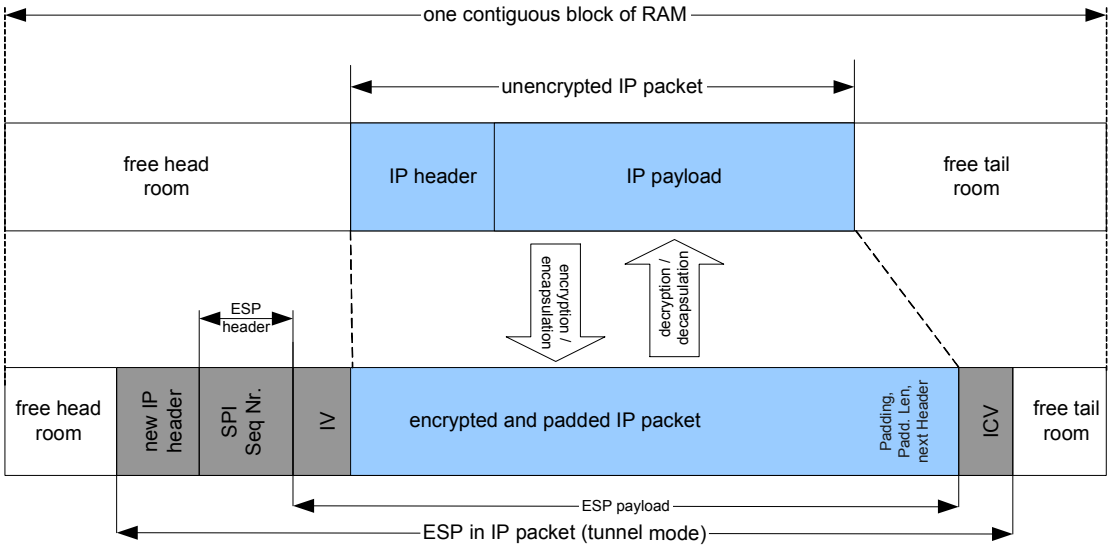


Figure 10: Clear-text packet and a packet encapsulated in ESP

ESP processing can be split up into inbound and outbound processing. These two parts are implemented with

- **ipsec_esp_decapsulate():** checks the content of the ESP header, and optionally verifies the authentication and anti-replay and decrypts the packet with the given key and initialization vector.
- **ipsec_esp_encapsulate():** sets up a new ESP header, encrypts the packet and optionally calculates the integrity check value (for authentication).

Both functions also setup the outer IP-header, which is needed for tunnel mode.

The next two paragraphs describe in more details how ESP was implemented according to RFC 2406.

7.6.1 ESP INBOUND PROCESSING

ESP inbound processing is implemented with the function ipsec_esp_decapsulate(). This function receives the following input parameters:

- Pointer to the IP packet which must be decapsulated
- Pointer to the SA which describes how the packet must be processed

After ESP processing has been done, two variables are passed back:

- Offset to the decapsulated IP packet (relative to the address of the input IP packet).
- Length of the decapsulated IP packet.

The processing it self described step-by-step:

1. A check in the SA structure indicates whether authentication need to be checked or not. If an authentication algorithm is specified within the SA, the ICV must be calculated and compared with the one stored at the end of the ESP packet. The ICV is calculated over the whole ESP header, IV and encrypted payload. Processing continues only when the packets ICV matches our recalculated one.
2. In the next step we have to decrypt the packet. The decryption algorithm and the secret key can be accessed over the SA. Because the packet was encrypted in CBC-mode, the IV must be copied out of the ESP packet. The IV is stored between ESP header and encrypted payload. The decryption happens in-place, so no copying must be done.
3. Since the IP packet has now been extracted out of the ESP packet, we can perform some sanity checks before terminating ESP processing. In our implementation we verify that the total length field in the extracted IP packet is within our valid range (20-1500 bytes).
4. Before everything is done the sequence number counter in the SA is incremented, and optionally the same is done with the anti-replay window. To let the caller of the ESP function know about the location and the size of the extracted IP packet, the offset and the packet length are give back.

7.6.2 ESP OUTBOUND PROCESSING

ESP outbound processing is implemented with the function ipsec_esp_encapsulate(). This function receives the following input parameters:

- Pointer to the IP packet which must be encapsulated.
- Pointer to the SA, which describes how the packet must be processed.
- Source IP address describing the tunnels source address (from outer IP header).
- Destination IP address, describing the tunnels destination address (from outer IP header).

After ESP processing has been completed, two variables are passed back:

- Offset to the decapsulated IP packet (relative to the address of the input IP packet).
- Length of the decapsulated IP packet.

It is very important that there is enough room before and after the packet. ESP encapsulation adds 36 bytes (20 byte outer IP-header, 8 byte ESP header and 8 byte IV) in front of the packet and 2 till 16 bytes at the end. One must also be aware of the fact that after ESP encapsulation a Ethernet header will be added.

The processing itself described step-by-step:

1. The first step of encapsulation is to test whether the decremented TTL field of the IP header reaches zero. If this is the case, the packet must be discarded in order to prevent endless straying of packets.
2. Then we have to calculate how much padding must be added to fulfill the requirements of the encryption algorithm. In our case (DES/3DES) we must have the payload aligned to 8 bytes because the block size of DES/3DES is 64-bit (8 bytes). The right amount of padding bytes is added at the end of the payload. The fields: padding length and next header, are appended right after the padding.
3. Encryption is now performed according to the settings in the SA. After encryption, the used IV is copied in front of the encrypted payload.
4. ESP header is now added in front of the IV. Inserted are a incremented sequence number and the SPI taken out of the SA.
5. As was done in inbound processing, the SA must be checked to see if authentication is enabled. If this is the case, then the ICV must be calculated according the SA's settings. The ICV, which is calculated over

the ESP header, the IV and the encrypted payload, is copied at the end of the payload.

6. Now the outer IP header can be constructed using the tunnels source and destination address given as input arguments to the function. The TOS field is copied from the inner IP header.
7. Finally, the offset and the length are passed back, so that the caller can update it's data structure (in our case the pbuf), where the packet is stored.

7.7 ATTEMPT TO IMPLEMENT ISAKMP

This paragraph is somewhat different, because no preliminary work was done. So we will first start with a short introduction to IKE/ISAKMP. Unlike other paragraphs in this chapter, the implementation itself is not discussed. This paragraph should be a helper for a later IKE/ISAKMP implementation.

A comfortable IPsec implementation requires a dynamic way to generate Security Associations. A further task of our work was to find out how such an implementation could be done for a 16-bit embedded system. It quickly became obvious that there is not enough time to implement ISAMKP. The corresponding standards[8] alone would have confronted us with as much literature as the remaining IPsec standard. We estimated that twice the amount of work would be needed to add IKE.
On the other side, not having IKE run on our system has the following disadvantages:

- Interoperability is reduced, because some IPsec systems (Windows 2000) work only with IKE or other dynamic SA generation mechanisms.
- Administration of many IPsec devices is not comfortable.
- The system must be configured at compile time and cannot be changed later on or must use proprietary configuration mechanisms.

7.7.1 WHAT IS ISAKMP?

ISAKMP (Internet Security Association and Key Management Protocol) is a framework combining the security concepts of authentication, key management and security associations in order to establish the required security.

[8] RFC 2407, 2408, 2409

Packet formats are defined to negotiate, establish, modify and delete SAs for various kinds of security services, independent of the layer (e.g. IPsec, TLS). ISAKMP does not rely on a special key generation, encryption or authentication mechanism. Since it is just a framework, other protocols are used for these specific tasks.

IKE (Internet Key Exchange), which is often mentioned together with ISAKMP, is a specific key exchange mechanism using part of the Oakley protocol and part of SKEME in conjunction with ISAKMP to obtain authenticated keying material. The keying material is what we need for our SAs to properly process AH and ESP packets.

7.7.2 WHAT WAS IMPLEMENTED

As already mentioned, we decided not to implement ISAKMP. But during development we soon realized that having a more dynamic way for configuring SAs would be helpful. We also wanted to prove that our implementation is ISAKMP-ready.

On top of the lwIP stack we wrote a small application, which listens on UDP port 500, like an ISAKMP daemon. This daemon simulates our ISAKMP SA negotiation. Over UDP we are now able to send SA's over the network to the microcontroller. The microcontroller itself is able to add them on-the–fly to the SPD and SAD table. After such an entry is added, the IPsec system behaves like a new SA was established.
In our case, the new SA must be sent manually from a Linux console using our IPsec administration tool. Using this tool, SAs can also be deleted, so that the SPD/SAD configuration can be manipulated whenever required.

It is important to note, that the use of this utility in production destroys the whole security of IPsec. This tool can only be used for testing and demonstrating.

7.7.3 REQUIREMENTS FOR A LATER IKE IMPLEMENTATION

Before one can start implementing IKE, the whole set of RFC must be read and understood:

- RFC 2407: The Internet IP Security Domain of Interpretation for ISAKMP

- RFC 2408: Internet Security Association and Key Management Protocol (ISAMKP)
- RFC 2409: The Internet Key Exchange (IKE)

It is obvious that the implementer must also be familiar with the general IPsec RFC 2401, so that he understands how the things are glued together.

Cryptographic functions must be added to support Diffie-Hellman key exchanges and to allow asymmetric encryption.
This kind of cryptography is very time consuming on a 16-bit microcontroller. Asymmetric key generation may take several minutes *[ZILE]*. To prevent unexpected performance penalty, tests should prove that bad performance does not make an implementation impossible.

During implementation of IPsec within lwIP, restrictions appeared which will have an impact on an ISAKMP implementation. The lwIP implementation with IPsec can be seen as one single running process on the microcontroller. Whenever the microcontroller will try to negotiate SA over ISAKMP, it will have to stop processing the IP packet passed to us by the lwIP stack. Stopping in the middle of packet processing would block the whole stack (TCP/IP as well as IPSec), so the ISAKMP daemon would not be able to communicate over the network. Putting such packets into a queue until they got their proper SA could be a workaround for this deadlock.

8 TESTING AND QUALITY ASSURANCE

This chapter explains our test framework that enabled us to test early and often. We continuously controlled our work in order to be able to provide good quality of coding.

Our test framework consists of three main parts. They are all described in the next paragraphs (*see also [BEAT]*).

8.1 STRUCTURAL (WHITE BOX) TESTING

8.1.1 WHY STRUCTURAL TESTING?

This type of testing can be done early in the project phase. It involves testing of a single unit of software. In our case these units are our modules. The tests should check all the important elements of the module. We were able to add all structural tests to one unit so all the structural tests can be performed in a batch process (all tests at the same time). This enabled us to run the batch often and regularly. The tests can also be run after each small change in the code, so we always knew that our software was still working in a consistent state.
Structural testing is also called white box testing because the exact kind of functions and code tested is known.

8.1.2 HOW WE IMPLEMENTED STRUCTURAL TESTING

Structural testing means running a test for each implemented function. We decided to apply the eXtreme Programming paradigm "test driven development". This forced us to write down testing procedures first. After the test procedures were implemented were able to start coding the actual problem.

Test-driven development helped us to:

- First think about what the tested function really needs to do.
- Define the input and output data types and value ranges.
- Be able to run a test after having finished coding the function.

We wanted to go even a bit further. We first wrote down all the test functions needed to test the whole IPsec implementation. This helped us thinking about the whole programming structure of our end product. The test functions were added to our main test routine. At first, all these test functions were just stubs, without any implemented code. They were then implemented with a predefined interface, so that they could easily be called from the main test routine. The test routine of a certain module was responsible for the testing of the whole structure of the module.

The module test function needed to implement the following features:

- Implementation with a predefined interface and naming convention.
  - Interface: int function(void)
  - Name: m*odulename*_test()
- When the test function is returned without an error, we can be sure that the module is implemented properly and without bugs.
- It is able to do many tests of the same function if required. Input data may be random (i.e. for value range testing) or statically programmed.
- It is able to call many different functions of the module.
- The tests need to be reproducible.
- As long the test function is not implemented, it must print out that the module / module test function has not yet been implemented.

When the module test functions were implemented according to these rules we were able to run the module test function out of one main (executable) function. The first time we ran the test function we got a list of printed messages. Each line corresponded to one module test function and it showed that the module had not yet been implemented.

Our goal for the end of the project was for each module test function to print a line indicating that all test were successful.

8.2 FUNCTIONAL (BLACK BOX) TESTING

8.2.1 WHY FUNCTIONAL TESTING?

The second step in our testing concept was functional testing. In this phase we tested the functionality of our IPsec implementation. The reason why it is also

called black box testing is because we didn't care how the tested functionality was implemented and what functions are needed to implement this functionality. It was a kind of abstraction: we just wanted to know whether certain functionality did its job properly. This kind of test is not based on modules but on functionality. As an example we wanted to know if an AH packet can be verified properly. In order to be able to run such a test we needed functions of many modules (HASH-MAC, AH, etc...).
Functional testing can always be started with small tests. The more modules were implemented, the more functional tests could be done. The objective of the test was to check whether the modules work properly with each other. The further the project proceeded the more complex the tests became. The last functional test was a test that checked the whole function of our IPsec implementation. After having run the functional test properly we expected to be able to run the implementation on the desired hardware in the real world.

For functional testing we used similar rules that were need for structural testing. Failing functions need to print out the input, output and expected output data in order for the error to be reproducible.

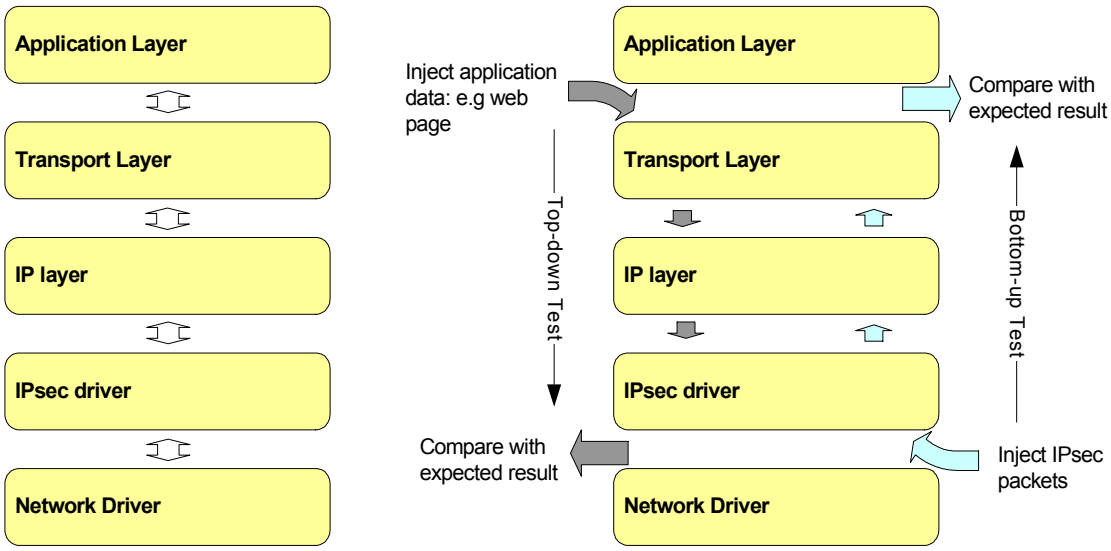


Figure 11: Normal dataflow and dataflow of functional tests.

As wanted side-effect, each functional test performs many different tests on the various involved modules.

8.2.2 How we Implemented Functional Testing

Since functional testing was implemented as the project proceeded, we did not specify the structure of functional tests. Because functional testing demanded a lot of memory for static test data, we were not able to make one executable. We split up the functional tests into functional units.
There were two possible approaches: Top-Down and Bottom-Up tests. Both could be implemented. They were independent to each other and they tested the functionality in different ways.
In our IPsec implementation there are two directions of data flow. One direction is from the network device upwards the stack, until data reaches the application. The other direction is downwards the stack, starting at a normal TCP or UDP packet which gets encapsulated and injected into the Ethernet.

Top-Down Tests

Top-down starts at the top of the stack and goes down the whole stack. The following is an example of what a top-down test could look like:

- Lets assume that there is a function called esp_encapsulate(char *data, char *esp_packet). This function creates an ESP packet out of some user data (e.g. TCP packet).
- We capture an IPsec stream from two communicating IPsec peers. They use an already working IPsec implementation like FreeS/WAN or ipsec_tunnel. We know the keys and algorithms from the configuration and we know the plain text data of the packet, because a well-known sequence of packets has already been captured.
- We can now feed the esp_encapsulate() function with the known plain text data. This could for example be the content of a small web page packed into a TCP packet.
- The function must now return an ESP packet looking equal or very similar (maybe there are fields which may be different i.e. sequence numbers, etc) to the original packet we captured from the IPsec stream.

With such a test we were able to ensure that the whole functionality of generating an ESP packet was implemented properly. Of course we still didn't know whether it was possible to check if we were able to verify and decapsulate an ESP packet. This was done in the Bottom-Up tests. With such a test we can check a part of

the functionality of our implementation. Another test could have done the same for AH packets.

Bottom-Up Tests

This test starts at the bottom of the stack. The last piece of software right before the hardware is the network interface driver. A bottom-up test could look as follows:

- From an IPsec stream we capture one or multiple IPsec packets.
- From the configuration we know all the parameters.
- We now write a dummy-driver, which is able to simulate the driver and/or the hardware. The dummy-driver is noting but a simple I/O function, which feeds the appropriate IPsec function with a captured IPsec packet.
- We can now see if our implementation is able to find the appropriate SAD (Security Association Database) entry for this packet. An we are able to check if the implementation is able to find out if the packet needs to be processed by IPsec or not.

8.2.3 DESCRIPTION OF A FUNCTIONAL TEST

A major functional test is the integration of the IPsec functionality and the TCP/IP stack.

This test requires a consistent sequence of packets fed to the IPsec layer. It can be archived by dumping packets from reference scenarios, such as a PC is sending 4 ICMP ping messages to the microcontroller board. A virtual device, "dumpdev" simulates the physical device driver (CS8900) and injects the dumped sequence packet by packet. The simple dumpdev device is suitable for testing both IPsec and TCP/IP stack inside the uVision2 simulator.

8.3 AUTOMATED TESTING

All the tests that have been described so far are executed manually. If long-term test should be performed, manual execution becomes almost impossible. For such cases, a few test scripts were written. The general idea of this undertaking was to have the possibility to carry out tests over night. These scripts were written quickly.

In the first series of test scripts, we just wanted to test ping (ICMP) or http (TCP) sessions during the whole night. Within a small bash script, pings and http sessions were executed within a loop. A counter, incremented during each round, indicated the number of passed tests. Obviously, the ping session was done with the ping utility. The http transfers were done with telnet. Parallel to this test, the logging screen of the PC could give us more information on failures.
Using such a test, thirty to sixty thousand IPsec packets were sent and received during a one-night test. Normally, all these packets were processed without one failure.

After the first series of test had shown that our code worked, more sophisticated tests were needed. Here the test-utility (described in section 7.7.2 on page 37) was a great help. Thanks to this IKE simulation tool, we were able to change the microcontrollers configuration (SPD and SAD) during run-time. Our testing PC, equipped with FreeS/WAN also has the possibility to change the configuration (SPD and SAD) during runtime. With this flexible configuration, we were able to write a test script, which allowed us to test all IPsec features within a one-night test. This test script executed the following steps for each IPsec feature:

Restart FreeS/WAN. This was necessary because FreeS/WAN did not clean up the routing table properly.

```
scuol #: ipsec setup restart
```

Reset the microcontrollers SPD and SAD configuration

```
scuol: # admin FLUSH INBOUND 192.168.1.3
scuol: # admin FLUSH OUTBOUND 192.168.1.3
```

Load the new SAD and SPD configuration for the microcontroller

```
scuol: # admin ADD ah_config.conf 192.168.1.3
```

Load the new configuration for FreeS/WAN

```
scuol: # ipsec manual –-up ah_config
```

Perform the desired tests, i.e. a series of pings or a certain amount of http sessions.

All these steps were done for:

- AH with HMAC-MD5
- AH with HMAC-SHA1
- ESP with 3DES
- ESP with 3DES and HMAC-MD5
- ESP with 3DES and HMAC-SHA1

8.4 INTEROPERABILITY TESTING

A useful implementation must be able to talk with other IPsec products. Interoperability means that our implementation should be able to establish IPsec tunnels with other peers in the network.

First of all we analyzed different IPsec products, which we could use for the interoperability tests. We had to choose a product that is easily available and suits as a good reference. We had to take a product that was already widely used and thus well tested. We soon focused on the following IPsec implementations:

- ipsec_tunnel: a simple and free implementation for Linux
- FreeS/WAN: a free Linux IPsec implementation
- Windows IPsec: on a Windows 2000 operating system
- PGPNet: another free IPsec implementation for Windows and Mac

The next paragraph describes the products we looked at more closely.

8.4.1 TESTING ENVIRONMENTS

8.4.1.1 ipsec_tunnel for Linux

The ipsec_tunnel developed by Tobias Ringström *[TRS]* is an elegant, minimal implementation of IPsec tunnel functionality for the Linux 2.4 kernel series. It is designed as kernel module and the ipsec_tunnel module itself does not require any kernel patching. The only requirement is the presence of the CryptoAPI code in the kernel source (some kernels are shipped with it already, others require the addition of strong cryptography by applying the International Kernel Patch). After successfully having loaded the ipsec_tunnel module, an unconfigured network device "ipsec0" becomes available. The "ipsecadm" utility is used to create, list and modify SA records for ipsec_tunnel. Selecting a previously defined

Security Association to create a tunnel is the final step in configuring ipsec_tunnel.
The standard Linux network tools "ifconfig" and "route" can be used to establish a route through the ipsec0 device.

Since ipsec_tunnel relies on the encryption and digest functionality provided by CryptoAPI, all common algorithms are supported:

Ciphers	**Digests**
3DES, DES, AES, Blowfish, Twofish, Cast5, DFC, IDEA, Mars, RC5, RC6, Serpent, null	MD5, RIPEDEM160, SHA1, SHA256, SHA384, SHA512

The supported protocols of ipsec_tunnel are ESP and ESP with authentication. Support for dynamic configuration is NOT available.

As long as manual keying is supported by the other party, ipsec_tunnel looks promising regarding to interoperability. There are already successful interactions with FreeS/WAN and OpenBSD documented.

8.4.1.2 FreeS/WAN

FreeS/WAN was interesting for us from the beginning because it is widely used in the Linux community. There is a lot of documentation available and in case of problems or additional interest we would be able to have a look at the source code, which of course is freely available. FreeS/WAN also supports many features, which we wanted to implement in our IPsec library.
With FreeS/WAN we would be able to perform the following tests:

- AH tunnels
- ESP tunnels
- Encryption with 3DES
- Authentication with MD5
- Authentication with SHA1
- Manual Keying

Manual keying was a very important feature for us because we were not yet sure whether we would be able to implement automatic keying. Besides this, manual keying simplifies debugging because it's independent from complicated key generation and negotiation.

By default, FreeS/WAN is a very restrictive implementation and so the developers do not like to implement features that decrease the security. The following features can only be added using patches:

- Single DES encryption
- Null encryption

We did not want to lose time with patching the FreeS/WAN source and recompilation of kernels. The supported features were rich enough to provide a good test environment.

FreeS/WAN also provides good debugging facilities. This helped us to analyze problems and observe what happened in the IPsec kernel. Unfortunately, documentation is poor and bad structured so that getting starting with FreeS/WAN turned out to be a time consuming task.

8.4.1.3 Windows 2000

We also had a look at the Windows IPsec implementation. Of course Windows would be a spectacular testing environment because of its wide spreading. Windows 2000 and Windows XP have built-in IPsec support.
Soon we found out that manual keying is not a possible configuration. The advantages of Windows IPsec would be:

- Support of Null encryption (good for debugging)
- Support of single DES encryption (good for performance)

Because we were not able to find out whether and how manual keying can be done with windows, we had to omit this implementation for our tests.

8.4.1.4 PGPnet

PGPnet is an easy to install and configure VPN software for Windows and Macintosh computers. It implements the IPsec and IKE protocols and supports OpenPGP keys for authentication in addition to X.509.
The tested version 7.0.3 cannot handle manual keying and therefore cannot be used to test the simple embedded IPsec configuration.

8.4.2 SECURITY TESTS

This chapter describes how the security of our IPsec implementation can be tested.

Implementing according to the standard guaranties certain security features, but nothing assures that these features really work and that coding was done properly. With the intention of proving good security in our implementation, certain attacks to our implementation and its results are discussed and explained below.

Not all security features can be tested easily. For example, we were not able to prove the security of ESP encryption. The user of our IPsec system has no other choice but to trust the DES standard and carefully inspect its implementation.

8.4.2.1 Packet was altered during transmission

Scenario: A bad guy may want to modify the content of some network packets and somehow manages to alter the content, lets say of a HTTP transmission. If the packet was authenticated, he will not be able to recalculate the proper ICV because he doesn't know the required secret key to update the ICV in the packet.

Proper IPsec processing: After the packet has entered the IPsec system it is processed by either the AH or ESP module. When authentication is enabled (AH or ESP with HMAC), IPsec recalculates the ICV using the secret authentication key. The recalculated value will not match the one stored in the packet and therefore the packet will be discarded.

Verification of this threat: If a packet with some changed bits either in the packet itself or in the ICV value is injected into an IPsec stream our implementation will discard the packet with the following message:

```
ERR ipsec_ah_check:          -2 AH ICV does not match
```

or

```
ERR ipsec_ah_check:          -2 ESP ICV does not match
```

8.4.2.2 Non-IPsec packet, which should be one

Scenario: A bad guy may try to send non-IPsec packets to our IPsec enabled host. He may hope that a non-IPsec packet (which should be one according to the SPD) will reach the IP stack without any intervention.

Proper IPsec processing: When a clear text packet enters the system, it is checked against the SPD table (see Figure 5 on page 25). If the policy says APPLY, the packet will be discarded (because the packet should be encrypted and IPsec decapsulation can't be applied to non-IPsec traffic). The goal of this test is to prevent that non-IPsec traffic bypasses the IPsec engine as requested by the RFC[9].

Verification of this threat: The incoming packet will fail on the above-mentioned SPD lookup. The packet will be discarded with the following message:

```
AUD ipsecdev_input:   3: POLICY_APPLY: got non-IPsec packet which should be one
```

8.4.2.3 Packets are Resent

Scenario: A bad guy may want to resend a certain IPsec packet

Proper IPsec processing: This threat can only be caught if authentication is activated. This is the case when AH or ESP with authentication is used. Otherwise, the packet passes IPsec processing without any problems. In case of activated authentication, the anti-replay check will find out that the packet has already been sent, because the packet's bit in the bit-mask is already set to 1.

Verification of this threat: It is necessary to resend a packet that has already been processed and thereby verify that authentication is turned on. Our IPsec implementation will discard this packet with the following message:

```
AUD ipsec_ah_check        :  7 : packet rejected by anti-replay check
```

or

```
AUD ipsec_esp_decapsulate :  7 : packet rejected by anti-replay check
```

[9] RFC 2401, Section 4.4.1

8.4.2.4 Packets that are out of the window

Scenario: A bad guy may want to disturb IPsec processing by sending IPsec packets with a high sequence number. This could lead the anti-replay mechanism to shift the anti-replay window in such a way, that the normal IPsec packets seem to arrive out of the window (their sequence number would be too low to be accepted).

Proper IPsec processing: The anti-replay check (see section 7.4 on page 29) is only performed when authentication is turned on. This is the case when AH or ESP with an authentication algorithm is used. Before the packet is authenticated, a preliminary check of the sequence number is done to avoid wasting CPU time for authenticating packets that are obviously out of sequence. Only if the sequence number is valid (not yet obvious since it is within the window), the packet is passed to the authentication function. When authentication has passed, it will again be checked for validity of the sequence number before the packet is marked as seen and anti-replay window is shifted.
If the sequence number was altered, the integrity check would fail and the packet would be discarded.

Verification of this threat: If a forged sequence number is injected into an IPsec stream, our IPsec implementation will discard this packet with the following message:

```
AUD ipsec_ah_check        : 7 : packet rejected by anti-replay update
```

or

```
AUD ipsec_esp_decapsulate : 7 : packet rejected by anti-replay update
```

8.4.2.5 Packets with a bad SPI

Scenario: A bad guy may want to send IPsec packets with a forged SPI. If he monitors the IPsec traffic of a certain host, he is able to set a packet with a valid sequence number and a valid SPI, which could lead the IPsec system to properly process the packet. The correct sequence number is only required when authentication is activated. For example, the sequence number is not tested in the case of a ESP without authentication

Proper IPsec processing: Such an attack would be possible if only an SA lookup was performed on incoming IPsec packets. The SA lookup uses only the

outer destination address, the IPsec protocol and the SPI to determine the appropriate SA [10]. However, the standard requires that after an inbound IPsec packet has been processed properly, an SPD lookup must be performed [11]. A successful SPD lookup gives back a security policy with a pointer to the SA describing how the packet must be processed. To prevent such attacks, the SA pointer from the policy must now point to the same SA that was used to process the packet (SA returned by the SA lookup, see Figure 5 on page 25).

Verification of this threat: Inject a forged IPsec packet into an "ESP 3DES only" packet stream. The inner IP packet's fields may be modified in such a way, that there is no matching entry in the SPD. Our IPsec implementation discards such a packet with the following message:

```
AUD ipsec_input             :      6: SPI mismatch
```

8.4.2.6 Non-IPsec packet, which should be one

Scenario: A bad guy may try to send non-IPsec packets to our IPsec enabled host. He may hope that non-IPsec packets (which should be one according the SPD) will reach the IP stack without any intervention.

Proper IPsec processing: When a clear text packet enters the system, it is checked against the SPD table (see SPD lookup in: Figure 5: Inbound SPD/SAD processing). If the policy says APPLY, the packet will be discarded (because the packet should probably be encrypted and IPsec decapsulation can't be applied on non-IPsec traffic anyway). The goal of this check is to prevent that non-IPsec traffic bypasses the IPsec engine as requested by the RFC[12].

Verification of this threat: The incoming packet will fail on the above-mentioned SPD lookup. The packet will be discarded with the following message:

```
AUD ipsecdev_input:   3: POLICY_APPLY: got non-IPsec packet which should be one
```

[10] RFC 2401 section 4.4.3
[11] RFC 2401 section 5.2.1
[12] RFC 2401, Section 4.4.1

9 Implementation Review

Close to the end of our diploma work, the feature list grows and embedded IPsec seems to be become a nice little IPsec implementation. Regarding RFC conformance, first priority was given to the features requested for our diploma work. We tried to work according to the RFC's as often as possible. Since there is no 100% RFC conform IPsec implementation we were not able to fulfill this task either.
The next paragraphs show which features were actually implemented and which features were not yet implemented or require more testing.

9.1 Facts and Figures

9.1.1 Features

Our implementation is still a prototype but all the features that were requested for our work are quite well tested. We created test cases for almost each feature. Several functional tests were run over night to detect memory leaks. The nightly tests usually processed between thirty to sixty thousand packets without crashing and failing. Such tests showed with reasonable certainty that features listed below are implemented in a quite stable manner. Utilization in a busy real-life network environment would probably show some not yet known shortcomings that were not apparent up to now due to the clean lab environment.

Our implementation has the following features:

- Dynamic Security Policy management
- Dynamic Security Association management
- AH protocol
- ESP protocol
- Support for AH with HMAC-MD5 and HMAC-SHA1
- Support for ESP with 3DES, 3DES-HMAC-MD5 and 3DES-HMAC-SHA1
- Support for tunnel mode
- Anti-replay protection
- Support for manual keying
- ISAKMP ready
- Interoperable with Linux FreeS/WAN and ipsec_tunnel

9.1.2 PERFORMANCE

Up until the end of the project, we did not exactly know how good our IPsec implementation would perform. In the preliminary work (see Table 3 on page 8), we just estimated a round-trip time. Here are now the facts.

Performance was tested in a network where only two testing machines were connected. A release version of our IPsec implementation was flashed into the Phytec (see section 6.7 on page 13) board. As already explained we chose a Linux machine running FreeS/WAN for testing. The scenario remained the same for all tests. The two test machines were loaded with the appropriate configuration to process the test IPsec packets properly. To measure round-trip time, we performed ping series from the Linux machine to the microcontroller. After a series of ten pings, we took the average round-trip time as the reference time. Series with pings of different sizes showed how performance behaves on different packet sizes. Using ping (ICMP echo request) to measure the round-trip time has the following advantages:

- Easy round-trip time measurement (use of ping utility)
- Almost no TCP/IP stack overhead, because echo request/reply are very simple to implement
- Easy change of packet size because the Linux ping utility has an option to specify the packet size

We created such a test with measurement of the average round-trip time for each different IPsec configurations and for many different packet sizes (see Table 7 on page 54).
In order to have a neutral reference, we also measured the round-trip time of normal (non-IPsec) packets. The increasing time is caused by the device driver, which first must copy the echo request from the Ethernet device into a pbuf. From there on almost no processing is done until the echo replay has been copied back from the pbuf into the Ethernet controller. Table 7 shows the average round-trip time together with the size of the encrypted/authenticated payload.

IPsec mode	round-trip time [ms]						
	64 bytes	128 bytes	256 bytes	512 bytes	768 bytes	1024 bytes	1280 bytes
non-IPsec	5.2	5.7	6.3	8.3	10.1	11.8	13.3
AH HMAC-MD5	19.3	22.2	27.7	38.5	49.7	60.6	71.6
AH HMAC-SHA1	39.3	45.8	59.3	85.6	112.6	139.1	165.8
ESP 3DES	60.2	101.7	185.4	351.7	518.5	685.2	851.9
ESP 3DES HMAC-MD5	73.7	117.8	205.4	381.2	557.2	733.1	908.7
ESP 3DES HMAC-SHA1	93.3	141.5	237.1	428.9	620.6	812.3	1011.3

Table 7: IPsec performance Table

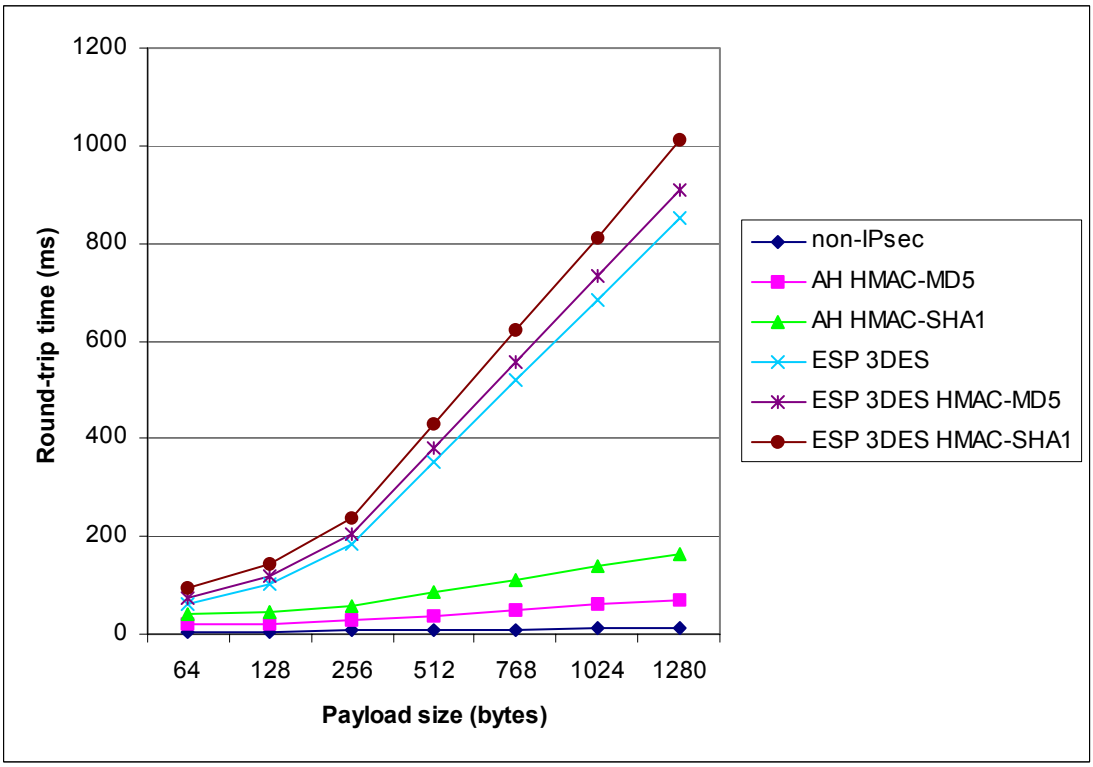


Figure 12: IPsec performance measuring visualized

Not much comment is needed since the figure can easily be interpreted. In some aspects, the results were as expected. The relative difference between the different cryptographic algorithms, correspond to the performance test, which were done in *[CSNS03]*. If the ESP 3DES with HMAC-MD5 results were extrapolated, the round-trip time would marginally exceed the one-second mark. This fact is somewhat surprising.

The results from this chapter show, that depending on the chosen cryptography, various types of security can be added with various amounts of cost.

If only authentication and integrity is required, IPsec packets can be processed in quite a short time. Using strong encryption, a noticeable performance loss must be accepted.

9.1.3 MEMORY REQUIREMENTS

We analyzed the linker output file and got a meaningful overview of the memory used in each module. It is interesting to see how bloated cryptographic functions taken from OpenSSL perform. The heavy usage of macros and the code design thought for workstations have a bad impact on code size. A significant performance improvement is expected by using a microcontroller-specific implementation. Dmitry Basko's DES page *[DBDES]* shows that DES can be efficiently implemented on a similar microcontroller in about 2.8kB of code in Assembler respectively in 4kB of code when ANSI C is used.
Not present on Figure 13 are the 1-2kB RAM needed for a dynamic SAD/SPD databases.

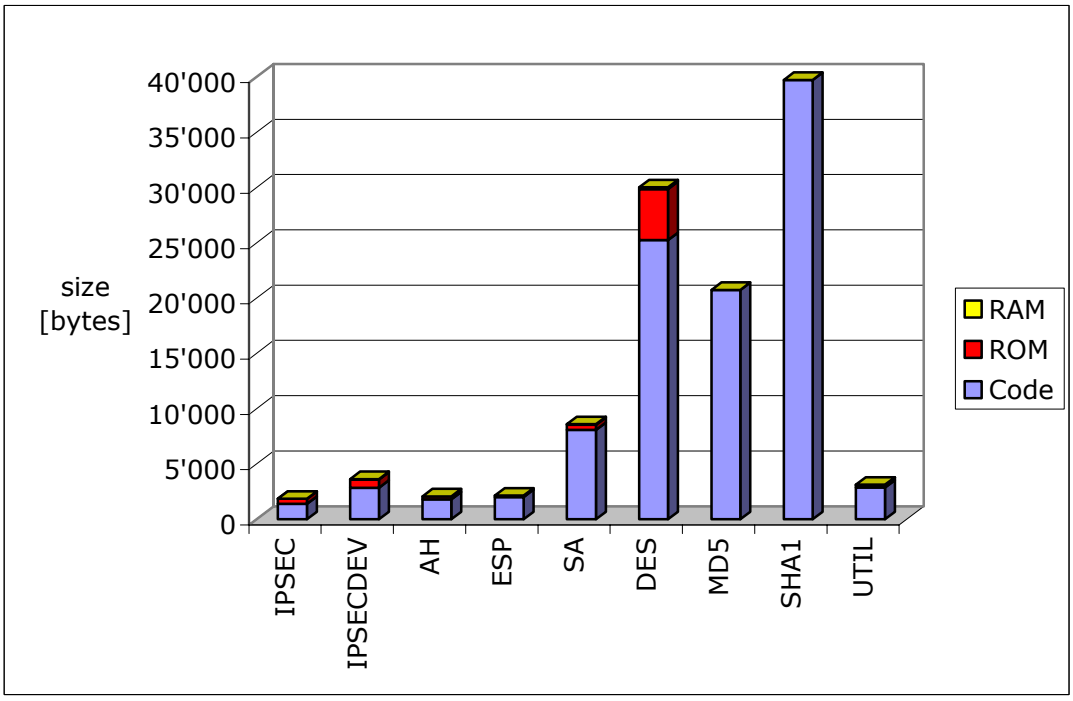


Figure 13: embedded IPsec module size diagram

Table 8 shows that over 81% of all code is DES, MD5 or SHA1.

Module name	Code [bytes]	ROM [bytes]	RAM [bytes]
IPSEC	1400	470	0
IPSECDEV	2860	737	46
AH	1798	272	8
ESP	2016	131	8
SA	8078	479	53
DES	25246	4631	162
MD5	20738	16	4
SHA1	39726	20	4
UTIL	2882	209	36

Table 8: Size of embedded IPsec modules

A comparison with the lwIP CVS STABLE version from October 23, 2003 shows that the embedded IPsec library modules can compete in regard to code size. If the cryptographic functions are replaced by optimized versions, the complete code size of lwIP (including TCP and UDP) with the embedded IPsec would fill a 64kB ROM. The RAM requirements depend on the number of simultaneously active network connections. The MEM, MEMP and PBUF modules allocate RAM in huge char arrays and allocate it to active connections, buffered packets and the application itself. A minimum of 16kB RAM could be sufficient if the embedded device operates under light system load.
A reasonable system configuration with room for the application itself and the possibility to add improvements such as IKE/ISAKMP in the future would be 256kB ROM with 128kB RAM.

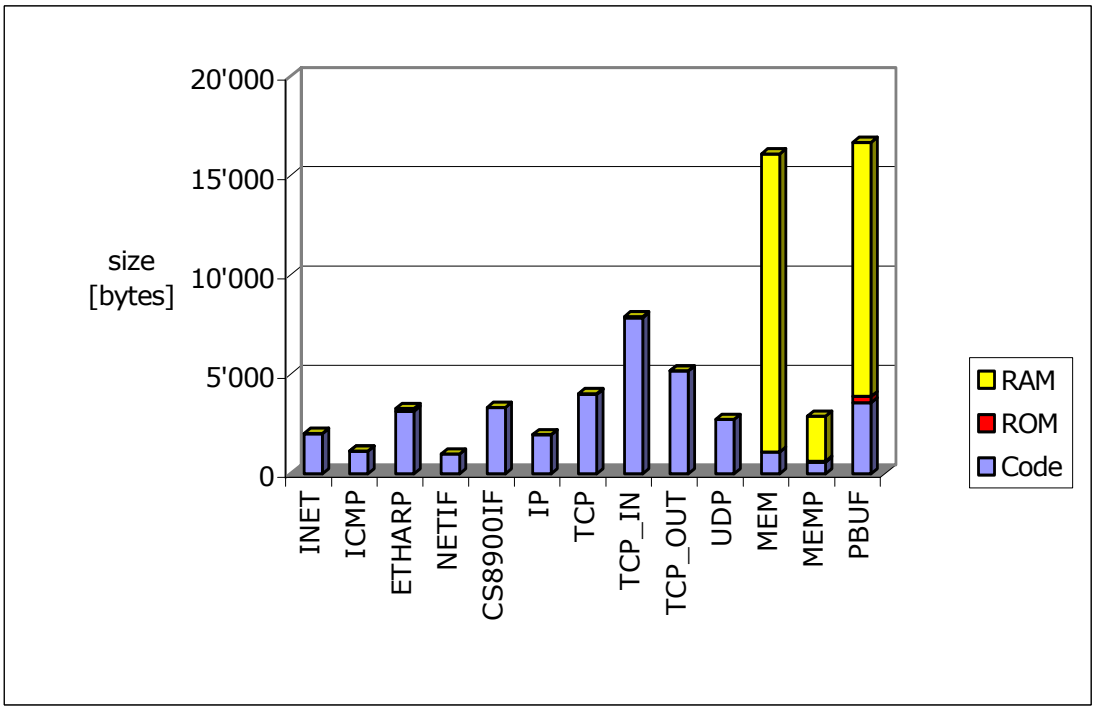


Figure 14: lwIP module size diagram

Module name	Code [bytes]	ROM [bytes]	RAM [bytes]
INET	2022	0	16
ICMP	1140	0	0
ETHARP	3142	140	6
NETIF	986	0	10
CS8900IF	3330	6	2
IP	1958	0	16
TCP	3992	13	27
TCP_IN	7856	0	46
TCP_OUT	5178	0	0
UDP	2724	0	8
MEM	1092	0	15017
MEMP	594	40	2267
PBUF	3578	308	12799

Table 9: Size of lwIP modules

9.1.4 INTEROPERABILITY

Interoperability is a key feature of each IPsec implementation. Our embedded IPsec supports only manual keying. Therefore, only very simple and particularly flexible IPsec solutions support this keying mode. We were able to successfully establish connections to two Linux implementations. The Windows based solutions both require IKE/ISAKMP, therefore interoperability with embedded IPsec is not possible. Please read section 8.4.1 on page 45 for a short description of the

tested IPsec implementations. To significantly improve the interoperability, dynamic configuration based on IKE/ISAKMP protocols would be required.

embedded IPsec configuration	FreeS/WAN V1.96	ipsec_tunnel v0.9	OpenBSD	Windows 2000	PGPnet v7.03
AH HMAC-MD5-96	✓	-	(✓)	n/a	n/a
AH HMAC-SHA1-96	✓	-	(✓)	n/a	n/a
ESP 3DES	✓	✓	(✓)	n/a	n/a
ESP 3DES HMAC-MD5	✓	✓	(✓)	n/a	n/a
ESP 3DES HMAC-SHA1	✓	✓	(✓)	n/a	n/a

Table 10 Interoperability with other IPsec implementations

✓supported and successfully verified
× not supported and verified
- Feature not available
(✓) supported but not verified
(×) not supported but not verified
n/a not testable (we don't support IKE)

9.2 MISSING FEATURES

As stated above, there were features that could not be implemented during the budgeted eight weeks. Some of them are stated as a "MUST" in the RFC. This does not mean that our implementation does not fully work. Some of the missing features can easily be separated, so that the nonexistent characteristics do not affect general IPsec processing.

Feature	RFC Reference	Comment
ISAKMP	Not essentially requested	ISAKMP would allow to dynamically generating SAs and improve interoperability.
Transport mode	RFC 2401, Section 4.1	Transport-mode could be used for peer-to-peer configurations
Single DES encryption	RFC 1827, Section 5	Single DES would increase the performance and interoperability
Reassembling fragmented IPsec	RFC 2401, Section 5.2	Fragmented IPsec packets must be reassembled

packets		before they are processed by the IPsec library

Table 11: List of missing features

9.3 POSSIBLE IMPROVEMENTS

Towards the end of our project, we also thought about what additional functionality would be required to have a marked ready product. A market ready product in our opinion signifies that the IPsec library is an all-purpose implementation, robust and easy to use for other developers. Portability to other hardware should be possible as well as integration into other TCP/IP stacks.

9.3.1 RE-ENTRANT CODE

Our lwIP environment, where the IPsec library was tested, is only a single threaded application running on top of the microcontroller. Therefore, the code did not need to be fully re-entrant. Nevertheless, the implementation should also run on other systems, like multi-threaded environments. The adjustments required for this feature would affect all functions using a global data structure. Accessing mutexes at the beginning and at the end of these functions and avoiding all remaining global variables would help making the code re-entrant.

9.3.2 ABILITY TO TURN OFF ANTI-REPLAY MECHANISM

During testing, we realized that the anti-replay mechanism might have a bad impact on IPsec processing if authentication is turned on. When a microcontroller's IPsec configuration forces anti-replay checks (this is the case whenever a HMAC is involved), the sequence number and the anti-replay window are continuously checked and updated. When one of the two communication peers is restarted, the SA must be re-established and the sequence number is reset to zero. The sequence number of the two parties gets out of synchronization and as soon as it exceeds the anti-replay window size, all further communication is rejected.

This problem is a reason for this feature to be used optionally [13]. If ISAKMP is not present on a system, then applying anti-replay check does not make sense because SA can't be re-established after a one-side shutdown.

[13] RFC 2401, section 3.2 and section 4.2

9.3.3 SUPPORT MORE CIPHERS

Implementing DES and 3DES according the RFCs may be sufficient for most PC applications because there is enough performance available to encrypt and decrypt packets within reasonable time. However, microcontrollers are much slower and therefore algorithms that are more effective are desirable.
Two interesting encryption algorithms would be nice to have. The first one is IDEA. IDEA is even faster than single DES and provides high security. Before AES appeared last year, IDEA was considered the most secure symmetric block cipher. The second one would be the new AES standard, which enjoys increasing popularity. Many products already support AES. Having AES in our crypto library could increase performance, security and interoperability with other IPsec systems.

9.3.4 IMPROVE CODE STRUCTURE

Due to our hardware restrictions, the IPsec implementation was only tested with one network interface. We planed for the code to support multiple interfaces but because of time restrictions, we did not spend too much time on such details.

Now, the static SPD/SAD configuration is done in the module of the IPsec device driver. This is not an optimal solution because once such a driver is written for a certain TCP/IP stack, it should not be necessary to always recompile this driver after a SPD/SAD reconfiguration. It would also be nice, if the user (the developer using our IPsec library) could do all the configuration aspects in the main file. This change would require a few setter functions for the SPD and SAD configuration where a certain configuration can be associated to a network interface.

Further changes would also be required to fully support multiple network interfaces. If no appropriate hardware is available, then this situation could be simulated using dummy device drivers.

The cryptographic functions, which were ported from a 32-bit system, should be replaced by a more optimized microcontroller specific version (see section 9.1.3 on page 55).

10 DEVELOPER'S MANUAL

This chapter discusses the integration of embedded IPsec with regard to the lwIP TCP/IP stack, Keil C166 tool chain and a phyCORE167-HS/E board.

10.1 INTRODUCTION

This manual shows the essential steps required to integrate embedded IPsec with lwIP TCP/IP stack.
The used stack, written by Adam Dunkels of the Swedish Institute of Computer Science *[DUNK]* is now being actively developed by an active word-wide community of developers.
Keil C166 tool chain is used to compile and debug the code on a Phytec phyCORE167-HS/E 16-bit microcontroller board.

Please note that the approach is analogue for other compilers and targets. This manual should give you an idea of how to get started. Basic knowledge of IPsec, lwIP and the used hard- and software is helpful to understand and reproduce the setup.

10.2 PREPARATION

The following requirements must be fulfilled in advance:

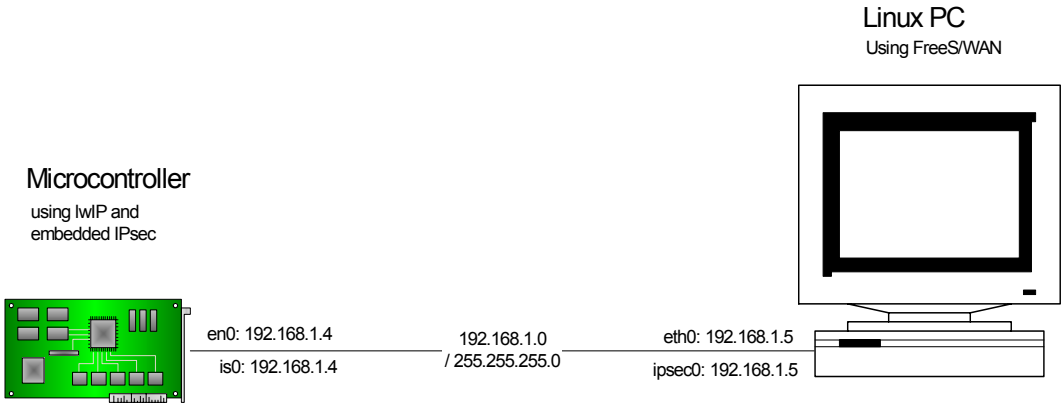


Figure 15: Network configuration used in this manual

PC with Keil C166 compiler and uVision2 IDE

A PIII 450MHz with Windows 2000 Professional, Keil C166 v4.27 and uVision2 v2.38a is sufficient.
More information here: http://www.keil.com

Linux system running FreeS/WAN

A PIII 450MHz with Debian GNU/Linux 3.0 (Woody) running Kernel 2.4.18 and FreeS/WAN 1.96 in combination with tcpdump proved to be suitable. Please note that the Linux system is only needed to test embedded IPsec.
More information here: http://www.debian.org
http://www.freeswan.org/

Phytec phyCORE167-HS/E rapid development kit

The phyCORE board is equipped with an Infineon SAK-C167CS-L40M microcontroller, a Crystal LAN CS8900 Ethernet controller, 1MB Flash-ROM and 1MB RAM. A monitor program is available that supports debugging on the target using the Keil uVision2 IDE. The Phytec Spectrum CD and FlashTools must be installed.
More information here: http://www.phytec.de

Latest stable version of the lwIP TCP/IP stack

The version used here is a copy of the lwIP Savannah CVS "STABLE" branch from October 20, 2003.
More information here: http://www.sics.se/~adam/lwip/

Latest stable version of embedded IPsec

More information here: http://www.hta-bi.bfh.ch/Projects/ipsec/

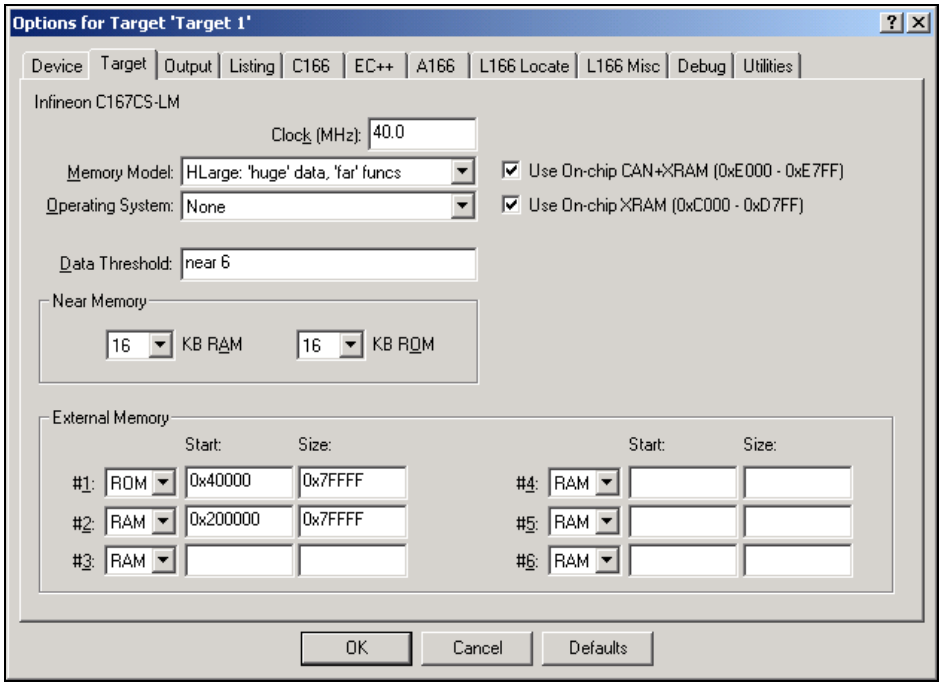
The proposed folder structure may look complicated, especially when adding relative include paths for the C compiler. As soon as the developer wants to update independently lwIP, embedded IPsec or the custom project from a CVS server, the strict separation of the three parts is particularly helpful. The folder structure of embedded IPsec is similar to lwIP:

Path	Description
.\ipsec\doc\	IPsec source code documentation
.\ipsec\src\core\	IPsec core functionality source code
.\ipsec\src\netif\	IPsec network interfaces
.\ipsec\src\include\	IPsec header files source code
.\ipsec\src\testing\	IPsec test framework and verification code
.\lwip\doc\	lwIP source code documentation
.\lwip\src\core\	lwIP core functionality source code
.\lwip\src\core\ipv4\	lwIP IPv4 specific source code
.\lwip\src\core\netif\	lwIP network interfaces
.\myproject\src\demo\	The custom application
.\myproject\src\ipsectest\	IPsec test framework port

Table 12: embedded IPsec path layout

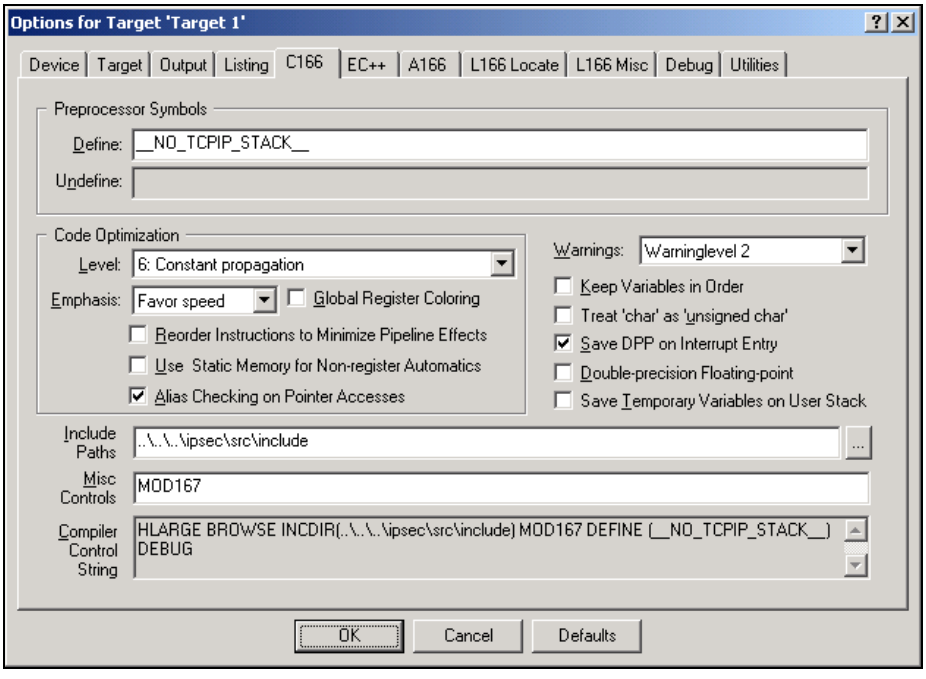
10.3 CREATING A NEW UVISION2 PROJECT

After starting the uVision2 IDE, a new project must be created and saved in the project folder (".\myproject\src\ipsectest\" for example). "C167CS-LM" must be selected as target CPU.
Depending on the version of uVision2, the IDE proposes to add C16x/ST10 startup code to the project. Since the phyCORE board has a slightly different configuration, we recommend using the startup file provided by Phytec. If the Phytec Spectrum CD and FlashTools are properly installed, this startup file should be copied from C:\PHYBasic\pC-167HSE\Demos\Keil\Debug\Start-pC167HSE_debug.a66 into the project folder and added to the 'Source Group 1' using the menu "Project", "Targets, Groups, Files".
The default project options must be slightly adjusted to specify valid memory areas. For development, the "HLarge" memory model gives the freedom to use the entire addressable memory.



Screenshot 1: Target options

The options for the C166 compiler define the include path of the embedded IPsec header files ("`..\..\..\ipsec\src\include;`" in this example). As long as no TCP/IP stack is present, the preprocessor symbol "`__NO_TCPIP_STACK__`" must be set.



Screenshot 2: Compiler options for IPsec

If the Keil target monitor is used, "Reserve" must be set to "8H-0BH, 0ACH-0AFH" in the "L166 Misc" option tab. All other settings do not require any particular adaptation.

10.4 VERIFICATION OF COMPILER AND EMBEDED IPSEC

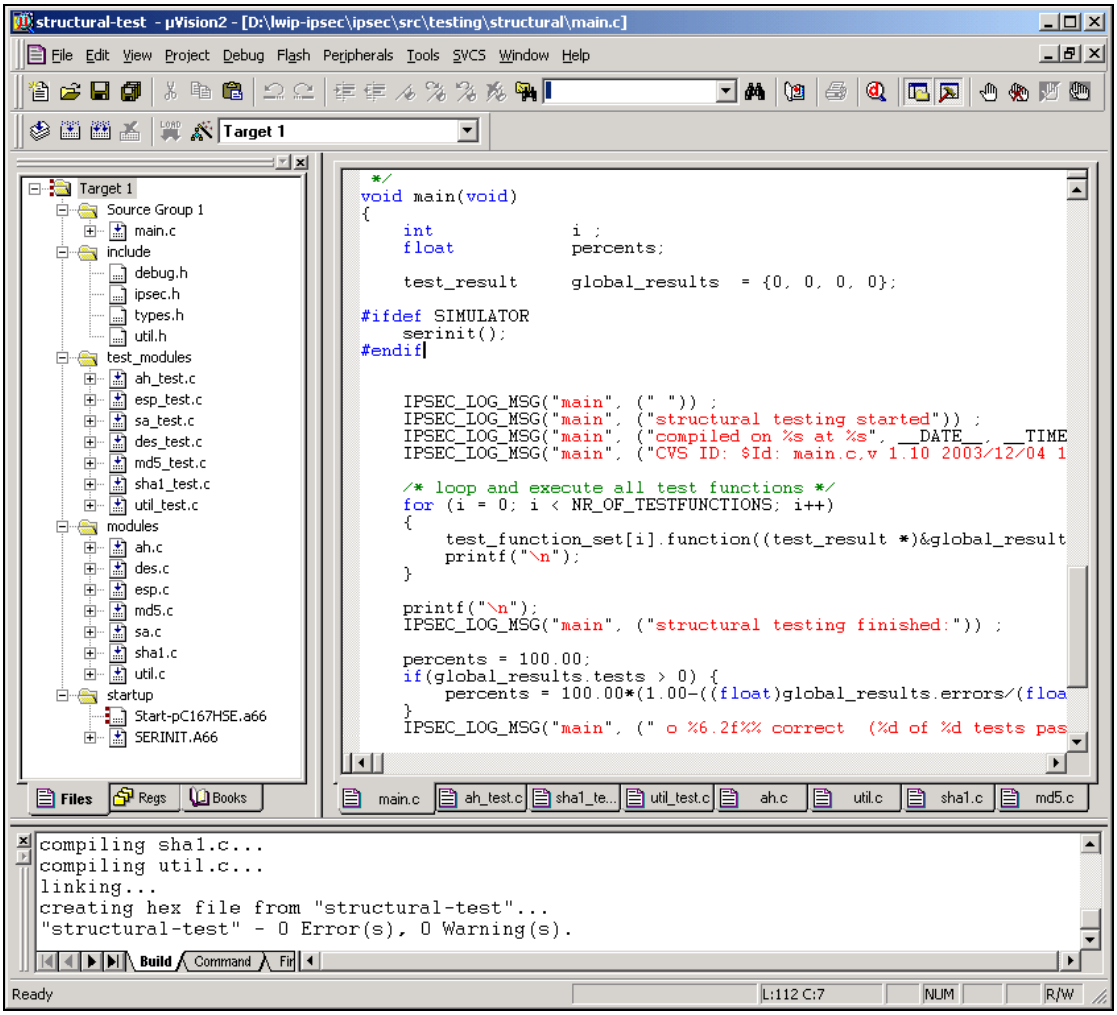
Before integrating the embedded IPsec code with the TCP/IP stack, it is strongly recommended to adjust the embedded IPsec type definitions in `types.h` and verify the correctness of the IPsec library by running the structural tests.

Name	Size	Value range	Keil C166 data type
__u8	8 bits	0 to 255	unsigned char
__s8	8 bits	-128 to +127	signed char
__u16	16 bits	0 to 65535	unsigned short
__s16	16 bits	-32768 to +32767	signed short
__u32	32 bits	0 to 4294967295	unsigned long
__s32	32 bits	-2147483648 to +2147483647	signed long

Table 13: embedded IPsec data types defined in `types.h`

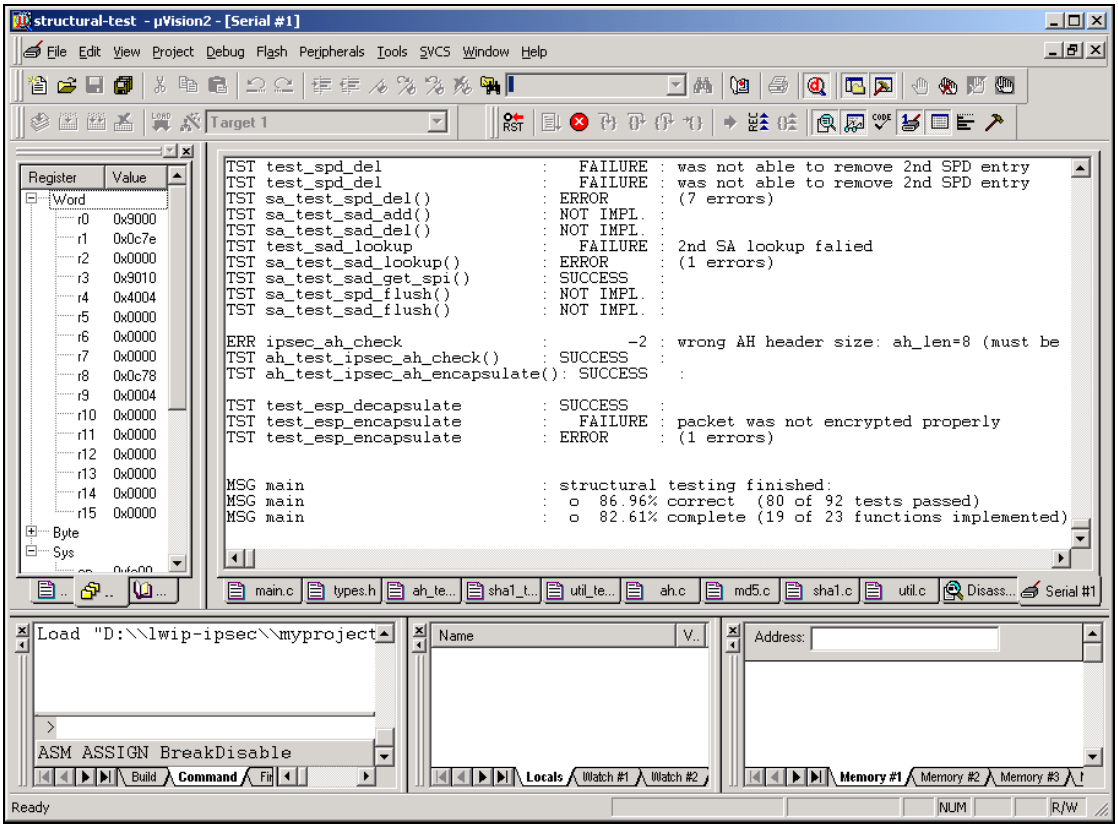
Care must be taken that the compiler and linker respect the byte alignment of all defined structures, especially when they represent network packet headers.

The verification of the embedded IPsec core functionality is performed as structural test without TCP/IP stack. Thus the preprocessor symbol "__NO_TCPIP_STACK__" must be set.
All the test framework files must be added to the uVision2 project and should compile flawlessly and without warnings. Instead of giving a complete list of all required files, Screenshot 3 shows all necessary files.



Screenshot 3: Test framework with all needed files in uVision2

Flawless compilation itself does not automatically imply flawless operation of the embedded IPsec core routines. Every single test must pass with a SUCCESS notice and the correctness of all tests must be 100.00%. As long as some tests fail, there might be something wrong with the compiler settings (memory alignment of structures, …), errors in the embedded IPsec source code or its test cases. After running all structural tests, the framework's result may look as in Screenshot 4. In this case, only 86.96% of the code could be verified. Wherever a FAILURE or NOT IMPL. (not implemented) message is displayed, the corresponding IPsec functions and its test cases must be manually verified.



Screenshot 4: Test framework results

10.5 EMBEDDED IPSEC – LWIP INTEGRATION

Prior to integration of embedded IPsec with lwIP, the CVS version of lwIP must be adapted for the Keil compiler and Phytec board. Leon Woestenberg *[LEO]*, an active developer of lwIP, has written a driver for the CS8900 Ethernet controller that can be used with our hard- and software with only minimal changes. It is recommended to add "`PHYCORE167HSE`" and "`LWIP_DEBUG`" to the pre-processor symbols and update the include path field to "`.\include; ..\..\..\lwip\src\include; ..\..\..\lwip\src\include\ipv4; ..\..\..\ipsec\src\include`" in the C166 option field.

10.5.1 CONFIGURING LWIP

The lwIP TCP/IP stack is configured using the "`lwipopts.h`" file. The most important options are described in Table 14. More in-depth information about these and all remaining options can be found directly in the lwIP source code or in the lwip-users mailing list archive *[lwUsr]*. The result will be an up-to-date lwIP port similar to *[CSKP]*.

#define…	Value	Meaning
MEM_ALIGNMENT	2	16-bit controller requires 2-byte alignment
MEM_SIZE	15000	15'000 bytes heap, used for pbuf memory
MEMP_NUM_PBUF	8	Means 8 concurrent pbuf ROM buffers are available. This value should be larger if most data is sent from ROM (i.g. static web pages).
PBUF_POOL_BUFSIZE	500	This defines the size of each pbuf pool payload. Some bytes are required for the pbuf structure and the Ethernet frame header. In this case, the effective IPSEC MTU will be around 420 bytes per packet. All pbuf pool buffers must fit within MEM_SIZE, so there is room to scope left to favor more, but smaller or less, but larger network packets.
PBUF_POOL_SIZE	24	Means 24 concurrent pbuf RAM pool packet buffers can be held in memory. It is desirable to have a lot of these buffers to be able to queue more incoming packets if they arrive at a short time interval. But since each pbuf requires approximately PBUF_POOL_BUFSIZE bytes of RAM, such a pbuf type is very costly.
MEMP_NUM_TCP_SEG	16	If TCP is used, the number of simultaneously "active" TCP segments must be watched. Depending on the application, it may make sense to poll the user application more frequently and transfer a lot of smaller TCP segments or have bigger segments or fewer, but larger ones.
TCP_SND_QUEUELEN	8	The TCP send queue can rapidly grow if lwIP does not receive the ACK packets. If the queue is full, the user application may fails. In this case, 8 TCP packets can be simultaneously "on the way".

Table 14 Important changes in the lwIP configuration file `lwipopts.h`

10.5.2 PATCHING LWIP PBUF CODE

As described in section 7.3 on page 26, our goal was to avoid memory copy operations wherever possible. With a simple patch in the lwIP's packet buffer structure, we were able to ensure that buffers were large enough to add IPsec headers if required.

Changes in file `.\lwip\src\core\pbuf.c`:

Add the following include file approximately at line 75 :

```
#include "netif/ipsecdev.h"   /* add room for ESP and AH header */
```

Add the following code approximately at line 273, inside the pbuf_alloc() function after the first switch() block. It will ensure that there is enough head room for AH, ESP and outer IP headers:

```
/* add room for IPsec layer header */
offset += IPSEC_HLEN;       /* add room for ESP and AH */
```

The approach for other TCP/IP stacks is similar. It must be ensured that there is always enough room for AH/ESP and outer IP headers and trailers for each packet. Please refer to AH processing (see section 7.5 page 30) and ESP processing (see section 7.6 page 32) to understand how AH and ESP functions expect the data to be laid out.

10.5.3 ADAPTING IPSECDEV FOR LWIP

To intercept the traffic between the physical device driver and lwIP, the customized ipsecdev driver is used. For inbound interception, it must emulate the lwIP's ip_input() functionality and for outbound interception, the functionality of lwIP's cs8900_output() interface must be provided.

lwIP only (simplified):

INBOUND:	cs8900_input() calls ip_input()
OUTBOUND:	ip_output_if() calls cs8900_output()

lwIP with embedded IPsec (simplified):

INBOUND:	cs8900_input() calls ipsecdev_input() calls ip_input()
OUTBOUND:	ip_output_if() calls ipsecdev_output() calls cs8900_output()

Please refer to the source code and the Doxygen documentation for a detailed description of the interfaces.

10.6 SAMPLE PROGRAM

A sample program implements the RFC 862 Echo Protocol. All UDP packets sent to port 7 are echoed back to the sender. IPsec specific parts of this program are exemplified while the complete program source code is available separately on the project homepage.

10.6.1 CONFIGURATION

Both ends of the IPsec tunnel connection - the microcontroller and the FreeS/WAN client- need to share a common configuration. An AH configuration with HMAC-MD5-96 was chosen for this manual. Since AH does not encrypt the packet the sent data can easily be explored with a sniffer. The following minimal configuration was used on the Linux PC and stored in the file "`/etc/ipsec.conf`":

```
# /etc/ipsec.conf - FreeS/WAN IPsec configuration file
# basic configuration
config setup
        interfaces="ipsec0=eth1"
        klipsdebug=none
        plutodebug=none
        plutoload=%search
        plutostart=%search
        uniqueids=yes

# defaults for subsequent connection descriptions
# (mostly to fix internal defaults which, in retrospect, were badly chosen)
conn %default
        keyingtries=0
        disablearrivalcheck=no

# AH tunnel between MCU (192.168.1.4) and Linux machine "scuol" (192.168.1.5)
# with manual keying.
conn manual14
        authby=manual
        left=192.168.1.4
        leftid=@microcontroller
        right=192.168.1.5
        rightid=@scuol
        ah=hmac-md5-96
        spi=0x1014
        ahkey=0x01234567_01234567_01234567_01234567
```

The configuration of the Linux machine must be verified and should look similar to the following output:

```
scuol:~# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:A0:24:15:3E:12
          inet addr:192.168.1.5  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:459850 errors:0 dropped:0 overruns:0 frame:0
          TX packets:469409 errors:0 dropped:0 overruns:0 carrier:0
          collisions:11 txqueuelen:100
          RX bytes:69076140 (65.8 MiB)  TX bytes:51238384 (48.8 MiB)
          Interrupt:5 Base address:0x220

scuol:~# ifconfig ipsec0
ipsec0    Link encap:Ethernet  HWaddr 00:A0:24:15:3E:12
          inet addr:192.168.1.5  Mask:255.255.255.0
          UP RUNNING NOARP  MTU:16260  Metric:1
          RX packets:446207 errors:0 dropped:27 overruns:0 frame:0
          TX packets:449387 errors:0 dropped:18 overruns:0 carrier:0
          collisions:0 txqueuelen:10
```

```
          RX bytes:42665804 (40.6 MiB)  TX bytes:49098713 (46.8 MiB)

scuol:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.4     192.168.1.4     255.255.255.255 UGH   0      0        0 ipsec0
192.168.1.0     *               255.255.255.0   U     0      0        0 eth1
192.168.1.0     *               255.255.255.0   U     0      0        0 ipsec0
scuol:~# ipsec eroute
16         192.168.1.5/32     -> 192.168.1.4/32     => tun0x1014@192.168.1.4
scuol:~#
```

On the microcontroller, data structures with the initial SAD and SPD entries for inbound and outbound processing must be declared.
For inbound connections (coming from the Linux machine to the microcontroller), the databases look like this:

```
/* INBOUND SAD configuartion data */
sad_entry inbound_sad_config[IPSEC_MAX_SAD_ENTRIES] = {
    SAD_ENTRY( 192,168,1,4,                 /* destination address          */
               255,255,255,255,             /* network mask                 */
               0x1014,                      /* SPI                          */
               IPSEC_PROTO_AH,              /* protocol (AH or ESP)         */
               IPSEC_TUNNEL,                /* tunnel mode                  */
               0,                           /* cipher (0 == n/a) and key    */
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               IPSEC_HMAC_MD5,              /* authentication algorithm & key */
               0x01, 0x23, 0x45, 0x67, 0x01, 0x23, 0x45, 0x67, 0x01, 0x23, 0x45, 0x67, 0x01, 0x23, 0x45, 0x67, 0, 0, 0, 0 ),
    EMPTY_SAD_ENTRY
} ;

/* INBOUND SPD configuration data */
spd_entry inbound_spd_config[IPSEC_MAX_SAD_ENTRIES] = {
    SPD_ENTRY( 192,168,1,5,                 /* source address               */
               255,255,255,255,             /* source network               */
               192,168,1,4,                 /* destination address          */
               255,255,255,255,             /* destination network          */
               0,                           /* protocol (0 == ANY)          */
               0,                           /* source port (0 == ANY)       */
               0,                           /* destination port (0 == ANY)  */
               POLICY_APPLY,                /* policy                       */
               &inbound_sad_config[0] ),    /* SA pointer                   */
    EMPTY_SPD_ENTRY
};
```

For our outbound connections, thus sending data from the microcontroller towards the Linux machine, the corresponding database configuration is:

```
/* OUTBOUND SAD configuartion data */
sad_entry outbound_sad_config[IPSEC_MAX_SAD_ENTRIES] = {
    SAD_ENTRY( 192,168,1,5,                 /* destination address          */
               255,255,255,255,             /* network mask                 */
               0x1014,                      /* SPI                          */
               IPSEC_PROTO_AH,              /* protocol (AH or ESP)         */
               IPSEC_TUNNEL,                /* tunnel mode                  */
               0,                           /* cipher (0 == n/a) and key    */
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               IPSEC_HMAC_MD5,              /* authentication algorithm & key */
               0x01, 0x23, 0x45, 0x67, 0x01, 0x23, 0x45, 0x67, 0x01, 0x23, 0x45, 0x67, 0x01, 0x23, 0x45, 0x67, 0, 0, 0, 0 ),
    EMPTY_SAD_ENTRY
};

/* OUTBOUND SPD configuration data */
spd_entry outbound_spd_config[IPSEC_MAX_SPD_ENTRIES] = {
    SPD_ENTRY( 192,168,1,4,                 /* source address               */
               255,255,255,255,             /* source network               */
               192,168,1,5,                 /* destination address          */
               255,255,255,255,             /* destination network          */
               0,                           /* protocol (0 == ANY)          */
               0,                           /* source port (0 == ANY)       */
               0,                           /* destination port (0 == ANY)  */
               POLICY_APPLY,                /* policy                       */
               &outbound_sad_config[0] ),   /* SA pointer                   */
    EMPTY_SPD_ENTRY
};
```

First, the lwIP TCP/IP stack must be initialized by calling the various initialization functions provided by lwIP:

```
    /* initialize lwIP */
    etharp_init();  mem_init();   memp_init();  pbuf_init();
    netif_init();   ip_init();    udp_init();   tcp_init();
    printf("TCP/IP initialized.\n");
```

Followed by the setup of the IP addresses. The external Ethernet device and the IPsec device share a common configuration:

```
    /* configure Ethernet device */
    IP4_ADDR(&eth_ipaddr, 192,168,1,4);
    IP4_ADDR(&eth_netmask, 255,255,255,0);
    IP4_ADDR(&eth_gw, 192,168,1,1);

    /* configure IPsec device */
    IP4_ADDR(&ipsec_ipaddr, 192,168,1,4);
    IP4_ADDR(&ipsec_netmask, 255,255,255,0);
    IP4_ADDR(&ipsec_gw, 192,168,1,1);
```

The order of configuring and initializing the network interfaces is crucial. lwIP's netif_add() function initializes the data structures of the network, sets the input function (ip_input() or ipsec_input()) and calls the initialization routines of the device drivers (cs8900if_init() or ipsecdev_init()).

```
    /* Initialize the physical device first (so that ethif->next will point  *
     * to ipsecif). The IPsec device will process the packets and pass them  *
     * upper to the IP layer (using ip_input)                                */
    printf("Setting up et0...");
    ethif = netif_add(&eth_ipaddr, &eth_netmask, &eth_gw, NULL,  cs8900if_init, ipsecdev_input);
    printf("OK\n");

    printf("Setting up is0...");
    ipsecif = netif_add(&ipsec_ipaddr, &ipsec_netmask, &ipsec_gw, NULL,  ipsecdev_init, ip_input);
    printf("OK\n");
```

Before data can be sent or received over the IPsec device, the tunnel must be configured using the ipsec_set_tunnel() function:

```
/* configure IPsec tunnel */
ipsec_set_tunnel("192.168.1.4", "192.168.1.5");
```

Setting the IPsec device as default interface ensures that lwIP applications send and receive data only over IPsec:

```
/* set the IPsec interface "is0" as default interface for lwIP */
netif_set_default(ipsecif);
```

The main loop looks the same as a usual lwIP application with periodically polling of the CS8900 device driver. When data must be sent or received, the IPsec processing and transmission is automatically triggered:

```
while( 1 )
    /* blink status LED */
    P2 = ~P2;
    for( i = 0; i < 50000; i++ )
        /* delay or use this time for the application */
        _nop_(); _nop_();
        _nop_(); _nop_();
        _nop_(); _nop_();
        if ( (i % 250) == 0 )
        then                                   else
        /* periodically call to the CS8900 driver (polling mode) */   /* continue */
        cs8900if_service(ethif);
```

10.6.2 TEST

Using the configuration outlined in section 10.2 on page 61, we were able to create a demo application and test it with Linux as peer partner. For this test, the assumption was made that FreeS/WAN had been properly installed and the configuration from section 10.2 was stored under "`/etc/ipsec.conf`".
On the Linux machine, the following commands are needed to configure FreeS/WAN:

```
scuol:~# ipsec setup start
ipsec_setup: Starting FreeS/WAN IPsec 1.96...
scuol:~# ipsec manual --up manual14
scuol:~#
```

On the microcontroller board, the compiled application must be started (either programming the application into FLASH memory or running it from uVision2 using the Keil target monitor). The following start-up message is printed on the RS232 port (9600baud, 8 data bits, no parity, 1 stop bit, no flow control) appears:

```
lwIP - embedded IPsec integration demo(Dec 08 2003 at 11:45:46)

TCP/IP initialized.
Setting up et0...OK
Setting up is0...OK
Applications started.
```

If the "OK" string is not displayed after having called the interface et0 initialization routine, the CS8900 device could not be initialized. This can happen if the Ethernet controller is broken, or more likely, the base addresses are not properly set in the drivers source code.

As soon as both microcontroller and Linux machine have initialized their IPsec devices, the connection can be tested with sending a sequence of ping packets from the Linux machine to the microcontroller:

```
scuol:~# ping -c 4 192.168.1.4
PING 192.168.1.4 (192.168.1.4): 56 data bytes
64 bytes from 192.168.1.4: icmp_seq=0 ttl=64 time=72.5 ms
64 bytes from 192.168.1.4: icmp_seq=1 ttl=64 time=70.9 ms
64 bytes from 192.168.1.4: icmp_seq=2 ttl=64 time=71.0 ms
64 bytes from 192.168.1.4: icmp_seq=3 ttl=64 time=71.1 ms

--- 192.168.1.4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 70.9/71.3/72.5 ms
scuol:~#
```

Considerable time is needed to print logging messages on the microcontrollers serial port, thus disabling any logging will noticeably reduce response time. The microcontroller should properly receive and process the ping requests and return an appropriate answer to the sender. The following messages are logged:

```
MSG ipsec_input        : fwd decapsulated IPsec packet to ip_input()
AUD ipsecdev_output    :   3 : POLICY_APPLY: processing IPsec packet
MSG ipsec_output       : have to encapsulate an AH packet
MSG ipsec_output       : fwd IPsec packet to HW mapped device
MSG ipsec_input        : fwd decapsulated IPsec packet to ip_input()
AUD ipsecdev_output    :   3 : POLICY_APPLY: processing IPsec packet
MSG ipsec_output       : have to encapsulate an AH packet
MSG ipsec_output       : fwd IPsec packet to HW mapped device
MSG ipsec_input        : fwd decapsulated IPsec packet to ip_input()
AUD ipsecdev_output    :   3 : POLICY_APPLY: processing IPsec packet
MSG ipsec_output       : have to encapsulate an AH packet
MSG ipsec_output       : fwd IPsec packet to HW mapped device
MSG ipsec_input        : fwd decapsulated IPsec packet to ip_input()
AUD ipsecdev_output    :   3 : POLICY_APPLY: processing IPsec packet
MSG ipsec_output       : have to encapsulate an AH packet
MSG ipsec_output       : fwd IPsec packet to HW mapped device
```

As additional test, the UDP Echo Protocol is implemented in the test application. The microcontroller can be connected on port 7 using the Linux "nc" utility (works as "UDP client" when started with parameter –u). All entered data will be sent to the microcontroller and echoed back:

```
scuol:~# nc -u 192.168.1.4 7
Hello...
```

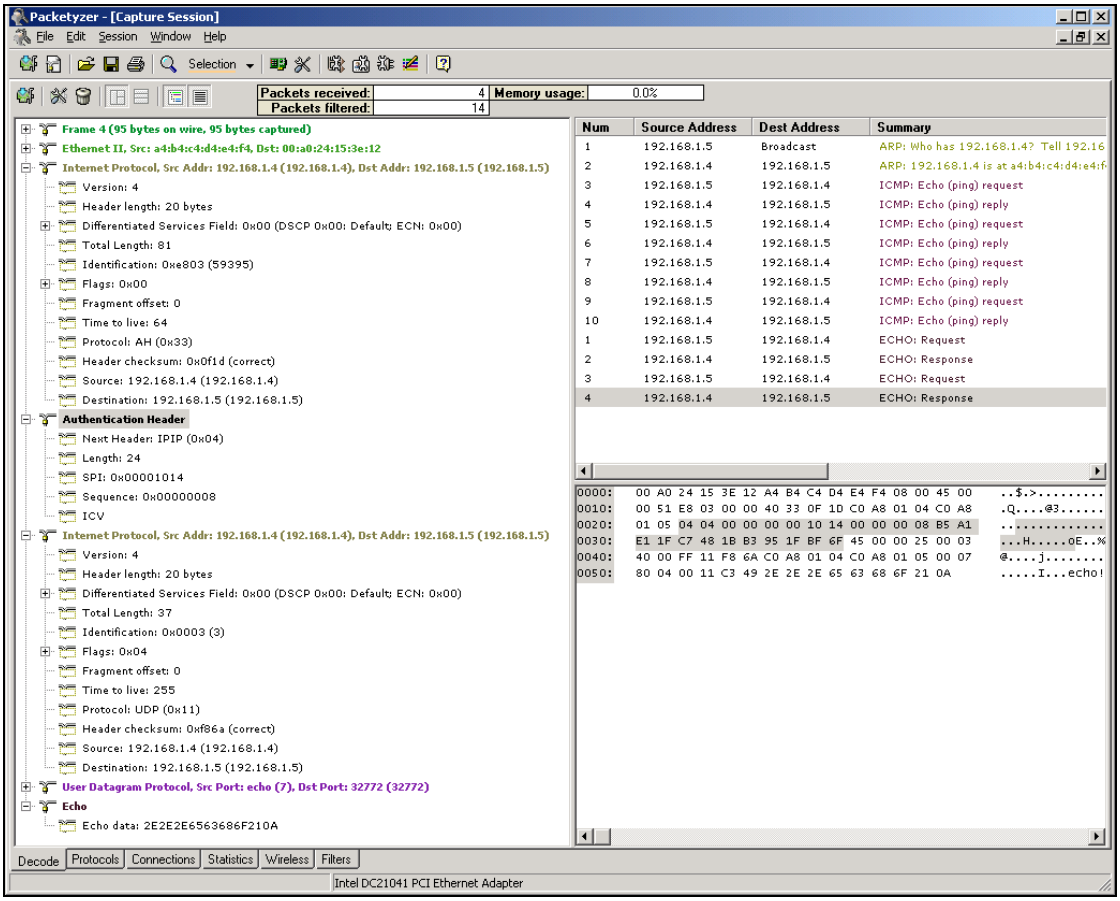
```
Hello...
...echo!
...echo!

scuol:~#
```

Log messages from the application show up between the IPsec messages:

```
MSG ipsec_input        : fwd decapsulated IPsec packet to ip_input()
UDP port 7 received and echo: Hello...
AUD ipsecdev_output    :   3 : POLICY_APPLY: processing IPsec packet
MSG ipsec_output       : have to encapsulate an AH packet
MSG ipsec_output       : fwd IPsec packet to HW mapped device
MSG ipsec_input        : fwd decapsulated IPsec packet to ip_input()
UDP port 7 received and echo: ...echo!
AUD ipsecdev_output    :   3 : POLICY_APPLY: processing IPsec packet
MSG ipsec_output       : have to encapsulate an AH packet
MSG ipsec_output       : fwd IPsec packet to HW mapped device
```

The whole test can be observed using a sniffer such as Packetyzer (read section 6.2 on page 2 for more details). Screenshot 5 shows all packets of the scenario described above.



Screenshot 5: Sniffed AH session

10.7 DEBUGGING

Debugging of the embedded IPsec library is very specific to the used environment. We implemented a collection of common debug macros that produce different kinds of logging messages. These are helpful for detecting configuration errors, inspecting the transferred traffic before and after IPsec processing and for tracing the program execution by printing the used function arguments. Table 15 lists the available macros and gives a short description.

#define...	Functionality
IPSEC_ERROR	If defined, severe configuration errors and not manageable states such as running out of memory are logged. It is recommended to have this feature enabled by default.
IPSEC_DEBUG	If less critical errors should also be logged, this feature must be enabled. The produced additional output can be a supplement to the error messages logged under IPSEC_ERROR.
IPSEC_MESSAGE	This feature controls informative messages. They are particularly helpful to have a

	"lightweight trace" of the program execution.
IPSEC_TRACE	If in-depth information with details of all passed and returned parameters is needed, IPSEC_TRACE must be defined. Since this feature produces a vast amount of information, it is recommended to disable it by default.
IPSEC_AUDIT	If defined, auditable events according to the IPsec RFC's are logged.
IPSEC_TEST	This feature is used only inside the test routines and prints log messages in an uniform style.
IPSEC_DUMP_BUFFERS	Printing a HEX-dump of large memory buffers can be very time consuming. Only if defined, dumping of buffers is enabled.
IPSEC_TABLES	Some information is printed in tables. To avoid this time consuming operation, this feature must be disabled.

Table 15 Debug and trace options defined in `util.h`

11 GLOSSARY

AH: Authentication Header

AH is an IPsec protocol header, which provides integrity, authentication and anti-replay protection. AH is documented in RFC 2402.

ESP: Encapsulating Security Payload

ESP is an IPsec protocol header, which provides confidentiality, authentication, integrity and anti-replay protection. ESP is documented in RFC 2406.

IPsec: Internet Protocol Security

IPsec is used to refer to security protecting the Internet Protocol. The standards (RFC's), which define IPsec are maintained by IETF and are available under http://www.ietf.org/html.charters/ipsec-charter.html. The main document is RFC 2401.

SA: Security Association

An SA is used to process IPsec packets. It describes how certain packets must be processed and it holds data about the current connection. Each current IPsec connection must have an SA in order to get the right cryptographic algorithms and keys. Sequence numbers, timeouts and SPI are also important fields of an SA. The SA is described in RFC 2401.

SAD: Security Association Database

The SAD is the database holding and maintaining multiple SA entries. It also provides functionality to search and access the entries. The SAD is described in RFC 2401.

SP: Security Policy

The SP is an entry describing which packet the policy must be applied to. The entries policy may be accept, bypass or discard. The SP is described in RFC 2401.

SPD: Security Policy Database

The SPD is the database holding and maintaining multiple SP entries. It also provides functionality to search and access the entries. The SPD is described in RFC 2401.

SPI: Security Parameters Index

The SPI is a 32-bit value used in the IPsec headers an in the SAs. It is used in the SA lookup, where it helps to map the SA to the appropriate IPsec packet, or vice versa.

Diffie-Hellman:

A cryptographic algorithm, which allows the exchange of secret keys.

ICV: Integrity Check Value

The ICV is calculated when IPsec header processing with authentication has been completed. The ICV is a checksum, which can only be calculated and verified with the appropriate secret key. When an ICV of a message is verified, it is guaranteed that the message was not altered by someone and that the message belongs to the expected recipient.

IV: Initialization Vector

The IV is used to initialize a CBC mode encryption or decryption.

DES: Data Encryption Standard

A widely used encryption algorithm.

3DES: Triple DES

An improved form of DES, when encryption/decryption is done three times instead of just once. 3DES is much more secure that DES but also three times less performing than DES.

MD5: Message Digest 5

A hash algorithm that calculates a checksum of a message. This checksum can be used to verify that a message has not been altered.

SHA1: Secure Hash Algorithm 1

A hash algorithm that calculates a checksum of a message. This checksum can be used to verify that a message has not been altered.

HMAC: Hash Message Authentication Code

A mechanism for message authentication that uses cryptographic hash functions. HMAC can be used with any iterative cryptographic hash function in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function.

RFC: Request For Comment

A huge collection of freely available Internet Standards.

IETF: International Engineering Task Force

The organization that maintains the IPsec standard.
http://www.ietf.org/html.charters/ipsec-charter.html

IKE: Internet Key Exchange

The key management protocol used in IPsec, IKE combines the ISAKMP and Oakley protocols to create encryption keys and security associations

ISAKMP: Internet Security Association and Key Management Protocol

A protocol that allows the receiver of a message to obtain a public key and use digital certificates to authenticate the sender's identity. ISAKMP is designed to be key exchange independent; that is, it supports many different key exchanges.

Oakley:

Key determination protocol based on the Diffie-Hellman algorithm that provides added security, including authentication. Oakley can be used as key-exchange algorithm in ISAKMP.

CBC: Cipher Block Chaining

Operation mode block ciphers.

TLS: Transport Layer Security

Same as Secure Socket Layer. Encryption technology for the Web used to provide secure transactions, such as the transmission of credit card numbers for e-commerce.

MCU: Microcontroller Unit

An abbreviation for microcontrollers. Also called mC or uC.

ALU: Arithmetical Logical Unit

Device logic that is responsible for the mathematical (add, subtract, ...), logical (and, or, ...), and shifting operation inside a microcontroller or processor.

lwIP: light weight IP

lwIP is a light weight TCP/IP implementation for microcontrollers originally written by Adam Dunkels of the Swedish Institute of Computer Science *[DUNK]* and is now being actively developed by an active word-wide community of developers.

pbuf: Packet Buffer

Pbufs are data structures that help maintaining data belonging to network packets. Pbufs give the possibility of dynamically changing and inserting data.

replay-attack:

Re-sending preliminary captured packets. An attacker observes an authentication process and can later try to get access by re-sending the same packets without having to know its contents.

12 APPENDIX

12.1 LIST OF TABLES

12.2 LIST OF FIGURES

12.3 LIST OF SCREENSHOTS

13 REFERENCES

[BEAT]

Sean Beatty; Sensible Software Testing, Embedded Systems Programming at embedded.com

http://www.embedded.com/2000/0008/0008feat3.htm

[CSNS03]

Implementing IPsec on a 16-bit microcontroller

http://www.hta-bi.bfh.ch/Projects/ipsec/

[LOSH02]

Peter Loshin, Big Book of IPsec RFCs, Morgan Kaufmann 2000

[RFC]

IETF Internet Engineering Task Force

http://www.ietf.org/html.charters/ipsec-charter.html

[STEF02]

Secure Communications in Distributed Embedded Systems

http://security.zhwin.ch/MSy02_Steffen.pdf

[DUNK]

A lightweight TCP/IP stack

http://www.sics.se/~adam/lwip/

[lwUsr]

lwip-users mailing list archives

http://mail.gnu.org/archive/html/lwip-users/

[DBDES]

Dmitry Basko's page with optimized DES implementations

http://www.dbasko.com/

[TRS]
An IPsec tunnel implementation for Linux 2.4
http://ringstrom.mine.nu/ipsec_tunnel/

[ZILE]
Mini Web Server supporting SSL
http://www.strongsec.com/zhw

[CSKP]
lwIP port for Keil C166 and MCB167-NET
http://www.christianscheurer.ch/projects/lwip/

[NCP]
Network Chemistry Packetyzer - Packet Analyzer for Windows
http://www.packetyzer.com/

[KEIL]
Keil C166 compiler with uVision2 IDE, MCB167NET board
http://www.keil.com

[PHY]
Phytec PhyCORE167-HS/E board
http://www.phytec.de

[CS89]
Cirrus Crystal LAN 10Base-T Embedded Ethernet Controller
http://www.cirrus.com/en/products/pro/detail/P46.html

[FS]
FreeS/WAN, IPsec implementation for Linux
http://www.freeswan.org/

[PGPN]
PGPnet, IPsec implementation for Windows and Macintosh
http://www.pgpi.org/

[DEB]

Debian Linux

http://www.debian.org/

[INF]

Infineon, manufacturer of C167 microcontrollers

http://www.infineon.com/

[HTX]

Background information about the C166 family

http://www.hitex.co.uk/c166/highinteg.html

[LEO]

CS8900 driver and other lwIP contributions by Leon Woestenberg

http://www.esrac.ele.tue.nl/~leon/lwip/