



Master's Thesis in
Computer Science
Examensarbete 2G1015
10th October 2000

Secure Distributed Communication in SAFEERLANG

by

Bertil Karlsson

Department of Teleinformatics
The Royal Institute of Technology
KTH

Supervisor: Dan Sahlin, Computer Science Laboratory, Ericsson UAB
Supervisor: Per Brand, Department of Teleinformatics, KTH
Examiner: Seif Haridi, Department of Teleinformatics, KTH

Abstract

SAFEERLANG is a project that aims to make the programming language ERLANG more safe in open distributed environments. This involves limitations on resource usage, access control on objects, distributed code loading and secure communication. This thesis is a step towards a true implementation of this concept, and addresses secure communication between ERLANG nodes.

Means to establish secure communication is presented as well as problems one has to defeat when secure communicating in an open media is the goal. Although this thesis involves a specific programming language the way this task is solved is not language dependent, and therefore of interest even in a broader sense.

A well-known protocol and open-source implementation is modified and integrated in the distribution layer of the ERLANG run-time system.

Contents

1	Introduction	9
1.1	General background	9
1.2	Acknowledgment and personal comment	9
2	Security requirements in communication	11
2.1	Encryption techniques	11
2.1.1	Symmetric encryption	12
2.1.2	Asymmetric encryption	12
2.2	Key distribution	14
2.2.1	Symmetric key systems	14
2.2.2	Asymmetric key systems	14
2.3	Hash functions	15
2.4	Security goals to reach	16
2.5	Threats to communication	17
2.5.1	Key length and quality	17
2.5.2	Types of threats	17
3	ERLANG	21
3.1	ERLANG properties	21
3.1.1	ERLANG basics	21
3.1.2	Processes, ports and concurrency	22
3.1.3	Distribution	23
3.1.4	Real-time properties	23
3.2	Security deficiencies in open distributed environments	24
3.2.1	Resource management	24
3.2.2	Access rights	24
3.2.3	Code loading	24
3.2.4	Insecure communication	25
4	SAFEERLANG	27
4.1	Resource management	27
4.2	Access rights	28
4.3	Code loading	28
4.4	Key distribution	29
4.5	Motives for a safer communication between ERLANG nodes.	30
4.5.1	No encryption of data.	30
4.5.2	Limited security with cookies	30
4.6	Summary of SAFEERLANG	31

5	Design of secure communication in SAFEERLANG	33
5.1	Secure protocols	33
5.1.1	General properties	33
5.1.2	Authentication and secret-sharing	34
5.1.3	Time-stamps and nonces	34
5.2	Establishment of encrypted channels - the handshake protocol	35
5.2.1	Key distribution	35
5.2.2	Security goals with the handshake protocol	35
5.2.3	SAFEERLANG handshake protocol - first version	36
5.2.4	SAFEERLANG handshake protocol - second version	38
5.2.5	Different attacks	38
5.3	The Session protocol	41
5.3.1	Security goals with the session protocol.	41
5.3.2	Why using a known protocol and implementation.	41
5.3.3	Data flow in SSL record protocol	43
5.4	Easy-to-use system	44
6	Implementation comments	45
6.1	Establishment of TCP/IP channels	45
6.2	The protocol implementation and something about SSLeay	46
6.2.1	SSLeay	46
6.2.2	The implementation	47
6.3	ERLANG communication	47
6.3.1	The Internet driver	48
6.3.2	ERLANG processes	48
6.4	Integration of implementation in ERLANG and SSLeay	48
6.5	Performance	50
7	Further Work	55
7.1	Improvements of the implementation	55
7.1.1	The protocol	55
7.1.2	Integration of secure channels in SAFEERLANG	55
7.2	Other future tasks	55
7.2.1	Several simultaneous connections	55
7.2.2	A key - node cache	56
8	Conclusion	59
	References	61
A	The SSL version 3.0 handshake protocol	63
B	Test programs with synchronous send/receive	65
	Index	69

List of Figures

1	Use of encryption when sending data from source to destination. . . .	11
2	Symmetric or Private-Key encryption. The same key is used in encryption and in decryption.	12
3	Asymmetric or Public-Key encryption. Different keys are used in encryption and decryption.	13
4	Threats when A sends message to B. C is an intruder that in c) illicitly listens to the traffic between A and B, in d) illicitly changes a message and in e) fabricates a message that seems to come from A. This figure is copied from [19].	18
5	MIM attack by C.	19
6	D has received a capability created by A. Now D accesses the resource on A.	29
7	First version of SAFEERLANG handshake protocol.	37
8	A MIM attack by C when A tries to connect to B.	39
9	This picture shows how the protocol handles a replay attack from entity C.	40
10	Cipher-block chain.	43
11	The data flow in SSL record protocol in cipher block chaining mode. .	44
12	Establishment of a connection between nodes A and B.	45
13	Protocol between ERLANG code and the Internet driver in the nodes involved in the handshake.	50
14	The broken line shows the performance (time/message) of the original ERLANG system and the other line represents the SAFEERLANG system with encryption.	51
15	The gap between the curves shows the overhead from the implementation.	52

10th October 2000

1 Introduction

1.1 General background

Computer programs that utilize network communication is part of each man's week-day, and new areas where communication is involved arise very often. A network like the Internet is an open network that is vulnerable to attacks, which can be caused by deliberate attempts to cause damage or unaware mistakes made of harmless users. Whether the attack is deliberate or not, the damage can be great.

Sometimes one has the need to protect the information from the view of others. There is a lot of considerations to do about possible threats, the wanted level of security, means to achieve this, etc. It is very easy to overlook some subtle detail, when considering security. One should notice that the reached security level in a system is as strong as the weakest point.

This thesis aims to introduce secure communication between ERLANG *nodes*, see section 3.1.3. The idea is to have a system where secure communication in a network is very easy for the programmer to manage. The only thing he has to consider is how the keys are distributed among the nodes in the system. This report involves a programming language, ERLANG, but the solution of the secure communication problem for this system is not language dependent. Thus it could be applied on other network layers as well.

In the following section I give a general presentation of security requirements in communication, which is required to understand the discussion in section 5. The programming language ERLANG and the SAFEERLANG project, which aims is to make ERLANG more safe in distributed environments, are presented in section 3 and 4. Then is the design of the secure communication in SAFEERLANG presented in section 5. In section 6, the implementation of the features presented in section 5, is commented.

1.2 Acknowledgment and personal comment

During this thesis, which has been carried out at the Computer Science Laboratory (CSLAB) at Ericsson UAB, I have been working close to my classmate Rickard Green who has done his master thesis simultaneously with me within the SAFEERLANG project. All discussions with him have given a lot of inputs for this thesis. My supervisor at Ericsson UAB, Dan Sahlin, has also been a great support through discussions, ideas and a very positive attitude. I also feel grateful for all help from other people at the Computer Science Laboratory and the OTP department at Ericsson UAB.

My thesis has also been the final moment of my studies at KTH. There have been some rewarding discussions with my supervisor at KTH, Per Brand.

This project has been an important experience for me. It is the first occasion when I have studied and done changes in large computer systems. Sometimes there have been moments when I have felt that the task have been too difficult to solve, but there have also been a lot of fun during this months of intensive work.

10th October 2000

Author of this paper is Bertil Karlsson. It is the most visible part of my master's thesis at the Department of Teleinformatics, KTH, Stockholm, carried out at the Computer Science Laboratory at Ericsson UAB.

The author can be reached by e-mail
at KTH: d95-bka@d.kth.se
or at work: Bertil.Karlsson@uab.ericsson.se

10th October 2000

2 Security requirements in communication

This section does not cover the topic in its uttermost extent, but only the necessary definitions and discussions to make the rest of the paper understandable.

2.1 Encryption techniques

When we have a message to communicate it is possible that it will fall into the wrong hands. If we use encryption to hide the information in the message we can send it without revealing the content to unauthorized entities. The original *plain-text* is transformed by the encryption to *cipher-text*, sent on the communication channel impossible for unauthorized entities to read, and converted to plain-text again by the decryption, as in figure 1. The notation used in this paper for encryption matters in general are the following:

- P plain-text
- C cipher-text
- E encryption algorithm, i.e. transformation from plain-text to cipher-text, like $C = E(P)$ or $C = E(P, K)$ if a key, K , is used.
- D decryption algorithm, i.e. transformation from cipher-text to plain-text, so that $P = D(C)$ or $P = D(C, K)$ if a key is used.
- K_{A-B} key used by entity A and entity B to perform encryption on a connection between those entities.

The third and fourth points together give that $P = D(E(P))$ or $P = D(E(P, K), K)$ if a key is used. If a key is used by the encryption algorithm the cipher-text depends on both the plain-text and the key together.

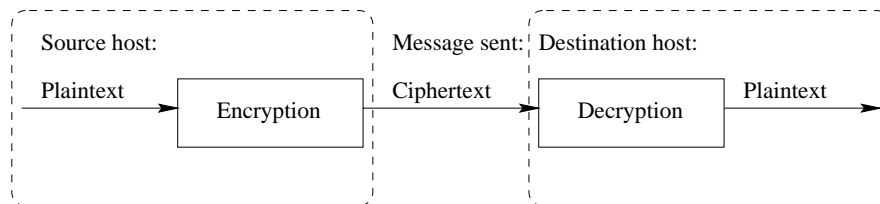


Figure 1: Use of encryption when sending data from source to destination.

In the rest of the report it is referred to both *symmetric* and *asymmetric* encryption, and the techniques are shortly explained in the next subsections.

2.1.1 Symmetric encryption

If the same key is used for both encryption and decryption the algorithm is called a symmetric or a *private-key* algorithm, see figure 2. In this case two communicating entities A and B have to share the same key, which has to be kept secret. If any other entity C knows that particular key it can interact in the secret communication with A and B. The most common case is when only two parties keep a communication secret, but in some systems one can interact with a group of computers using the same symmetric key. In [10] a system, Enclaves, for secure interaction by participating entities over the Internet is discussed. The participating entities shares a secret key by which they encrypt the data sent to the other entities in the group.

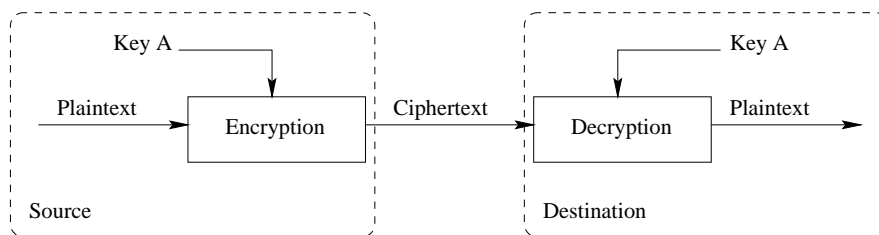


Figure 2: Symmetric or Private-Key encryption. The same key is used in encryption and in decryption.

The security of a symmetric key algorithm depends on both the length and quality of the key and the properties of the algorithm. The key is discussed in section 2.5.1. Good characteristics of a symmetric key encryption algorithm are discussed in [16]. *Confusion* is one important property, which is when one could not predict how a change of a plain-text character would affect the cipher-text. Another important property, called *diffusion*, is that information from the plain-text should affect the entire cipher-text. If these properties are achieved the algorithm will be hard to break.

Symmetric encryption is preferred when there is a high amount of data that will be sent, because of much better performance than asymmetric encryption. Examples of widely used symmetric algorithms are DES, 3DES and RC4. However asymmetric encryption has some important benefits in favor of symmetric encryption.

2.1.2 Asymmetric encryption

The following presentation follows very much the outline in [7, 16]. Asymmetric encryption or *public-key* encryption as it also is called has got its name from the property of the keys. Each entity has a pair of encryption and decryption keys K_e and K_d . The decryption key inverts the encryption of the encryption key, so $P = D(E(P, K_e), K_d)$ as in figure 3. Thus different keys are used for encryption and decryption. The encryption key and the encryption algorithm are made publicly known.

A sender, who wants to send a secret message, encrypts the message with the

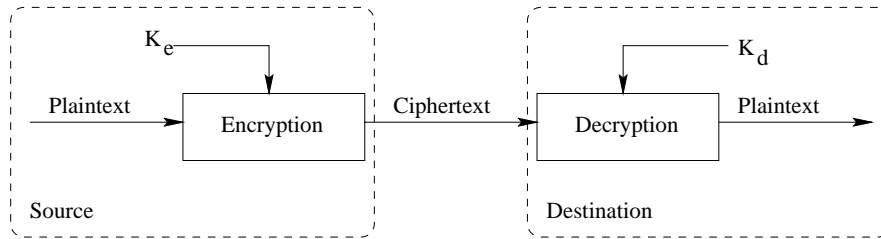


Figure 3: Asymmetric or Public-Key encryption. Different keys are used in encryption and decryption.

receiver's public key K_e or K_{pub} , since it is publicly available and sends the message. Only the receiver can decrypt the message, since it only can be done using the receiver's private key K_d or K_{priv} . In the following the notation K_{Ipriv} and K_{Ipub} is used for a specific private and public key owned by the entity I.

If we consider the public key system as it is solved by the RSA algorithm it has an additional property: E and D are both publicly known and are inverse functions of each other, so $P = D(E(P, K_{pub}), K_{priv}) = D(E(P, K_{priv}), K_{pub})$. This property gives the opportunity both to make a message secret and to authenticate a sender.

If someone, e.g. entity B, sends a message encrypted by the receivers public key, like $C = E(P, K_{Apub})$ then only the receiver, entity A, can decrypt the message since the key K_{Apriv} is needed to decrypt C . Then B knows it will be safe when it is sent. This scheme is used when one wants to keep the message confidential.

The integrity of a message, i.e. when only authorized entities can manipulate a message, is kept by using the scheme $Message = E(P, K_{Apriv})$. This message can any entity that knows the corresponding public key decrypt. Thus every one will know that the message comes from entity A, since it only could have been produced by A's secret key.

Furthermore if entity A wants to send a message P to B $E(E(P, K_{Apriv}), K_{Bpub})$ provides both security of the message and authenticity of the sender. Only B can decrypt the outermost encryption and only A could have produced the innermost encryption.

How robust an asymmetric encryption algorithm is depends on how difficult it is to solve a very hard mathematical problem.¹ The robustness in the RSA algorithm relies on the difficulty in factoring a large number on which the keys depend. Common for asymmetric key algorithms are that the encryption and decryption procedure requires calculations with huge numbers, which affects the computation speed negatively.

¹The security of the first asymmetric key algorithm, invented by Diffie and Hellman, depends on the difficulty in calculating discrete logarithms in finite fields.

2.2 Key distribution

2.2.1 Symmetric key systems

When only symmetric encryption is used, each entity has to share a unique key with each other entity or group of entities it would send data to or receive data from. If an entity wants to communicate with a large amount of other entities it becomes a problem to store all different keys needed. In the Enclaves system [10] they solve this problem partly by having a group leader who generates the symmetric key used by a group and administrates distribution, revocation and updating of that key. Thus keys used for group communication would not be necessary to store any longer than the group exists. Still each entity must have a shared secret key with each other entity with which it wants to have a secure two-part communication. The number of keys required in such a system increases approximately with the square of the number of entities.

To decrease the number of keys in a system one could have a key server that generates a key for each new pair of entities that want to communicate. This implies that each entity only needs to keep a shared secret with the key server. However this scheme result in high traffic on the key server, which might become a bottleneck in the establishment of a new secret communication.

Another problem is how each entity initially gets the keys it shares with other entities. One solution is that the system has to be initialized with keys for each possible pair of entities, but this assumes a static system. The asymmetric key system gives an elegant solution on this problem.

2.2.2 Asymmetric key systems

The asymmetric key system reduces the number of keys needed per entity to one pair of asymmetric keys. Unfortunately is the speed of encryption/decryption too slow to use in a session to send data back and forth. Therefore one wants a symmetric key system to use in the session, but an asymmetric key system to distribute the symmetric keys with. A key distribution protocol between entity A and entity B could be the message $E(E(K_{A-B}, K_{Apriv}), K_{Bpub})$ from A to B when A has chosen the symmetric key K_{A-B} . To ensure that both parties have got a fresh key and not a replay of an old key distribution message A and B could include sending and re-sending of an encrypted time-stamp or random number into the distribution protocol.

One major problem for an entity that uses another entity's public key is how he can know or *trust* in the mapping between a public key and the identity of the other entity. If for instance A wants to send confidential information to B, but A doesn't have either the public key or the address to B. Then C wants to get that secret data and claims he is B. A picks C's public key and address and sends the data intended for B to C. A is satisfied because he believes he has sent the data to B. B doesn't know anything about this event. C is satisfied, he can read the secret data because his key was used in the communication. If A had used B's public key for encryption but had sent the data on wrong address to C, no harm would have happened because

C couldn't have decrypted that message. Now, what A needs is a guarantee that a public key and an identity are mapped to each other.

One solution is that A gets a public key in a trustworthy way from each entity he wants to communicate with, perhaps by a physical hand over. This scheme would imply that A can trust all those entities, but it would force A to keep a public key for each such entity. Thus again we have the storage problem of keeping a great amount of keys. Instead one only want to keep a public key to one or a few entities that can guarantee the validity of the connection between a certain pair of public key and identity.

In Public Key Infrastructure (PKI) systems the goal is to establish trust among communicating entities through the possibility of binding a public key to the identity of an entity. Furthermore it is possible for an entity to validate this binding and thus get assured that the binding still is valid. A certificate, that not is possible to falsify, is used to bind a public key to an identity. A Certification Authority (CA) is an entity that creates and revokes certificates. Certificates are often given a certain valid period, which is included in the certificate.

If an entity A wants to communicate with another entity B, and be confident that the other part has the right identity. A needs a *chain* of certificates that ends up with the other entity's identity and passes by a CA that A trusts.

When an entity validates a certificate, it checks whether the certificates in the chain are within the valid time-period and that no certificate is revoked. Whether a certificate is revoked can be controlled either off-line, e.g. by retrieving an appropriate Certificate Revocation List (CRL), or on-line, e.g. by querying the CA that created the certificate.

Entities that use a PKI system to establish trust among collaborating parties, only have to keep knowledge about a limited number of entrusted entities, i.e. trusted pair of identity and public key. The idea is to build up a hierarchy of entities, where an entity at a low position trusts some entity at a higher position that in turn can guarantee the binding of a certain identity and public key.

However, even if a PKI system is used to establish trust among entities in a system, each entity have to trust in at least one entity, which is a CA, from the beginning. This must be carried out in a trustworthy way for each entity. For a more thorough survey see [3, 4].

2.3 Hash functions

A one-way hash function, H , is a function that takes an input of variable length and produces a value, called hash value, with predefined length. H has the following properties on the message, M , and the hash value, h :

- It should be easy to compute, $h = H(M)$.
- If h is given it should be very hard to compute M , i.e. the inverse of H is hard to figure out.

- It is very unlikely to find two messages that give the same hash value by the same hash function.

Another name for a hash value is a *message digest*. Often a hash function is used to produce a kind of finger print of a message. If two parties use a hash function to compute finger prints on messages sent to each other, they either have to keep the hash function secret or have a second input, which is a shared secret, to the hash function.

What happens if A sends a message to B, and on the way C tampers with the message? If there is no hash value appended to the message, B could not discover that it is manipulated, while he does not know the content of the message. If there is a hash value B can compute a hash value from the message and compare it with the appended hash value and thus discover if it is the original message.

2.4 Security goals to reach

In the area of computing and communication between computers one wants to reach the principally security goals *confidentiality*, *integrity* and *availability*. Confidentiality means that only authorized entities would have *access* to data. When sending data on a network this implies that no one without access right can read, manipulate or even know about data in traffic. Integrity of data means that only authorized entities can *modify* that data. This implies that nobody can modify or create a message without being authorized. At the same time as keeping the data's integrity and confidentiality one wants to make data *available* to those who have authority to access the data. An authorized entity should not be denied access to data, i.e. *denial of service* (DOS).

In some sense there is a trade off between the confidentiality and integrity goals and the availability goal. If one considers the property how fast one can access data as an availability characteristic, then would high confidentiality and integrity affect availability negatively. This is because of the computation cost high security involving cryptography results in.

There is also a trade off between cost and security. A high level of security has a high cost, both in terms of performance and system complexity. It is not advisable to use stronger security means than is necessary for the goals one wants to reach.

With the SAFEERLANG system we want to reach a very high security level. We focus on confidentiality and integrity aspects in security. There is, however, also efforts to make an efficient system and in that sense we aim to reach availability goals to. However, we do not consider DOS threats.

Much of the security is transparent for the programmer, so it is important that he can rely on the security in the system and by SAFEERLANG be able to build applications with very high security requirements. A goal is to make an easy to use system, that eliminates a lot of implementation errors. According to Ross Anderson in [1] bad implementations is one of the most common reasons to security shortages. Very few of the successful attacks against the systems discussed by Anderson was due to failures in algorithms or protocols, rather the major part of the security problems were related to bad implementations, poor key management, etc.

Another ambition besides an easy to use system is to prevent all common types of attacks on systems operating on the Internet. Those different threats are discussed in the next section.

2.5 Threats to communication

A lot of factors makes the communication in an open network vulnerable. The media is public, the techniques have undergone a rapid development, the network has grown rapidly, etc. A system's reliability depends on all parts in the system and the weakest link will set the level of security the system would reach. Therefore it is important to be aware of the different kinds of attacks that are known, although new kinds of attacks will appear when new weaknesses in systems are discovered.

2.5.1 Key length and quality

The security of a symmetric and asymmetric key algorithm is proportional to both the length and quality of the key. A key with bad quality, e.g. the telephone number of the firm, would make the encryption algorithm vulnerable, since an attack first of all checks such bad key choices. A too short key would be easy to figure out by *brute force* methods, i.e. checking all possible permutations of a number of same size as the key.

It is a trade off between cost and the value of the key if it is feasible to crack the key. The cost is due to the price for computer power. Consider that Moore's law claims that each eighteenth month computing power has been doubled. This means that a key that is safe today may be unsafe tomorrow. The value of the key is due to how an intruder estimates the value of the secrets the key protects. A discussion about this can be found in [17].

2.5.2 Types of threats

There are a number of possible attacks on a system in an open distributed network. They are often categorized as *interruption*, *interception*, *modification* and *fabrication* showed in figure 4.

Threats during an ongoing session:

The normal situation is depicted in figure 4a) but several different events could threaten the communication.

When a communication channel is interrupted the availability goal is threatened. This occurs when a wire is physically cut, when a network is partitioned or if a host is crashed. In this case there is a state of denial of service, which might have been caused by an attack.

Eavesdropping is when someone illicitly intercepts a communication channel and reads the data. If the traffic is encrypted eavesdropping means that the intruder can decrypt the data. This kind of attack breaks the confidentiality of the data. One sort of eavesdropping is *traffic analysis*, which could be done without decrypting the

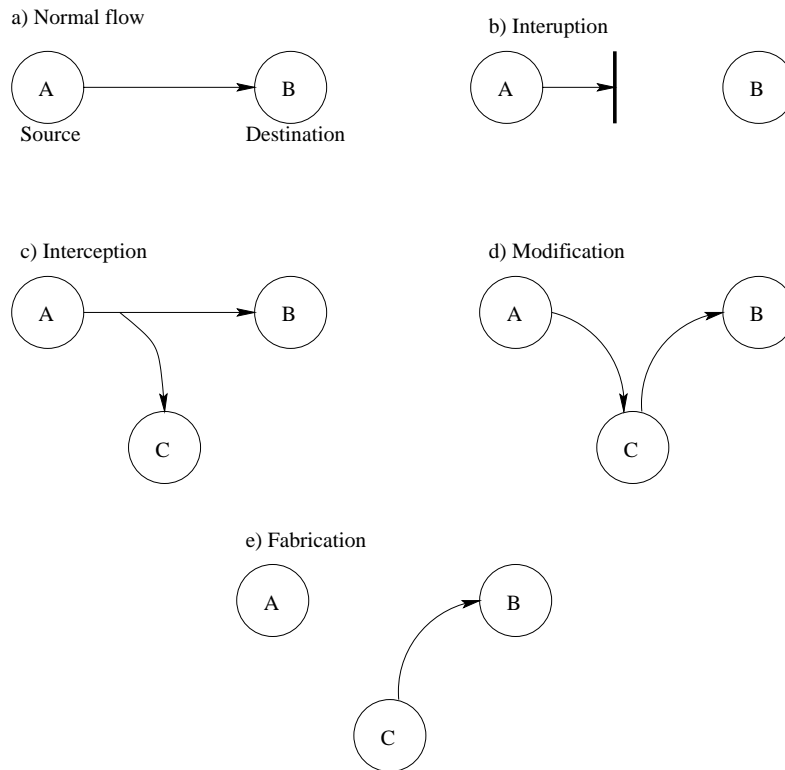


Figure 4: Threats when A sends message to B. C is an intruder that in c) illicitly listens to the traffic between A and B, in d) illicitly changes a message and in e) fabricates a message that seems to come from A. This figure is copied from [19].

traffic. Traffic analysis looks at addresses of source and destination, amount of data sent and other characteristics of the traffic. After that some conclusions is made from the collected statistics. Eavesdropping and traffic analysis are interception attacks, which are hard to discover since neither the sender nor the receiver would discover any changes to the communication.

Switching or modification is when a third entity reads a message and replaces it with a modified one. In this case the intruder also has to know the encryption if the traffic is encrypted. The integrity of the data is affected by this attack.

When a message is fabricated by a third entity C and sent to B in the communication channel between A and B there is a *masquerade* or *forgery* attack. C pretends to be A when communicating with B. If the A to B communication is encrypted C must be able to encrypt it according to the scheme used by A and B. A special case of fabrication is *replay* attacks, i.e. a message is resent. To do this kind of attack it is not necessary to decrypt the message. In this case the intruder just has to copy the

message and re-send it at an appropriate time. In this class of attacks the entities authenticity is broken.

A *man in the middle* attack (MIM attack) is when someone tries to become a not observed middleman between two communicating parties. If A and B communicates then an intruder C impersonates A for B and B for A, see figure 5.

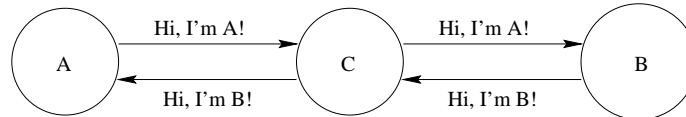


Figure 5: MIM attack by C.

In this attack C might be listening and just forwards the messages between A and B or he can modify or create messages too.

Threats during the establishment of a communication:

The description of the different threats above and in figure 4 has threats against an existing connection in mind. However, the most of the threats does also exist in the establishment phase of a connection. Since traffic analysis is an attack that often takes a lot of data into account and gathers statistics from the data, it is most often considered in the case of an ongoing session.

According to Chow and Johnsson [7] one can effectively prevent confidentiality and integrity attacks with cryptography. A critical attack is replay attacks, i.e. when an intruder uses a recorded message from a previous establishment phase. Thus the most protocols for establishment of secure communication focuses on this problem. See [7]. For further discussion on secure protocols see section 5.1.

10th October 2000

3 ERLANG

ERLANG is a programming language originally developed by Ericsson Computer Science Laboratory for telecommunication applications. These kinds of applications must have a high degree of robustness and real-time properties.

3.1 ERLANG properties

ERLANG is a concurrent functional language with a declarative syntax. It has soft real-time properties and is developed for distributed fault-tolerant systems. ERLANG has an error handling system with `throw/catch` clauses developed for dealing with errors in a distributed environment. Below are some ERLANG concepts relevant for this paper introduced and explained. For a full and detailed presentation see [2] and [8].

3.1.1 ERLANG basics

ERLANG has *constant* and *compound* data types, where constants is either numbers (integers or floats) or atoms. Atoms are typed with a beginning lower-case letter or within apostrophe signs as: `atom` or `'Atom'`. The compound data types are *lists* and *tuples*. Lists are used for storing a variable number of items and are written like: `[]`, `[atom, 10, 'Atom']`. Tuples can store a fixed number of items and their syntax are as: `{}`, `{1, atom, 12}`.

Variables can be used to store any ERLANG data type. They are typed with a beginning upper-case letter and can be assigned only once. When assigning a value to a variable *pattern matching* is used. Here follows some examples of pattern matching with the `=` operator:

<code>A={a, 10}</code>	The tuple with elements <code>a</code> and <code>10</code> is stored in the variable <code>A</code> .
<code>{a, B} = {a, 10}</code>	The value <code>10</code> is stored in variable <code>B</code> .
<code>{a, B} = {b, 10}</code>	This pattern matching will fail, because the constants <code>a</code> and <code>b</code> will not match.

ERLANG uses pattern matching when calling a *function* too, which the example from [2] shows:

```
...
convert({fahrenheit, Temp}, celsius) ->
    {celsius, 5 * (Temp - 32) / 9};
convert({celsius, Temp}, fahrenheit) ->
    {fahrenheit, 32 + Temp * 9 / 5};
convert({reaumur, Temp}, celsius) ->
    {celsius, 10 * Temp / 8};
```

10th October 2000

```

convert({celsius, Temp}, reaurmur) ->
    {reaurmur, 8 * Temp / 10};
...

```

When the function `convert/2` is called the terms in the function call will match against the patterns in the function definition. If a match occurs the expression following the arrow will be evaluated and returned. Pattern matching also occurs in other statements in the language.

3.1.2 Processes, ports and concurrency

ERLANG is a *process* oriented language, where each ERLANG process is a light weighted process with own memory heap. Such a process runs as a parallel computation scheduled and administrated by its runtime system. A process is created by the *BIF*² `spawn/3`³ like:

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

The argument `Module` is the code file where the function `FunctionName` is implemented. `ArgumentList` is a list of all necessary arguments to the function. For a process to be spawned on another node one has to use the function `spawn/4`. The functions `spawn/3` and `spawn/4` returns an identifier to the process to which it is possible to send messages and receive messages from. In ERLANG the message sending is asynchronous, which means that the sender will not wait for an answer from the receiver when a message is sent. The exclamation mark is the send operator in ERLANG. By the following code a message `Message` is sent to the process identified by `Pid`:

```
Pid ! Message
```

A message is received by a `receive` statement. If a process comes to a receive statement it waits, i.e. suspends, until a message matching one of the receive statements arrives. The syntax of a receive statement is:

```

receive
    Message1 ->
        do something;
    Message2 ->
        do something;
    ...
end

```

The patterns in `Message1` and `Message2` match against arrived messages. Each process has a message box into which all messages arrive and are kept in the same

²Built-in functions BIFs are a number of functions performing such things that are hard or impossible to implement in ERLANG code. See [2] for more details.

³The `'/3'` characters in for instance `spawn/3` displays the number of arguments required by the function and is called the functions *arity*.

order as they arrive. When a match occurs the message in the message box of that process will be removed from the message box and the statement following the arrow will be evaluated.

Each time the ERLANG system needs to communicate with a resource, (e.g. a file, another ERLANG node, etc.), outside the system it self, this is managed by a *port*. A port is opened by the BIF `open_port/2`. A port is similar to a process in that one can send messages to it. The process executing `open_port` manages the port and receives all messages that is sent to the port, this process is called the *connected* process.

3.1.3 Distribution

An ERLANG runtime system running on a machine is called a node. If a node will be able to communicate with other nodes it must be running in an *open* mode, which is decided on the node start up, see [2]. For a process to send a message to another process on another node, on the same machine or on another machine on the network, the syntax is almost the same as for sending a message to a local process:

```
Pid ! Message  
or  
{Name, Node} ! Message
```

The first alternative sends a message to a process on another node. The *process identifier* `Pid` is either the return value from `spawn/4` or earlier received in a message.

The second alternative sends a message to a *registered* process; where `Node`, the name of the node, is an atom and is unique in the entire network. `Name` is the registered name of the process on the remote node. For a process to be registered on a node the BIF `register(RegisteredName, Pid)` must be invoked. `RegisteredName` is an atom and the process `Pid` is reachable by that name on the local node. When a message is sent to a process on another node to which there is no TCP/IP connection present a connection is established automatically. The programmer does not deal with explicit TCP/IP connections between nodes.

A connection to another node can be setup in other ways too. Generally one only has to use a name of a remote node in any expression that involves a remote node. For instance by using the BIF `spawn/4`, where one must specify the name of the remote node.

3.1.4 Real-time properties

One of the main purposes with ERLANG was to make a system with soft real-time properties, which is required for instance in telecommunication applications.

Scheduling: In the system there are several simultaneously running processes. Thus all processes must get a fair time slice to execute within. The run-time system schedules all processes to achieve this fairness.

Memory management: Another property that is important in this context is the memory management, i.e. allocating and deallocating of memory. These tasks must not block the system. This also demands a garbage collecting that will be efficient and that will spread out the tasks in time. For a deeper discussion about the garbage collector see [12].

I/O: A third quality that is important is that I/O is non-blocking. This includes I/O on the local host as well as external I/O. Today ERLANG does not support non-blocking I/O on the local host. The external I/O is, however, more crucial in this respect. All these I/O features are handled through ERLANG ports.

When a port is opened the system controls which type of I/O it is going to perform. After that it is connected to the proper *driver*⁴. For instance, if an external socket connection is desired the port would be connected to the Internet driver (inet driver), see section 6.3.1. The driver implements the low level functionality needed for the I/O.

Each time the system schedules a process it checks if there are any ongoing I/O attempts. If so each pending I/O would get the opportunity to perform one write or read each.

3.2 Security deficiencies in open distributed environments

There is no possibility for an Erlang programmer to directly access memory through pointers or memory allocation. This is a benefit that eliminates a lot of runtime errors and unauthorized memory accesses. However, there are some security deficiencies in Erlang. In [15] and in [6] a number of such limitations are defined.

3.2.1 Resource management

In ERLANG a process can consume all available memory or spawn any number of other processes. In the case when all available memory is used the node will crash, in the other case the process, that tries to spawn a process more than the system can manage, will crash.

3.2.2 Access rights

The access rights on processes, nodes and ports are unlimited. For instance, anyone can kill any process by the BIF `exit/2` and a node can be halted using the BIF `halt/0`. This is not acceptable when intruders can get access to a running ERLANG system.

3.2.3 Code loading

When an ERLANG program runs that uses functions from other modules the code is loaded into the executing environment at run-time. The module is loaded by

⁴A driver is an interface to handle communication through ports, see [2].

name of the file that constitutes that particular module. If a program was written intended to use a module that is exchanged by another module with the same name in the environment where it is running, then the program would not behave as it was intended. It would load the new module that exists in the running environment, because modules are always searched for on the local file system. Thus ERLANG does not support loading of code remotely.

Furthermore loaded code is not controlled. So, if the code misbehaves it would cause damage to the running system.

3.2.4 Insecure communication

The communication between nodes is not encrypted. Today the messages are sent in plain-text.

Furthermore in ERLANG a *cookie* is used in the establishment phase of a connection and later on appended to all messages to get a kind of *authentication* between nodes. A cookie is a secret random number. When a node wants to communicate to another node it has to know the cookie of that node. Moreover, for each communication channel a node has an out-cookie and an in-cookie. A node's out-cookie corresponds to the other nodes in-cookie and vice versa. When a message is received in a run-time system the appended cookie is compared to the node's in-cookie for that connection. If the comparison fails the `net_kernel`, see section 6.3.2, is noticed about the unauthorized attempt and takes care of it. This scheme gives only limited security, see section 4.5.2.

10th October 2000

4 SAFEERLANG

The SAFEERLANG venture started during a master's thesis project by Gustaf Naeser [15], which focused on mobile code and related security issues. Naeser implemented a prototype of SAFEERLANG, called PoSE (Prototype of Safe Erlang) in this paper. Later Lawrie Brown, during his sabbatical, improved and enhanced the prototype and called the resulting prototype SSErl [5]. In 1998 Otto Björkström implemented a distributed application using SSErl to evaluate how useful the prototype was. Dan Sahlin at CSLAB was supervisor for Naeser and Björkström. He also took part in the work of Brown and he has been a spokesman for the ideas behind SAFEERLANG.

Simultaneously with the thesis this paper describes Rickard Green investigates and implements the access control mechanisms in a real ERLANG system, see [12]. The purpose is to merge the implementation of Green and this thesis into a real ERLANG system, which later on will be extended with other features from the prototypes. These current implementations of secure properties in a real ERLANG system are, in this section, referred to as SAFEERLANG, i.e. the same term as used for the general ideas and the theoretical discussions in the project that concerns all work from the beginning [5, 6, 12, 15].

The effort with SAFEERLANG has been to make ERLANG safer, especially in distributed environments. Areas where limits have been defined and solutions have been implemented in prototypes are access control on resources (as processes), restricting the use of unreliable code (by means of *sub-nodes*, see section 4.1) and remote module loading [6]. However, the previous works have not dealt with secure communication between different ERLANG nodes, though the need for it has been stressed.

The following subsections give a short description of the SAFEERLANG project and solutions up till today. Each following subsection have an introductory presentation of each issue that presents the general idea behind the solution. Then for each prototype (PoSE and SSErl) and the current implementation (SAFEERLANG) it is stated in what extent they solve that problem.

4.1 Resource management

The resource management limitations in ERLANG is solved in SAFEERLANG by extending the node concept; In ERLANG a node is an executing ERLANG system, but in SAFEERLANG it could be a hierarchy of nodes, where the runtime system is the most privileged of these nodes. From the run-time system sub-nodes could be created, which could be given limited right to use system resources. The idea is to run unsafe programs in sub-nodes, then only the sub-node will run out of resources.

PoSE This prototype implements this concept.

SSErl The sub-node concept is also implemented in this prototype.

SAFEERLANG Sub-nodes are not implemented in this system.

4.2 Access rights

The prior works on SAFEERLANG address the need to limit access rights on processes, nodes and ports. To get a control mechanism for accesses on resources capabilities have been used. They are created and validated on the node that possesses the resource. Capabilities are unforgeable references to such resources, and they contain the rights one can access the resource with.

PoSE In this prototype capabilities are used only for processes and nodes. They are divided into two parts, one for the reference to the protected entity and the other for the access rights. For protection from tampering one suggestion is that the entire capabilities should be encrypted. This implies that each time a capability is to be accessed it has to be decrypted, which creates a great amount of computation.

SSEr1 The use of capabilities was extended to ports⁵, modules⁶ and user defined references. This type of capability is not encrypted as a whole, but contains a protection value. This protection can be either an encrypted checksum or a password. In case of an encrypted checksum it is appended to the capability and each time it is used for access of the resource the checksum is validated. When password protection is used the creator of the capability chooses a unique password for each capability and saves a copy of each capability in a table. Every time the resource is accessed the capability has to be compared to the stored copy.

SAFEERLANG Processes, ports, references and user defined capabilities are protected by capabilities in this system. In this system the capabilities carries the public key of the origin node of the capability. A thorough discussion of access control by means of capabilities and how it is implemented in SAFEERLANG could be found in [12].

In this implementation a new capability can come up by the network, by restricting⁷ an old capability, by reading from text format or by creating a new resource, i.e. process, node, port etc.

4.3 Code loading

Since ERLANG doesn't support remote code loading the idea in SAFEERLANG is to have a kind of static scope for each code, so that the right code is loaded.

⁵A port in ERLANG is the gate through which ERLANG code can communicate with objects outside the ERLANG system, see [2].

⁶ERLANG modules is the main means to modularize code. In each module there is a separate name space. Functions in one module could be used in code outside that module. When code is executed used modules is loaded into the executing environment. See [2].

⁷When restricting a capability one limits the access rights.

PoSE and SSerl The prototypes address these problems by introducing module identifiers (Mids). This Mid is a reference to the origin of the module, and it could be used to reference code on a remote node as well as local code. Moreover when code is loaded one wants some type of code verifying. In Java the byte code is transferred and controlled by class loading and verifying in the Java virtual machine [11]. Transferring and verifying of compiled byte code has been discussed in [15] as a more efficient solution than the present one. The prototypes transfers the parse tree, then the code is compiled locally on the node that will execute the code.

SAFEERLANG This functionality has not yet been considered for a real ERLANG system.

4.4 Key distribution

The basic idea in SAFEERLANG, when secure communication channels are provided, is to append the public key of the node to each locally created capability, which have been performed in [12]. This results in that the public key will be spread to those sites that receives a *resource identifier*, i.e. process identifier, *port identifier*, *node identifier*, etc.

In the example in figure 6 a capability is created in A and thus contains the public key of A. Then the capability is transferred to node D via nodes B and C. Nodes A and D do not know about each other but when node D receives the capability it can access the resource on node A by a direct secure contact to A.

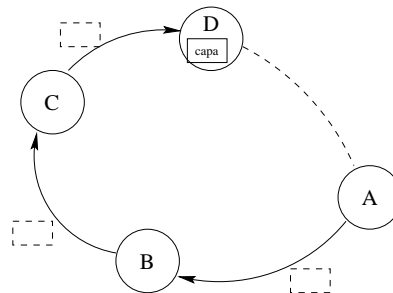


Figure 6: D has received a capability created by A. Now D accesses the resource on A.

This scheme leaves the decision, how an entity will get trust in a mapping between a public key and an identity, to the application programmer. Therefore a programmer has to be aware of which entities a resource identifier is distributed to. The desired level of trust in a system must be achieved by the application.

Another question that arises is how the system of reachable nodes ever can grow. If it only is possible to connect to a new node, e.g. by sending a message, if one earlier has received a capability from that node, by a direct or indirect connection, then the system will only live within some predestined boundaries.

What one wants is the ability to insert capabilities into the system from outside the borders of the system. One possible model for this would be some kind of application that serves the system (and other systems) with published capabilities and that also can receive capabilities for publication.

SSErl In the SSErl prototype it is possible to save a capability in a file as a binary term, and it is also possible to transfer a capability from one system to another in this form. With this feature it is easy to have a system where participating nodes change dynamically.

SAFEERLANG In the SAFEERLANG system [12] it will even be easier to transfer capabilities from one system to another than in SSErl. There it is possible to write a capability to text format, and thus it is very easy to publish a capability in any public catalogue, send it in an e-mail, etc.

4.5 Motives for a safer communication between ERLANG nodes.

The projects within SAFEERLANG have all assumed that the communication between ERLANG nodes was secure. The following subsection gives a motivation for encrypted channels and another authentication scheme than the present.

4.5.1 No encryption of data.

Since messages between ERLANG nodes are sent in plain text, see section 3.2.4, they could be read or tampered by anyone who illicitly monitors the traffic. In a SAFEERLANG context this is unacceptable, since messages could be fabricated and thus an unauthorized and harmful user could by purpose get access to secret data or access any resource.

PoSE, SSErl Neither of the prototypes have secure communication, though the need for it have been addressed.

SAFEERLANG The design and implementation of secure communication are presented in sections 5 and 6.

4.5.2 Limited security with cookies

The authentication by cookies briefly described in section 3.2.4 has severe security weaknesses.

This scheme is very easy to break for any malicious entity. It only has to copy the cookie that was sent in any previous message between the appropriate nodes, and then append that cookie to the fabricated message. Thus a third party can impersonate another entity or take over an ongoing communication.

PoSE, SSErl Both have this scheme left.

SAFEERLANG The use of cookies is changed to a system where each message is authenticated by a *MAC*, see section 5.3.

4.6 Summary of SAFEERLANG

This section summarizes the different limitations of ERLANG that have been mentioned in this paper and whether the prototypes and the SAFEERLANG implementation deals with each problem. The implementation of SAFEERLANG is divided into the part done by Green and the part performed during this thesis project, as Green and Karlsson respectively. However, the Green and Karlsson implementations are dependent of each other and are not intended to work independently. The summary is presented in table 1.

Limitation in ERLANG	PoSE	SSErl	SAFEERLANG, Green	SAFEERLANG, Karlsson
Strong authentication	no	no	no	yes
Secure communication	no	no	no	yes
Controlled memory consumption	yes	yes	no	no
Limitation of number of possible processes	yes	yes	no	no
Access control	yes	yes	yes	no
Remote code-loading	yes	yes	no	no
Secure environments for untrusted code	yes	yes	no	no

Table 1: This table shows identified limitations in ERLANG and which of the implementations of a safer system that deals with the problem.

10th October 2000

5 Design of secure communication in SAFEERLANG

This sections contains a description of the design of the encrypted communication in SAFEERLANG. First it is a discussion on secure protocols, then the establishment or *handshake protocol* is discussed, and finally the *session protocol* is considered.

5.1 Secure protocols

This section discusses protocols used to establish a secure communication channel between two entities.

5.1.1 General properties

Four good qualities of a protocol are defined by [7, 16]: The participating entities have to be *aware* of the protocol. They must also *follow* each step of the protocol according to the predefined order. Furthermore each step must be *well-defined* so that no entity ever is uncertain of which is the next step. Finally must there for every possible situation be a *predefined action* to take. These qualities are a kind of common sense and must be fulfilled for a protocol to work properly.

Besides the confidentiality and integrity goals this kind of protocols also *authenticates* one or both of the participating entities and exchanges a *shared secret* among the participants. The authentication is that both parties should become convinced that the other participant is the intended one. The sharing of a secret most often is to establish a shared session key used only by the participating entities in a communication session.

Arbitrated protocols use a third neutral party that authenticates the both entities. These protocols give a strong and effective authentication. However, there are some disadvantages with this type of protocols that often make it preferable to use other types of protocols. Among the disadvantages are: It might be hard to find a mutual trusted third party, the communication overhead for a third party might be high, it might be expensive to consult a third party, the third party becomes a target for intruders, etc.

If one wants a protocol that requires less network communication than an arbitrated protocol, but where a third party is available if it is needed, an *adjudicable protocol* may be the solution. In this type of protocol the third party is only invoked in case of a disagreement between the two parties. The idea is that during the protocol sequence enough information is saved so that a third party can judge in case of a dispute. The drawback with this protocol type is that the problem can be solved only afterwards a failure has happened.

According to [16, 17] a *self-enforcing protocol* is preferred before a protocol with an arbitrator or an adjudicator. In this kind of protocol both parties are guaranteed a fair treatment without involving a third party. If anyone tries to cheat it is discovered immediately.

Discussions of this kind of issues can be found in [7, 16, 17].

5.1.2 Authentication and secret-sharing

Protocols that perform authentication and secret-sharing use either a symmetric key scheme or an asymmetric key scheme. Moreover, either they involve a third party or not. In [7, 17] they discuss how these protocols have evolved, and what kind of threats the protocols have defeated. Pfleeger [16] gives an overview of the field. In the following we focus on self-enforcing protocols of the type that use an asymmetric key scheme.

In this scheme the entities, A and B , mustn't know the other entity's public key in advance. Instead, as a first step in the most general scheme, one receives the key in a trustworthy way, so that each entity is confident that he tries to contact the intended part and not anyone pretending to be the other part. The next step is that the entity initiating the communication, A , generates the secret that will be shared and sends it to the other entity like $E(E(\textit{Secret}, K_{Bpub}), K_{Apriv})$. B then decrypts the message with A 's public key and his own secret key, thus he gets the secret and it is shared. A knows that only the possessor of the K_{Bpub} can get the secret, and B knows that only A could have encrypted the secret since it is done with K_{Apriv} . The first step in this case involved the authentication and the second step involved the secret-sharing.

If the authentication part is weak an intruder may interchange its own public key with both A and B , and thus become a middle man in the communication and be able to eavesdrop and modify messages.

5.1.3 Time-stamps and nonces

A crucial task for a protocol, according to [7], is to prevent replay attacks. An intruder may record a message sequence from an earlier establishment phase, and then try to replay an old message sequence in order to establish a session with a previously used session key. For instance could an intruder record the message $E(E(\textit{Secret}, K_{Bpub}), K_{Apriv})$ in the simple protocol above. Assume that a session continues immediately after the secret is shared (and either the secret is used as a session key or a session key is generated from the secret), then the intruder can start a new session and hinders the messages from B to reach to A . Then A doesn't recognize the communication and B thinks he talks to A . Therefore one wants the ability to control the freshness of the secret the other part wants to make shared.

Two different means are used in protocols to control freshness. Either is a time-stamp or a nonce, i.e. a unique random number used only once, appended to the messages in a suitable way, and thus one can control if the message belongs to this sequence or not. If time-stamps are used it requires that the participating entities have synchronized clocks. The nonce technique requires that the generated random number is not predictable.

Finally, Schneier [17] states some important lessons learned from protocol development. Two of them are:

- Make every protocol feature explicitly.
- Optimizing a protocol may introduce new (not obvious) weak points.

5.2 Establishment of encrypted channels - the handshake protocol

The handshake protocol is the protocol used to establish a secure channel between two nodes. It is a self-enforcing protocol using an asymmetric key scheme to enforce confidentiality, integrity, authentication and secret-sharing. This protocol is run on top of the TCP/IP connection.

5.2.1 Key distribution

A primary precondition, in the design of this SAFEERLANG implementation, for a node to be connected to a remote node is that it has the public key of the remote node. This has it received earlier in a capability sent to it or reading it from text format, see section 4.4. The propagation of capabilities and thus the propagation of public keys follows the same semantics as resource identifiers in ERLANG.

Of course, each node also has to have an own pair of public and private keys, K_{Apriv} and K_{Apub} for node A. Every node have got these keys on node start up.

5.2.2 Security goals with the handshake protocol

When two parties A and B want to establish a secure channel between their nodes, they need to reach a state when they share a common symmetric session key, K_{A-B} , used to encrypt and decrypt data during the secure session protocol, described in section 5.3. There is a main security goal with the handshake protocol in SAFEERLANG, which is as follows:

- In the case when nodes A and B establish a secure channel the goal is that only the entities that own the private keys K_{Apriv} and K_{Bpriv} know about the shared symmetric key, K_{A-B} .

Within this goal there are two subgoals, first, there is an authentication that both parts know that the other entity is the owner of a specific secret key. And secondly, there is a need, in a secure way, to exchange a shared secret.

The first subgoal is desired because the pair of a public and a private key of a node is assumed to be unique⁸ for the entire network⁹. If node A wants to communicate securely with node B and node A has the public key of B (K_{Bpub}), then A wants to be sure it communicates with the entity that owns the private key that corresponds to K_{Bpub} . We want the quality that each entity (A and B) would discover if an entity with another private key than the desired one tries to start a secure communication.

There is also a requirement that:

- all common well-known attacks, e.g. brute force, replay and MIM attacks, on the Internet shall be prevented, see section 2.5.2.

⁸The uniqueness depends on the probability that two equal keys would be generated and the randomness of the algorithms used for generating keys.

⁹This depends entirely on how the administration of keys is handled. If for instance the same key is used by all nodes on the same host the assumption is wrong.

To summarize sections 5.2.1 and 5.2.2 we have:

The initial state:

node	knows
<i>A</i>	$\bar{K}_{Apriv}, \bar{K}_{Apub}, \bar{K}_{Bpub}$
<i>B</i>	$\bar{K}_{Bpriv}, \bar{K}_{Bpub}$

The final state:

node	knows
<i>A</i>	$\bar{K}_{Apriv}, \bar{K}_{Apub}, \bar{K}_{Bpub}, \bar{K}_{A-B}$
<i>B</i>	$\bar{K}_{Bpriv}, \bar{K}_{Bpub}, \bar{K}_{Apub}, \bar{K}_{A-B}$

5.2.3 SAFEERLANG handshake protocol - first version

The sequence of messages, showed in figure 7, is sent and received when A initiates a connection between A and B, e.g. when a process on A sends a message to a process on B.

The protocol uses the notation presented in section 2 above. The following comments to the handshake protocol shows the message numbers in parenthesis. When {} brackets are used in the protocol it only shows that the message contains more than one element, it has nothing to do with ERLANG tuples.

Initialization

The hello message (1) from A contains suggested *session id*, supported cryptographic algorithms and *compression*¹⁰ algorithm. B responds (3) either with the same contents, if B supports the suggestions, or with new suggestions from B. A session in this context is a specific communication period that lasts as long as the involved parties so wishes. It can went on from one socket connection through a period when the connection is interrupted to a new socket connection. The idea is that a pair of nodes could reuse a session key and algorithms that they already have agreed upon. The hello message also indicates the beginning of a handshake procedure.

A sends both the node name and the own public key to B in (2). This is necessary since node B do not have access to this data, and B needs A's key later in the protocol. The node name is propagated deeper into the run-time system, where it is needed for the ERLANG distribution to work, see sections 6.3.2 and 6.4. Messages (1-3) are sent in plain text, since the data in these messages are publicly available or is easy to guess.

Authentication

Node A needs to be sure about that the other entity is the owner of private key

¹⁰A technique used to reduce redundant data from a text, and to increase the encryption speed [17].

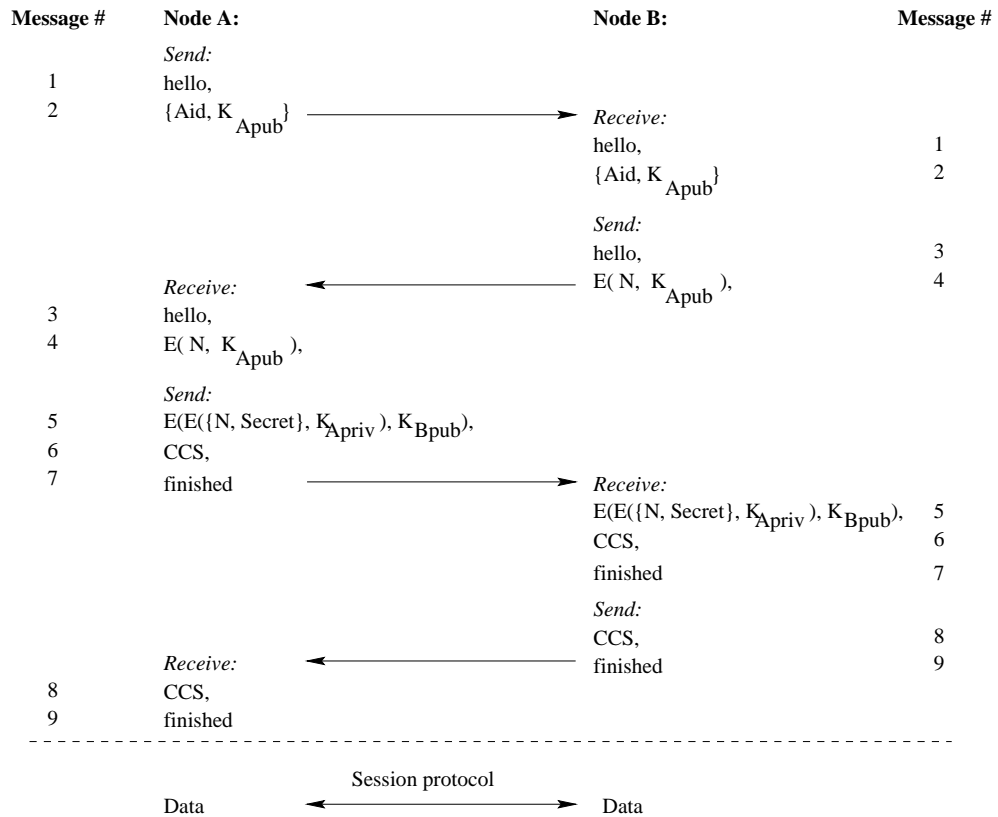


Figure 7: First version of SAFEERLANG handshake protocol.

K_{Bpriv} , and node B needs to know that the other entity is the owner of private key K_{Apriv} . This authentication is performed by messages (4) and (5).

When B sends message (4) it knows that only the entity with the private key K_{Apriv} can decrypt that message. When B receives the content of message (4) in the next message, encrypted with K_{Apriv} he knows that only the receiver of message (4) and the possessor of key K_{Apriv} could have sent this message. In this way B has been confident of the other entity's authorization.

In the other way around A wants to be sure that he communicates with the owner of K_{Bpriv} . When A sends message (5) he knows that only the owner of the key K_{Bpriv} can decrypt that message, thus the content of message (5) only will be known by the possessor of the keys K_{Apriv} and K_{Bpriv} .

This authentication makes the authentication performed by cookies superfluous. Thus it was removed when this protocol was integrated into the run-time system of ERLANG.

Shared secret

Message (5) is also involved in the scheme that makes the nodes to agree upon a shared secret, from which each node generates session keys. Since node A generates the secret, he only needs to know that only the possessor of the key K_{Bpriv} can read the message containing the secret. Node B wants to know that the possessor of the key K_{Apriv} is the only other entity that knows the secret.

Message (5) is encrypted with B's public key, K_{Bpub} . Therefore A knows that only the entity that owns K_{Bpriv} can open that message.

When node B receives message (5) it knows that only the entity with the key K_{Apriv} could have packed the secret from plain text, since the message is encrypted with that key.

The nonce

The N in message 4 is a nonce, i.e. a unique random number used only once. The idea with the nonce is to make the messages (4) and (5) unique in time, so they never could be reused in bad purposes, see section 5.2.5.

5.2.4 SAFEERLANG handshake protocol - second version

If two ERLANG nodes would use the same set of keys there is a serious weakness in the handshake protocol above. In that case the key K_{Apriv} and K_{Bpub} would be each others inverses. Thus message (5) would be sent in plain text over the network.

A solution on this problem would be to replace message (5) with: $\{E(\{N, Secret\}, K_{Bpub}), E(MD(E(\{N, Secret\}, K_{Bpub})), K_{Apriv})\}$. Which implies that only the possessor of the key K_{Bpriv} can decrypt the first part - $E(\{N, Secret\}, K_{Bpub})$ - of the message and thus get the secret or control the value of the nonce. The outermost layer of the other part of the message can anyone that has the key K_{Apub} decrypt. What they get then is a message digest, see section 2.3, of the first part of the message. This digest cannot be used to deduce the message from. Only the possessor of K_{Apub} can use it to verify the message.

This message have the same properties as the previous one according to authentication and the shared secret.

5.2.5 Different attacks

The discussion below about some conceivable attacks towards the handshake protocol refers to section 2.5.2. The pictures in this section shows the second version of the protocol. However, the first version also protects against those attacks.

MIM

If a node C tries to be a middleman between two nodes A and B we have a MIM attack. For this attack to be possible C must be able to prevent messages from A or B to reach the destination. Furthermore C must be able to read messages on the network.

The assumptions made for the behavior as is shown in figure 8 is that C have not access to any of K_{Apriv} or K_{Bpriv} . However the attack would also fail even if C had access to only K_{Apriv} , though C could have opened message (4) he never would have got the secret. But if C had had access to K_{Bpriv} the attack would have succeeded, although C could not read message (4).

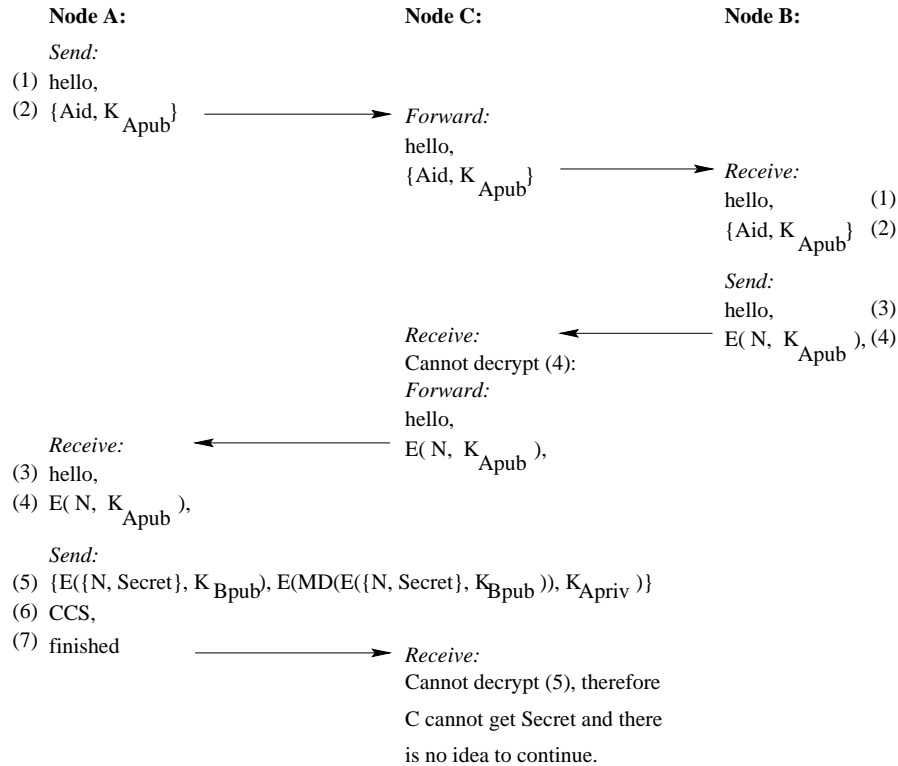


Figure 8: A MIM attack by C when A tries to connect to B.

An intruder as C has access to the keys K_{Apub} and K_{Bpub} , since they are publicly available.

In the first part C just forwards the messages and for each of the entities he claims he is the other part, see figure 5. Because C cannot decrypt the messages encrypted by A or B he cannot decrypt messages (4) - (7) or any message in the session protocol. The only message he can decrypt is the outermost encryption of the second part of message (5), which contains a message digest of the first part of the message. However, the message digest is also cryptographically secure. In this case it is no idea for C to be a middleman.

If C knows the key K_{Apriv} he can decrypt all messages encrypted by message K_{Apub} . In this case C can get the nonce in message (4). Still he cannot decrypt the parts of message (5) that contains the secret and thus the attack would fail.

On the other hand, if C knows K_{Bpriv} he can decrypt the parts of message (5) that contains the secret. However these attacks are prevented by use of strong keys in the protocol.

Replay

A common sort of attack is a replay attack. In this case an intruder previously have recorded an old handshake sequence. Maybe he have cracked the session key which the old handshake agreed upon and therefore he wishes to reestablish a connection that uses that specific old session key.

In figure 9 it is shown how the protocol discovers a replay attack.

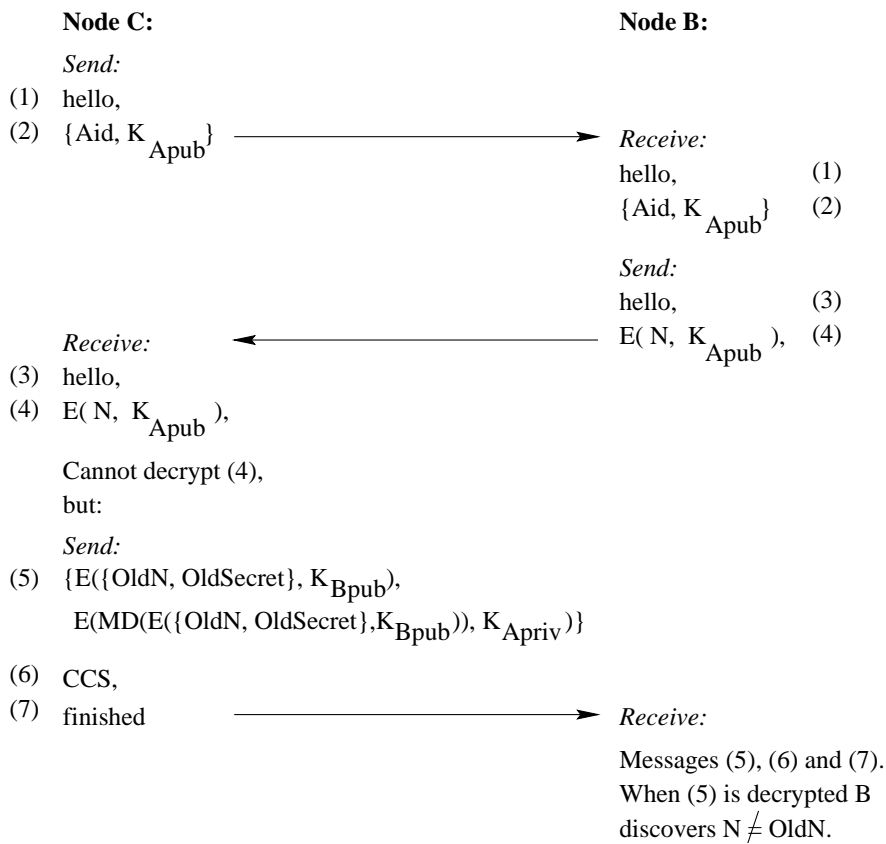


Figure 9: This picture shows how the protocol handles a replay attack from entity C.

When C receives message (4) he cannot decrypt it and get the nonce, N, thus he cannot compose an own message (5). Instead C tries to send an old recorded message that contains the secret, OldSecret, he wishes to use in the session protocol. However, when B receives the replayed message he discovers that the nonce is not the same as

he sent in message (4). Then B refuses to continue.

Other attacks

Brute force attacks on the keys are prevented by proper quality and size of the used keys, see section 2.5.1. It is up to the user of an ERLANG system to install good keys.

The handshake protocol does not send security sensitive details in plain-text, which prevents simple attacks.

However, this protocol has evolved from logic reasoning, and it has not been formally proved. Thus there is possible that hidden vulnerabilities exists.

5.3 The Session protocol

The security of this protocol relies on the security of the handshake protocol. If it is a poor handshake procedure before the session it does not matter how secure the session protocol is. However can this protocol fail upon own weaknesses. The reason for the chosen solution are discussed in this section.

5.3.1 Security goals with the session protocol.

The goals for this protocol is to reach a high level of security and thus we want to prevent the known kinds of attacks. However, DOS attacks have not been considered in this thesis.

The solution in the present system to reach some degree of security is to append a secret cookie to each message. However this will not give strong security as discussed in section 4.5.2. To give strong protection against take over attacks the communication is encrypted.

5.3.2 Why using a known protocol and implementation.

To create a secure protocol there is a lot of subtle threats to consider. A goal with the project is to reach a *reliable* secure communication, therefore we have tried not to invent to many subtle security holes. While choosing a well known, analyzed and widely used protocol for this part of the communication we got a reliable solution. During the initial phase of this thesis we studied some other protocols, and especially SSL, for secure communication. We found that with some, vital, changes of the SSL handshake protocol, we could use the entire SSL record (session) protocol. With the assumption that our SAFEERLANG handshake protocol properly sets up a secure communication, we would then have a secure communication.

The SSL Version 3.0 protocol [9], see appendix A, has been analyzed by [18, 20], they state that the record protocol is a very secure protocol. Below follows a short presentation of the security properties in that protocol, and how they prevent common attacks.

Key management

In the SSL handshake protocol a shared secret is established, from which session keys are generated. Separate keys are used for encryption in each direction of the connection. Each different instance of a connection do also have different keys.

This property gives a good protection against eavesdropping. The use of strong encryption keys makes a *known-plain text*¹¹ attack unfeasable, because it would be too laborious to make an exhaustive search. Strong keys is also an efficient defense against brute force attacks.

Message Authentication Code (MAC)

A MAC is a one-way hash function that also depends on a key. The input into the MAC function is data and a key. The output is a fingerprint of the data. Only an entity that has the same key as the producer of the fingerprint can verify the correctness of the data in conformity with the fingerprint.

SSL 3.0 uses different keys for encryption of data and for calculation of the MAC. Different keys are also used for each direction. The algorithm used by SSL¹² are strong and gives a good protection.

This property protects the message's integrity, e.g. modifications of messages.

Sequence number

A sequence number is included in each packet. Then the receiver can determine if anyone tries to re-send an old message or if messages arrives in wrong order.

Cipher-block chaining

The SSL protocol supports cipher-block chaining in the block-cipher mode. This technique starts with an initialization vector, a random number of fixed length, which is combined with the first plain-text block by exclusive-or before it is encrypted.

In figure 10 the cipher-block chaining method is showed. The IV is the initialization vector, P_1 and P_i is the first and the i :th plain-text block. In the same way C_i describes the i :th cipher-text block. The first cipher-text block is a combination of the first plain-text block and the initialization vector. Each of the consecutive cipher-text blocks are the encryption of a combination of the plain-text block in order and the previous cipher-text block.

By this method are patterns in the plain-text hidden. If two equal blocks are sent, they will not look the same as encrypted. Thus if traffic analysis looks at the patterns of the cipher-text, this method gives good protection. Furthermore cannot someone, trying to resend an encrypted block, just replace a block without beeing discovered.

Padding

The length of plain-text and cipher-text are the same in most encryption methods. In cipher-block chain mode short messages are padded with random padding. This

¹¹A known-plain text attack is possible when a packet has predictable data. To perform this attack one has to test all possible encryption keys on that text and compare it to the encrypted data.

¹²SSL uses the HMAC construction to compute a safe MAC [20].

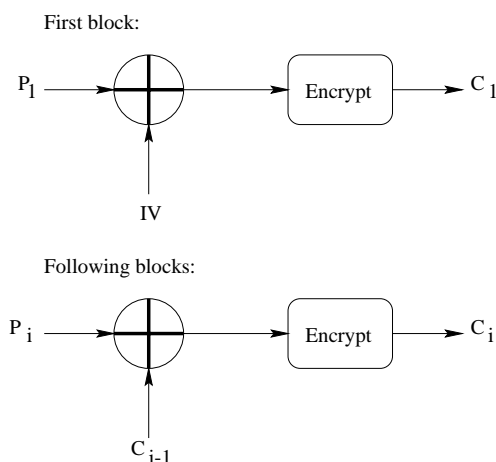


Figure 10: Cipher-block chain.

makes it hard for an intruder to know the exact plain-text length. He merely can guess the approximate length. This property protects against some aspects of traffic analysis.

Different ciphers are supported

Several different ciphers are available to choose as the preferred cipher. The entity that initiates a connection suggests a cipher to use. If the other part supports this cipher it will be used during the session, otherwise the other part chooses another cipher that the initiating part supports. This agreement is handled in the hello messages, see section 5.2.3.

5.3.3 Data flow in SSL record protocol

When data receives to the SSL record protocol from higher level application protocols it is uninterpreted. After that it undergoes the treatment showed in picture 11, see also [9], if the protocol uses cipher block chaining mode. Thereafter it flows into a lower level TCP/IP protocol for output.

When encrypted data is received from the TCP/IP level the scheme is followed the other way around.

The steps data flows through are:

Fragmentation The data can arrive in any length and are fragmented into a maximum length of 2^{14} bytes.

Compression Takes away redundancies in the text.

Message authentication An encrypted hash sum of the text. See section 2.3.

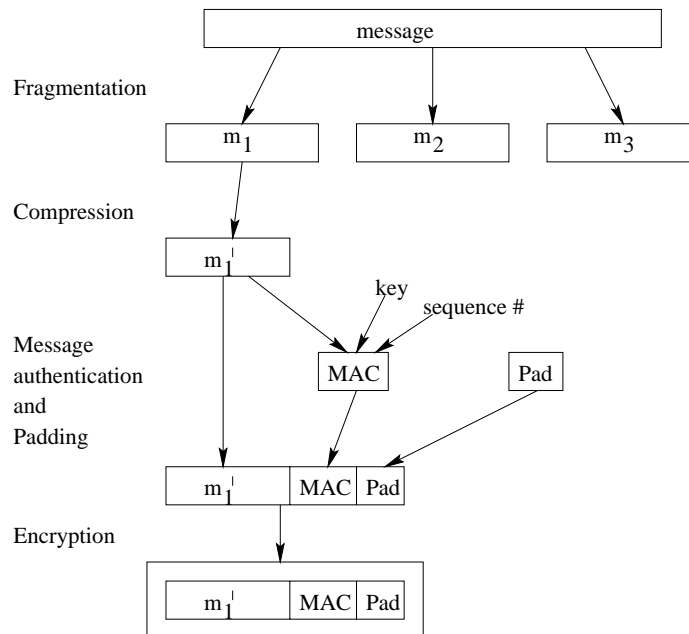


Figure 11: The data flow in SSL record protocol in cipher block chaining mode.

Padding Prolongs a too short text block with random bits.

Encryption Turns the entire compressed, MACed and padded block into a ciphertext block.

5.4 Easy-to-use system

As mentioned earlier a goal has been to create a system that is easy to use. In this system the programmer only has to consider to which nodes the resource identifiers are propagated, see section 4.4. The establishment of a secure channel is managed by the system, and the programmer is never involved in such matters.

6 Implementation comments

As already mentioned in the previous section we have benefits from the SSL protocol and an open source implementation of it [13, 14]. The following sections describes how the secure protocol has been implemented and how it works in the ERLANG run-time system.

6.1 Establishment of TCP/IP channels

When a message is sent to a node to which no TCP/IP connection exists a new connection is established transparently to the programmer. In figure 12 the most important events in the run-time system are showed that happen when a message is sent to a not connected node.

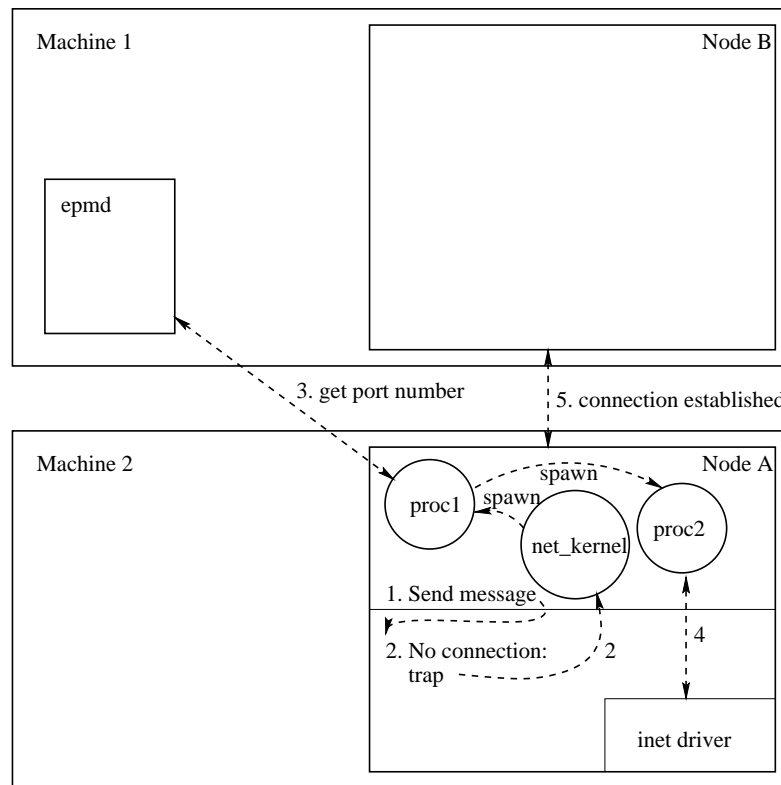


Figure 12: Establishment of a connection between nodes A and B.

In this case it is a process at node A that sends a message to a process on node B. The first event that happens is the ERLANG code sending the message that is executed, (step 1 in figure 12), i.e. `Pid0nB ! {FromPid0nA, message}`. When this

code has run the run-time system recognizes that there is no connection. An ERLANG *trap* function is invoked (step 2) which in turn tells the `net_kernel` to administrate a connection setup to the node in question.

The `net_kernel` spawns a process that among other things establishes a separate TCP/IP connection to the `epmd`¹³ process running on the other node, (step 3). The only information that is sent by this connection is the node name and the port number onto which it listens.

Another process is spawned that in communication with the inet driver (step 4) sets up a TCP/IP connection. The inet driver is a low-level interface by means of ERLANG ports to manage TCP sockets, see section 6.3.1. Then the connection is established (step 5).

This thesis has concentrated on encrypted channels on top of the TCP/IP connection established in step 5. It is by this channel the message passing is performed. We have found it unnecessary or rather not desired to encrypt the communication on the channel between node A and the `epmd` server on the other machine. The reasons are twofold, firstly the only information an intruder can get from that communication is the port number, that the other node listens to, and the name of the other node. This information is anyhow sent in clear text in the TCP/IP packet even in the case of an encrypted channel on top of the TCP/IP connection. Secondly, the performance would be affected negatively.

One drawback with insecure communication between the `epmd` server on Machine1 and `procl` on node A, step 3 in figure 12, is that node B is susceptible for DOS-attacks before the TCP connection is established.

6.2 The protocol implementation and something about SSLeay

The handshake protocol described in section 5.2.3 has been implemented using the open source implementation, version SSLeay-0.9.0b¹⁴ (SSLeay in this document), by Young, see [13, 14]. Modifications have been done on the handshake protocol in SSLeay so that it fits our purposes.

6.2.1 SSLeay

SSLeay is a well-known implementation of the SSL protocol. It has been used in a lot of applications, among others the ERLANG SSL-2.2.1¹⁵ application that can be used for explicit external secure socket communications. Furthermore it has been used as a base for the OpenSSL¹⁶ project.

¹³The ERLANG Port Mapper Daemon (`epmd`) is a server that maps each node running on a machine to a specific port number onto which the node is listening.

¹⁴This open source software has been down-loaded from <ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL>, and are, according to the README file on this ftp site, free for commercial and non-commercial usage. Though the author, Eric Young, must be attributed.

¹⁵For information on ERLANG and documentation on ERLANG applications see <http://www.erlang.org/>; Internet, accessed 11 May 2000.

¹⁶For information on OpenSSL see <http://www.openssl.org/>; Internet, accessed 11 May 2000.

The SSLeay package implements the entire SSL protocol, and is designed to fit different platforms. The implementation is well structured and easy to understand, although it is a rather big system. The system contains implementations of different versions of SSL and TLS. It also includes libraries that is necessary for SSL, e.g. encryption algorithms, big numbers, etc.

ERLANG is a system with real-time properties, with for instance non-blocking I/O on sockets. Therefore it is a requirement that SSLeay also supports non-blocking I/O, and so it does.

The parts of SSLeay that have been focused in the implementation part of this thesis is the handshake for each entity (client and server) in a connection setup. Since a SSL session is stateful it is the handshake protocol's responsibility to synchronize the two entities. Therefore are the handshake procedures, both for the client and server, implemented as a state machine. Each side proceeds from one state to another for every message that has been sent or received. In each state there are functions that serializes/unserializes the messages.

The structure of the underlying implementation in SSLeay is layered where each layer performs bookkeeping for the session or performs some of the steps presented in picture 11. Finally is the I/O controlled by BIO (Basic Input Output) structures that either makes a direct output (or input), i.e. write on socket, or writes into a buffer. If writing into a buffer the buffer have to be flushed if an explicit write is wished. Thus SSLeay supports sending several messages in the same TCP package.

6.2.2 The implementation

SSL uses certificates to authenticate the entities in a communication. The certificates are not used in this project. Instead SAFEERLANG uses another scheme for key distribution, see section 4.4.

If comparing the SAFEERLANG handshake protocol with the SSL handshake protocol according to the specification of SSL version 3.0, see appendix A, the SAFEERLANG handshake keeps the messages ClientHello, ServerHello, ChangeCipherSpec and Finished. These corresponds to messages (1), (3), (6) - (9) as showed in picture 7. All messages in the SSL handshake that treats certificates are not needed, and the key exchange messages is different.

This thesis project changed the part of SSLeay that implements the state machine mentioned in section 6.2.1 and implemented serialization/unserialization functions needed for the new messages. This implementation was rather straight forward. The hard thing was to figure out how the details in SSLeay worked and how to introduce the desired functionality without initiating new unwanted side-effects.

6.3 ERLANG communication

The second step in the implementation was to interleave the handshake and session protocols into the existing ERLANG system. The following subsections describe the different parts of the system that are involved in this step. The integration is described in the next section.

ERLANG is a system that is implemented both in ERLANG and in C code. The low level functions are implemented in C and higher level control mechanisms are implemented in ERLANG. Then one got the advantages from ERLANG with, for instance, asynchronous message passing and nonblocking processes that wait for a message to receive.

6.3.1 The Internet driver

The Internet driver (`inet driver`) module manages sockets for communication on a network. Each port that is used for external communication is connected to a descriptor, that is created when the port is opened. To the descriptor is also the socket connected.

The module also implements drivers for TCP and UDP connections. Those drivers perform actions on the socket when they are invoked by the run-time system. The run-time system discovers the need for those actions and when it is appropriate in time it invokes the driver, see section 3.1.4.

When one sends a message to a port that is opened for external TCP communication the message will result in a command to the TCP driver, that will perform an action.

Some of the commands from the ERLANG level to the driver are synchronized and require a reply from the driver when the action is finished. Even though the action is synchronized it does not mean that the system will hang. While the connected ERLANG process, see section 3.1.2, waits for a reply it suspends and thus an other process can be scheduled. Other commands are not synchronized.

6.3.2 ERLANG processes

The `net_kernel` is a process running when the node is started in open mode, see section 3.1.3. This process coordinates and manages actions when a connection to a remote node is setup. It starts processes that supervise the TCP sockets and the handshake procedure. The `net_kernel` and this supervising functionality is implemented in ERLANG code.

The invocation of the remote node's `epmd` server, see picture 12, is another action that takes place from this level. Another process involved in the implementation is the process connected to the port.

The parts of the handshake procedure that take place in ERLANG code is some controls of the socket connection, e.g. that a proper disconnection takes place in the case the other node goes down. There is also a control that no simultaneous attempts to set up a connection from each node occur. Moreover the cookies is managed from this level.

6.4 Integration of implementation in ERLANG and SSLeay

This implementations has been done in the R6B.0 version of the ERLANG run-time system. A decision that had to be taken was at which level the implementation should

be made. In ERLANG code it is very convenient to write protocols where you get the non-blocking mechanisms very easy. However for each call from higher level code to lower level code there is a performance loss.

One effort with the implementation has been to minimize the communication between higher level and lower level modules. The reasons for this are listed below.

- To get better performance. The communication from ERLANG to C code implies overhead, for instance by scheduling of ERLANG processes.
- To minimize the flow of security sensitive information up in the ERLANG code. There is no thorough analysis made for this aspect, but in order to minimize the weaknesses of the security this precaution was taken.
- The implementation work and complexity are approximately the same, since the existing structure in the driver supported this implementation.

In the driver module has a new SAFEERLANG driver (`se_ssl_driver`) been implemented. When a port of this kind is opened the driver creates a special descriptor (`se_ssl_descriptor`) that keeps some information about the connection. The descriptor has:

- a reference to a session dependent structure, `session`, that contains information about the ongoing session.
- a state that either is `SE_STATE_PRE_HANDSHAKE`, `SE_STATE_HANDSHAKE` or `SE_STATE_CONNECTED`.
- a reference to the proper handshake function, which is different for the client and the server side.
- a reply that is delivered by the port when the handshake is finished.

The descriptor can be in one of three states, see above. It is changed according to in what stage the synchronization between the ERLANG processes and the inet driver has advanced.

In the ERLANG code a `se_ssl` port is chosen when a secure socket connection is desired. Furthermore is the protocol between the connected process (in the ERLANG code) and the driver extended with the messages according to figures 13 A and 13 B. The purpose of this is to synchronize the higher level handshaking actions in the processes with the actions in the driver.

When a TCP channel is set up different events are taken place on both of the nodes. On the initiating node, node A, the protocol between the driver and the connected process is showed, in figure 13 A, in the context of the handshake protocol performed on top of the TCP channel. Action 4 in figure 12 represents the messages *start handshake* and *handshake finished* in figure 13 A. The start handshake message tells the driver to perform a handshake with node B. All messages sent between the nodes in the handshaking are performed in a non-blocking mode. The handshake finished message reports when the handshake is finished.

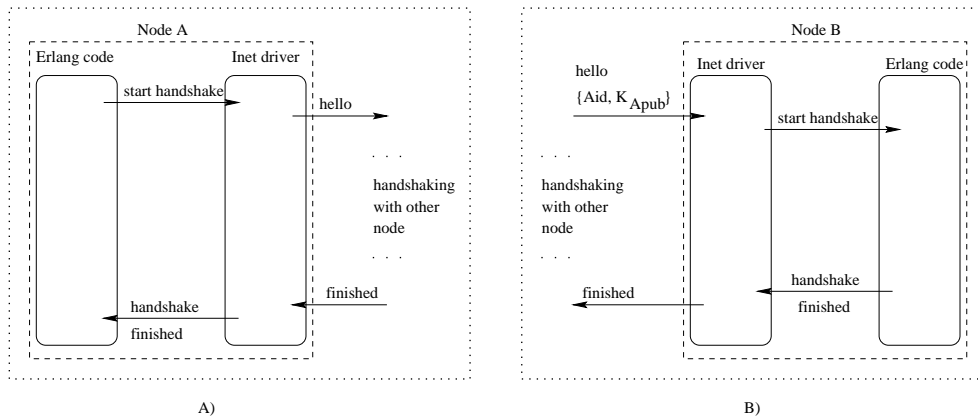


Figure 13: Protocol between ERLANG code and the Internet driver in the nodes involved in the handshake.

The *start handshake* message in figure 13 B synchronizes the process P, that administrates the handshake, with the reception of the node name and the public key of the other entity. When this happens P fetches the node identifier for the other node, by invoking the BIF `fetch_nid/1`. This information is necessary for the `net_kernel` and the process that performs the handshaking.

6.5 Performance

Late in this thesis project this system could manage to establish an encrypted connection. There were, and still are, some problems with the integration of the protocol implementation into the ERLANG run-time system. The system is not stable and when large messages are sent the system will hang. Those faults are not yet localized. The performance tests presented in this section are not complete and should be extended to do a full evaluation.

The SSLeay package includes a test program *speed* that gives performance figures for the different encryption algorithms as they are in the compiled system on the actual computer running the test. The program measures the amount of data processed per second. In all tests presented in this section the system used DES in CBC mode and SHA1 (Secure Hash Algorithm) for encryption. The results of speed for DES and SHA1 are showed in table 2.

size of data [byte]	8	64	256	1024	8192
SHA1 [kb/sec]	303.68	1342.92	2280.48	2758.94	2893.58
DES [kb/sec]	696.13	758.84	766.54	766.27	753.32

Table 2: Results when running “speed” on the computer used for the testing. The figures shows kbytes data processed per second, given a specific data size as input.

Tests on the SAFEERLANG system have been done with some test programs. The program in appendix B sends a message and waits for the return of the message from the other node before it sends the next message. This implies that the message is first encrypted by the sender then decrypted by the receiver. Thereafter encrypted and decrypted the other way around. Data that is sent/received is processed totally four times through encryption/decryption algorithms. The transmission time between the sender and receiver is supposed to be equal whether encryption is used or not. Both DES and SHA1 are applied on the data, so the time for both algorithms have been added¹⁷. That makes the results of table 3.

size of data [byte]	8	64	256	1024	8192	16384	32768
time [millisecond]	0.151	0.53	1.78	6.84	54.8	120	200

Table 3: Time consumed by the SSLey encryption/decryption algorithms when processed data has specified size and is processed four times.

In figure 14 is the SAFEERLANG system with encryption compared to the ERLANG R6B system without encryption. The time axis shows the time consumed for a mes-

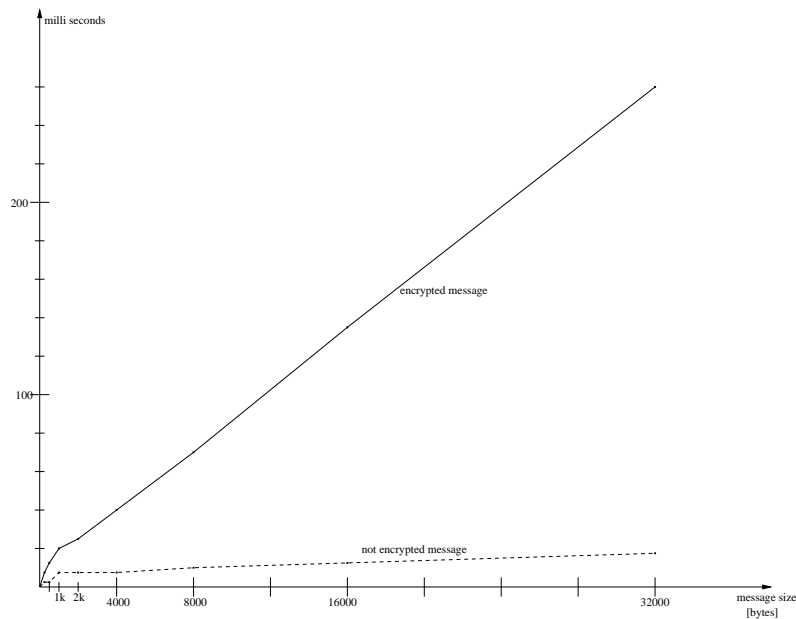


Figure 14: The broken line shows the performance (time/message) of the original ERLANG system and the other line represents the SAFEERLANG system with encryption.

¹⁷The figures are calculated by the formula: $A * (\text{Size of data chunks}) / (\text{Total amount of data processed during time } A)$. Where A is the time for data to flow through both DES and SHA1.

sage to be sent and resent between two nodes. The horizontal axis shows the message size. The two different nodes were running on the same computer. Thus the transmission time between the nodes was as fast as possible. Normally when two nodes are running on different hosts the transmission time will reduce the difference between the two systems considerably.

When the total measured time for the SAFEERLANG system is compared to the time of just the encryption/decryption processing we can see that the encryption/decryption takes about 35 % of the time with 1 kb data size, 78 % with 8 kb data size and 89 % with 16 kb.

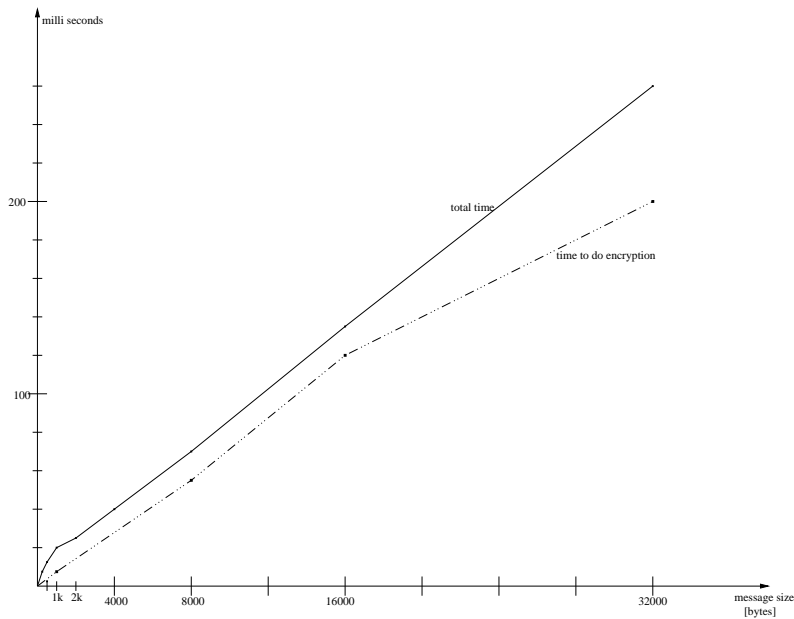


Figure 15: The gap between the curves shows the overhead from the implementation.

Comparing the new implementation, when the time for encryption/decryption is uncounted, with the ERLANG R6B system we will get the overhead of the new implementation. This comparison is the gap between the curves in figure 15 versus the lower curve in figure 14. The result is shown in table 4.

This result indicates that the implementation done in this thesis probably can be improved rather much. However, the result shows that security and particularly encryption has a high performance cost.

Data size [bytes]	8	64	256	1024	8192	16384	32768
SAFEERLANG [millisec]	11.71	8.61	7.04	12.59	15.34	14.76	61.70
ERLANG R6B[millisec]	2.67	4.13	3.69	6.69	9.00	11.32	17.17
Difference [%]	77	52	48	47	41	23	72
Difference [millisec]	9.04	4.48	3.35	5.90	6.34	3.44	44.53

Table 4: Difference between (measured time for SAFEERLANG - encryption/decryption processing) and measured time for ERLANG R6B.

10th October 2000

7 Further Work

7.1 Improvements of the implementation

At the moment of this writing neither of the implementations have been finished. Partly due to that a real system cannot be implemented fully until this implementation has been merged with [12], and partly to some faults that have appeared in the integration of the protocol implementation into the ERLANG run-time system. Furthermore there are some work to do in the protocol as well.

7.1.1 The protocol

This part of the work needs more testing to discover weak points. The failure handling in the protocol needs to be extended so that unexpected events would be taken care of properly.

One question is how a re-establishment of a previous connection should be set up. If one wants to use the previously negotiated session attributes (i.e. algorithms, secret, etc.), then the old `session` structure would be used when the run-time system starts the handshake. This has been considered in the implementation of the protocol, but there is no support in the run-time system for this.

7.1.2 Integration of secure channels in SAFEERLANG

Today there is a faked implementation of the key propagation to the point where the handshake is started. When this system is integrated with [12] there will be true capabilities in the system that propagates public keys used in the handshaking. Those changes are rather simple to do and are probably minor concerns in the integration work.

Functionality could be implemented that takes care of the choice whether old or new session attributes would be used when a connection is reestablished. Today is always new session attributes negotiated when connecting a remote node.

7.2 Other future tasks

During this thesis some tasks close related to distributed communication have been considered. A shallow feasibility study has been done on the following topics.

7.2.1 Several simultaneous connections

Today is the ERLANG system strongly limited in the possible number of simultaneously open TCP connections. The total number of file descriptors in the system sets the limit. When a connection attempt is done in the present system and all descriptors are occupied the attempt will be denied. In SAFEERLANG we want to be able to have a great number of simultaneous connections.

One solution could be to represent each connection with a *virtual descriptor* until the connection is explicitly shut down, and a *live descriptor* that represents a TCP

connection. When the system is out of file descriptors and there is a new attempt to establish a connection, one of the existing TCP connections, for instance connection A, must be shut down and the corresponding live descriptor is discarded. However, the virtual descriptor is saved and represents that connection. For the connection attempt a new virtual descriptor is created, and also a live descriptor when the TCP connection, connection B, is established. Later on when the virtual descriptor, representing connection A, again is used a new TCP connection must be set up.

One problem that must be solved is the way a connection is shut down. There should be some kind of agreement protocol between the two nodes. Otherwise could strange situations come up. For instance if node A and B are connected, and A decides to shut down the connection. Meanwhile a process on B sends a message that disappears. Later on the connection is established and again the process on B sends a message that arrives properly. Now we have got a situation where the first message have not arrived but the second has. We don't want this strange semantics.

Another problem that comes up when one temporary shuts down a connection is how to manage the *links*¹⁸ to objects on other nodes. If a connection goes down in ERLANG today, all links that are setup between processes on both sides of the channel will expire, and thus force the linked processes to terminate or to handle the expired link. This functionality must still exist but a temporary disconnected channel should not cause the links to expire. One solution could be, if links are present on a temporary disconnected channel, to reestablish the connection just to poll the linked processes.

7.2.2 A key - node cache

If the run-time system sends a node name and a public key explicitly with each external sent capability, there would be a great amount of unnecessary data sent. Each key is of the size 128 - 254 bytes and each node name is potentially very big. All resource identifiers sent from a node have the same pair of node-name and public key. Thus it is only necessary to send a resource identifier including the pair the first time an identifier is sent and thereafter send a number representing that pair. This kind of cache is today used for atoms in ERLANG.

For each communication channel a node has a cache, which consists of an array for incoming pairs and another array for outgoing pairs. The array for incoming pairs corresponds to the array for outgoing pairs on the other node. Each field in the array is a pointer to a `KeyNode` structure. This structure is part of a capability and contains information about a node. Instead of sending a key - node pair the array index is sent.

A design decision, when one also has the feature described in section 7.2.1, is how long the entries in the cache should be left. One approach is to empty the cache when a TCP connection is shut down and the live descriptor is discarded. This would bring about that each time a connection is reestablished the caches on each side must be initialized again.

¹⁸In ERLANG processes can be linked to other objects (processes and ports). Then they are noticed if the linked object terminates.

Another approach is to save the cache entries as long as the virtual descriptor lasts. This solution saves overhead when a connection is reestablished. A serious drawback is that it forces the system to keep data structures that may have been garbage collected.

10th October 2000

8 Conclusion

One learning from this work is that it is not that easy to do changes in big existing systems. One has to consider all side effects of a change in the system. However the changes in the handshake protocol implementation seems to be successful. Even if the SSLey system is rather big it is easy to understand how it works. And changes on the level we have done in this work is not that difficult to do.

If someone considers to introduce secure communication in a network layer on top of the TCP layer, the scheme we have followed in this thesis could be recommended. The alternative is to implement a secure communication protocol of your own, which probably would be a much harder task.

Though there are still some bugs left in the integration of the ERLANG run-time system and the handshake functionality, this seems to be successful as well. This system is much more complex than the SSLey implementation, which is the main reason why the integration part of the work have been the most difficult to do.

10th October 2000

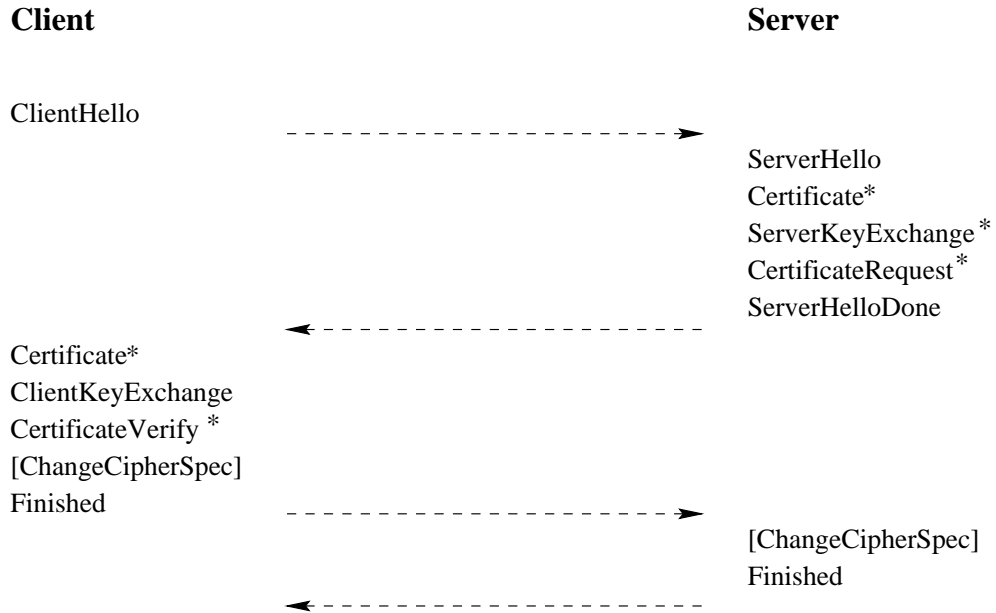
References

- [1] Anderson, R. 1993. *Why Cryptosystems Fail*; available from <http://www.cl.cam.ac.uk/users/rja14/>; Internet; accessed 26 October 1999.
- [2] Armstrong, J., Viriding, R., Wikström, C. and Williams, M. 1996. *Concurrent programming in ERLANG*. ISBN 0-13-508301-X, London: Prentice Hall Europe.
- [3] Arsenault, A. and Turner, S. 1999. *Internet X.509 Public Key Infrastructure PKIX Roadmap*; available from <http://www.ietf.org/internet-drafts/draft-ietf-pkix-roadmap-03.txt>; Internet; accessed 15 October 1999.
- [4] Branchaud, M. 1997. *A SURVEY OF PUBLIC - KEY INFRASTRUCTURES*; available from <http://home.xcert.com/~marcnarc//PKI/thesis/>; Internet; accessed 18 Jan 2000.
- [5] Brown, L. 1997. *SSErl - Prototype of a Safer ERLANG*; available from <http://www.adfa.edu.au/~lpb/papers/tr9704.html>; Internet; accessed 18 April 2000.
- [6] Brown, L. and Sahlin, D. 1999. *EXTENDING ERLANG FOR SAFE MOBILE CODE EXECUTION*; available from <http://www.adfa.edu.au/~lpb/research/sserl/icics99.html>; Internet; accessed 19 April 2000.
- [7] Chow, R. and Johnson, T. 1997. *Distributed Operating Systems & Algorithms*. ISBN 0-201-49838-3, Reading, Massachusetts: Addison-Wesley
- [8] Erlang 4.4 Extensions. 1997; available from <http://www.erlang.org/doc.html>; Internet; accessed 18 April 2000.
- [9] Freier, A. O., Karlton, P., Kocher, P. C. 1996. *The SSL Protocol Version 3.0*; available from <http://www.netscape.com/eng/ssl3/draft302.txt>; Internet; accessed 19 Dec 1999.
- [10] Gong, L. 1997. *Enclaves: Enabling Secure Collaboration over the Internet*; available from <http://java.sun.com/people/gong/papers/pubs97.html>; Internet; accessed 13 April 2000.
- [11] Gong, L. 1998. *Secure Java Class Loading*; available from <http://java.sun.com/people/gong/java/security.html>; Internet; accessed 18 October 1999.
- [12] Green, R. 2000. *Enhancing Security in Distributed ERLANG by Integrating Access Control, Approaching a Real SAFEERLANG Implementation*. Not yet published. Will be published at: <http://www.erlang.se/publications>.

10th October 2000

- [13] Hudson, T. J., Young, E. A. 1998. *SSLeay and SSLapps FAQ*; available from <http://www2.psy.uq.edu.au/~ftp/Crypto/>>; Internet; accessed 9 May 2000.
- [14] Hudson, T. J., Young, E. A. *SSLeay Programmer Reference*; available from <http://www2.psy.uq.edu.au/~ftp/Crypto/ssl.html#HDR0>>; Internet; accessed 9 May 2000.
- [15] Naeser, G. 1997. *SafeErlang*. ISSN 1100-1836, Uppsala University; available from <http://www.docs.uu.se/~gaffe/themes/tmp/publications.shtml>>; Internet; accessed 28 April 2000.
- [16] Pfleeger, C., P. 1997. *Security in Computing*. ISBN 0-13-185794-0, New York: Prentice Hall.
- [17] Schneier, B. 1996. *Applied Cryptography*. ISBN 0-471-12845-7, New York: John Wiley & Sons, Inc.
- [18] Smith, R., E. 1997. *Internet Cryptography*. ISBN 0-201-92480-3, Reading, Massachusetts: Addison-Wesley.
- [19] Stallings, W. 1997. *Data and Computer Communications*. ISBN 0-13-571274-2, New Jersey: Prentice-Hall.
- [20] Wagner, D. and Schneier B. 1996. *Analysis of the SSL 3.0 protocol*; available from <http://www.counterpane.com/ssl.html>>; Internet; accessed 8 May 2000.

A The SSL version 3.0 handshake protocol



* Indicate optional or situation dependent messages that are not always sent.

10th October 2000

B Test programs with synchronous send/receive

```
%%% File : receive_send.erl
%%% Author : Bertil Karlsson <bertil@erix.ericsson.se>
%%% Purpose : test of encryption in SafeErlang
%%% Created : 4 Oct 2000 by Bertil Karlsson

-module(receive_send).
-author('bertil@erix.ericsson.se').

-export([start/0, init/0, loop/0]).

% spawned on node 'a@punsch.du.uab.ericsson.se'
start() ->
    register(receive_send,spawn(receive_send, init, [])).

init() ->
    receive
        {From, Message} ->
            From ! {self(), Message}
    end,
    loop().

loop() ->
    receive
        {From, Message} ->
            From ! {self(), Message},
            loop()
    end.
```

10th October 2000

```
%%% File : send_receive.erl
%%% Author : Bertil Karlsson <bertil@erix.ericsson.se>
%%% Purpose : test of encryption in SafeErlang
%%% Created : 4 Oct 2000 by Bertil Karlsson

-module(send_receive).
-author('bertil@erix.ericsson.se').

-export([start/0, send/1, init/0, loop/4, finish/0]).

start() ->
    spawn(send_receive, init, []).

init() ->
    io:format("~n***** Test started *****~n~n"),
    StartTime = micro_sec(now()),
    send('connect'),
    receive
        {From, connect} ->
            Now = micro_sec(now()),
            Time = Now - StartTime,
            io:format("Connected to: ~w~nElapsed time:~w~n",
                [From, Time]),
            M = list_to_binary("abcd"),
            NewTime = now(),
            loop(From, M, NewTime, 100)
    after 2000 ->
        io:format("Connection timeout!~n")
    end.

loop(To, Message, Time, N) when size(Message) > 1048600 ->
    io:format("~n***** Test finished *****~n");

loop(To, Message, Time, N) ->
    sendloop(To, Message, N),
    compare_and_print(Message, N, Time),
    NextMsg = expand_msg(Message),
    NewTime = now(),
    loop(To, NextMsg, NewTime, N).

sendloop(To, Msg, 0) ->
    true;
```

10th October 2000

```
sendloop(To, Msg, Num) ->
  To ! {self(), Msg},
  receive
    {To, _} ->
      true
  end,
  sendloop(To, Msg, Num-1).

compare_and_print(Message, N, Time) ->
  Now = now(),
  MsgSize = size(Message),
  StartTime = micro_sec(Time),
  StopTime = micro_sec(Now),
  ElapsedTime = StopTime - StartTime,
  TimePerMsg = ElapsedTime / N,
  io:format("Number of messages: ~w ~nMessage size:"
           "~w bytes ~nElapsed time: ~w microseconds."
           "~nTime per message: ~w microseconds.~n",
           [N, MsgSize, ElapsedTime, TimePerMsg]).

expand_msg(Msg) ->
  list_to_binary([Msg, Msg]).

micro_sec({T1, T2, T3}) ->
  Time = T3 + (T2 * 1000000) + (T1 * 1000000000000),
  Time.

send(Message) ->
  {receive_send,
   'a@punsch.du.uab.ericsson.se'} ! {self(), Message}.

finish() ->
  {receive_send,
   'a@punsch.du.uab.ericsson.se'} ! {self(), 'END'},
  io:format("Finished.~n").
```

10th October 2000

Index

- access right, 16
- access rights, 24, 28
- adjudicable protocol, 33
- arbitrated protocol, 33
- asymmetric encryption, 12
- authentication, 25, 30, 31, 33–36, 38
- availability, 16

- BIF, 22
- BIO, 47
- brute force, 17, 35, 41, 42

- CA, 15
- capability, 28, 29, 35, 55, 56
- certificate, 15, 47
- Certificate Revocation List, 15
- Certification Authority, 15
- cipher-block chaining, 42
- cipher-text, 11
- code loading, 24, 28
- compound data type, 21
- compression, 36, 43
- Computer Science Laboratory, 9, 21
- confidentiality, 16
- confusion, 12
- connected process, 23, 48, 49
- constant data type, 21
- cookie, 25, 30, 37, 48
- CRL, 15
- CSLAB, 9, 27

- denial of service, 16
- diffusion, 12
- DOS, 16, 41, 46

- eavesdropping, 17, 42
- Enclaves, 12
- epmd, 46, 48
- Erlang, 9, 21

- fabrication, 17
- forgery attack, 18
- fragmentation, 43

- handshake function, 49
- handshake protocol, 33, 35
- hash function, 15, 42
- hash value, 15

- inet driver, 24, 46, 48, 49
- initialization vector, 42
- integrity, 16
- interception, 17
- Internet driver, 24, 48
- interruption, 17

- key distribution, 14, 29, 35, 47
- key management, 16
- key-node cache, 56
- known-plain text attack, 42
- KTH, 9

- links, 56
- live descriptor, 55

- MAC, 30, 42
- man in the middle attack, 19
- masquerade attack, 18
- memory management, 24
- Message Authentication Code, 42
- message digest, 16, 38, 39
- Mid, 29
- MIM attack, 19, 35, 38
- modification, 17
- module identifier, 29

- net_kernel, 25, 46, 48
- node, 9, 23
- node identifier, 29, 50
- node name, 36, 46, 50, 56
- non-blocking I/O, 24, 47
- nonce, 34, 38–40

- OpenSSL, 46
- OTP, 9

- padding, 42, 44

pattern matching, 21
Pid, 22, 23
PKI, 15
plain-text, 11
port, 23
port identifier, 29
PoSE, 27
private key, 13, 35, 37
private-key encryption, 12
process, 22
process identifier, 23, 29, 35
public key, 13, 14, 28, 29, 34–36, 50,
55, 56
Public Key Infrastructure, 15
public-key encryption, 12

replay attack, 18, 19, 35, 40
resource identifier, 29, 44, 56
resource management, 24, 27
RSA algorithm, 13

SafeErlang, 9, 16, 27
SafeErlang driver, 49
scheduling, 23
se_ssl_descriptor, 49
se_ssl_driver, 49
secret-sharing, 34
secure protocol, 33
self-enforcing protocol, 33, 35
sequence number, 42
session protocol, 33, 41
simultaneous connections, 55
SSErl, 27
SSL, 41
SSL handshake protocol, 42
SSL record protocol, 41
SSLeay, 46, 59
SSLeay-0.9.0b, 46
sub-node, 27
switching, 18
symmetric encryption, 12

time-stamp, 14, 34
TLS, 47
traffic analysis, 17, 19, 42

trust, 14, 29
virtual descriptor, 55