Master's Thesis in
Computer Science

27th June 2000

Enhancing Security in Distributed ERLANG by Integrating Access Control

Approaching a Real SAFEERLANG Implementation

by

Rickard Green

The Royal Institute of Technology
Kungliga Tekniska Högskolan

Examiner:	Prof. Seif Haridi, Ph. D., Department of Teleinformatics, Royal Institute of Technology
Supervisor:	Per Brand, M. Sc., Department of Teleinformatics, Royal Institute of Technology
Supervisor:	Dan Sahlin, Ph. D., Computer Science Laboratory, Ericsson Utvecklings AB

Title:	Enhancing Security in Distributed ERLANG by Integrating Access Control
Sub-title:	Approaching a Real SAFEERLANG Implementation
	Master's thesis. Part of requirements for a Master's degree in computer science at the Royal Institute of Technology.
	Examensarbete 2G1015, 20 poäng. Delmoment i examination för civilingenjörsexamen i datateknik vid Kungliga Tekniska Högskolan.
Available at:	http://www.ericsson.se/cslab/~rickard/thesis
Author:	*Rickard Green*
E-mail at Ericsson:	rickard.green@uab.ericsson.se
E-mail at KTH:	d95-rgr@d.kth.se
E-mail at home:	rickard.green@home.se

Abstract

As the need for interaction between software components in unprotected open networks, such as the Internet, increases, the need for programming languages suited for development of secure distributed software components increases.

Erlang is a programming language well suited for development of distributed software components. The lack of security mechanisms, such as access control and secure communication, makes Erlang less attractive for development of software components that are to be used in open network environments.

This report addresses the problem of how to integrate access control mechanisms in Erlang without changing the semantics of the language too much and without too much performance penalty.

Contents

1 Introduction

Security issues are becoming more and more important when developing software today. This due to the fact that software is becoming more and more exposed to security threats. This is because software is being used in environments which are very open, such as the Internet.

SafeErlang is an enhancement of the programming language Erlang, which addresses these security issues. The intent with SafeErlang is to offer a programming language in which distributed software components with heavy security requirements can be developed easily.

1.1 Previous Work on SafeErlang

The idea to extend Erlang with security mechanisms was brought up by Dan Sahlin and Gustaf Naeser. This was during Gustaf Naeser's master's thesis work in 1996, at Computer Science Laboratory at Ericsson Utvecklings AB, with Dan Sahlin as supervisor. They examined how Erlang would have to be modified to support mobile agents operating on Erlang systems. When having mobile agents operating on a set of systems, the security issue becomes very apparent. Mobile agents are in essence programs that automatically migrate between different host systems. If one is going to have potentially malicious programs entering one's system, one wants to be able to constraint the programs' possibility to commit hostile acts. Gustaf Naeser's master's thesis work resulted in a prototype of a safer Erlang system [N97]. This prototype was named SafeErlang.

In 1997 Dr. Lawrie Brown, at the Australian Defence Force Academy, got into contact with SafeErlang and continued the work on SafeErlang during his sabbatical. He modified and extended the SafeErlang prototype that Gustaf Naeser had implemented in 1996. The prototype Lawrie Brown implemented was named SSErl [B97]. Lawrie Brown had close contact with Dan Sahlin during this work.

In 1998 Otto Björkström also did his master's thesis, at Computer Science Laboratory at Ericsson Utvecklings AB, with Dan Sahlin as supervisor. This work wasn't on further development of the prototypes, but instead on programming in the SafeErlang environment. Was SafeErlang a programming language in which one could create secure distributed programs? The work resulted in a distributed game which operated on top of a set of SafeErlang systems.

In 1999 I and Bertil Karlsson began working on our master's theses with the goal to produce a SafeErlang system integrated in Erlang's run-time system, i.e., taking SafeErlang from a prototype to a real system. We also worked at Computer Science Laboratory at Ericsson Utvecklings AB. At Ericsson we had Dan Sahlin as supervisor and at the Royal Institute of Technology, KTH, we had Per Brand as supervisor. Bertil and I worked on different parts of SafeErlang and this report is the result of my work on SafeErlang. Bertil's work on SafeErlang is presented in [K00].

1.2 Some Clarifications

I am in this report going to refer to Gustaf Naeser's prototype of SAFEERLANG as PoSE[1] and I will use the term SAFEERLANG either when I refer to the idea of a safer ERLANG or when I refer to my and Bertil's implementation of SAFEERLANG. This so that there will not be any confusion about which SAFEERLANG implementation I am talking about.

When I write “we”, “our”, “us”, etc, I mean me and my supervisors Dan Sahlin and Per Brand.

1.3 Acknowledgment

I would especially like to thank my supervisors Dan Sahlin and Per Brand for all their support, ideas, and help during this work. All employees at Computer Science Laboratory and at the ERTS team at Open Systems, Ericsson Utvecklings AB have also been very helpful during this work and deserve thanks for that. Matthias Läng at Computer Science Laboratory also deserves thanks for proofreading my report. Finally, I would like to thank Bertil Karlsson, who also has been working on SAFEERLANG, and Erik Klintskog, at SICS[2], for fruitful discussions regarding SAFEERLANG.

[1]**P**rototype **o**f **S**AFE**E**RLANG.

[2]Swedish Institute of Computer Science

2 What is ERLANG?

ERLANG is a functional programming language, which provides support for explicit concurrency, developed at Computer Science Laboratory (CSLab) at Ellemtel, now known as Ericsson Utvecklings AB. The goal with the development of ERLANG was to create a programming language well suited for telecom applications.

One had previously looked at symbolic languages, such as Lisp and Prolog, at CSLab. Implementations in such languages require a very small amount of source code which shortens development times. However, the need for concurrency and simple error-recovery, which are very important concepts in telecom applications, were not well supported by these languages. When the decision to create a new programming language was made, the focus was on symbolic programming, error-recovery, concurrency, and real-time. The resulting programming language was ERLANG.

I will in this section present an overview of the most important concepts of ERLANG. For a more detailed description of ERLANG see [AVWW96, EDE97].

2.1 Data Types

ERLANG is a dynamically typed language; all data always have a type that is determined at run-time. The data types of ERLANG, also called ERLANG terms, are listed in table 1.

Constant (primitive) data types:		Examples
`integer`		`3`, `-5`
`float`		`3.6`, `23.2e-05`
`atom`	A textual atomic value.	`true`, `'An atom'`
`pid`	An identifier of an ERLANG process.	`<0.3.0>`
`port`	An identifier of an ERLANG port.	`<0,17>`
`reference`	A globally unique value.	`#Ref<0.0.0.318>`
`fun`	An higher order function.	`#Fun<my_module.2.35544541>`
`binary`	Arbitrary “raw” data.	`#Bin<24>`
Compound data types:		
`list`	A linked list.	`[true, 3, <0.3.0>]`
`tuple`	A fixed size compound data type.	`{false, 6}`, `{2, 6, 3}`
`record`	A modified tuple.	`{my_record, 14, hello}`

Table 1: ERLANG data types.

An ERLANG list is an ordinary linked list which can be manipulated as expected. A new list can be formed by adding an element to the front of an old list. The head element of a list can be dropped so that one gets the tail of the list, and so on.

Tuple is another compound data type which is intended for storing a fixed number of elements. There are no operations corresponding to those of the lists.

The record data type isn't really a separate data type; it's actually a tuple with a description of its interpretation so that one can access its elements by name, instead of by the elements' location in the tuple.

The ERLANG data type `reference` is a little peculiar in that its instances don't actually refer to anything. A reference is just a unique value which usually is used to "tag" compound data structures.

2.2 Pattern Matching

Variables in ERLANG are so-called "one time assignment" variables which are either bound or unbound. A variable is unbound until it has become bound to a value. Once it has become bound to a value, it can never be bound to another value. A variable is written as a text string beginning with an uppercase letter.

Variables get bound by pattern matching against ERLANG terms. An example:

```
...
TwoTuple = {one, two},
{FirstElement, _} = TwoTuple,
...
```

In this example the variable `TwoTuple` gets bound to a tuple of two elements and the variable `FirstElement` gets bound to the atom `one`. The underscore is an anonymous variable that can match anything.

Pattern matching is also used when a function call is performed. If the function call `is_two_tuple(TwoTuple)` is performed, the first function clause of the two function clauses below will match, and `true` will be returned. Function clauses are always tested from the top downwards and the first matching clause will be chosen.

```
is_two_tuple({_,_}) ->
   true;
is_two_tuple(_) ->
   false.
```

Pattern matching is also used in the `case` and `receive` expressions. The `receive` expression will be explained later.

2.3 Loops

ERLANG does not have loop constructs like the well known `for` and `while` loops which are typical for imperative languages. Loop constructs like the `for` and `while` loops are based on a loop variable, typically an integer variable that is incremented, together with some kind of condition for loop termination, typically a limit for the loop variable. In ERLANG, there exist no variables that can be assigned a value and then reassigned another value which makes it impossible to construct these kinds of loops.

Loop constructs in ERLANG are based on tail recursive calls. Normally a recursive call to a function means that a new activation record has to be allocated on the stack. This is not the case with tail recursive function calls. When a tail recursive function call is performed, the compiler or interpreter can reuse the present activation record for the new tail recursive call. This is called *last call optimization* and *has* to be performed by compilers and interpreters of languages like ERLANG. An example of the usage of tail recursion is presented below. The function `length/2`[3] uses tail recursion to count the number of elements in a list.

```
length(List) ->
   length(List, 0).

length([_|Rest], N) ->
   length(Rest, N+1);
length([], N) ->
   N.
```

`length/2` is only intended to be called by `length/1` which is intended to be called by other functions.

2.4 Concurrency

Concurrency is explicitly expressed by creation of ERLANG processes and communication between ERLANG processes.

A process is created by calling the built-in function[4] `spawn/3` with a module name, a function name, and an argument list as arguments. The `spawn/3` function will return a process identifier that later can be used for sending messages to the newly created process.

Communication between ERLANG processes is performed by asynchronous message passing. A message is sent by using the binary infix operator "`!`" as shown below. `Pid` is the process identifier of the process to which the message will be sent, and `Message` is an arbitrary ERLANG term which will be sent as a message.

```
...
Pid ! Message,
...
```

Every process has a message queue in which messages are stored until the process explicitly fetches them. A message can be fetched from the message queue of a process when the process reaches a `receive` expression. A message is fetched from the message queue by pattern matching. First the first message in the queue is matched against the patterns in the `receive` expression, from the topmost pattern downwards. If the first message didn't match any of the patterns, the next message will be tried in the same manner. This will continue until there are no more messages to try or until a message that matches a pattern is found. All messages, except for a message that is fetched by a match, will be left unaffected in the message queue. If no message in the queue matches any of the patterns in the `receive` expression, the process will be blocked until a message that matches one of the patterns is received. If the `receive`

[3]The slash and number after the function name, express the arity of the function, i.e. the number of arguments that the function takes.

[4]Built-in functions are, in the ERLANG community, known as BIFs.

expression contains an after expression, the process may continue even though no messages have matched after the time specified in the after expression.

An example:

```
remove_two_tuple() ->
   receive
      {special, _} ->
         special_two_tuple;
      {_, _} ->
         two_tuple
   after 1000 ->
      timeout
   end.
```

A process that calls the function remove_two_tuple/0 will get either the atom special_two_tuple, the atom two_tuple, or the atom timeout as return value. The atom special_two_tuple will be returned if the first tuple of size two that is found has the first element set to the atom special. If the first tuple of size two that is found doesn't have the first element set to the atom special, the atom two_tuple will be returned. If not tuples of size two are found in the message queue and no such messages are received within 1000 milliseconds, the call to remove_two_tuple/0 will return the atom timeout.

2.5 Error Handling

If a process executes code that causes a run-time failure, the process will terminate abnormally. This may not always be the desired behavior. In order to be able to modify this behavior, ERLANG has a catch expression which in essence catches the failure so that the program can inspect the failure and take appropriate actions. If a failure occurs, the call stack will be rewound and the first catch expression that is found will produce an ERLANG term representing the failure. If no catch expression is found, the process will terminate abnormally.

Failures can also be induced by ERLANG's throw expression. By executing the expression throw(MyFailure), the ERLANG term MyFailure will be handled by the system as if it was a run-time failure. Of course, MyFailure can be caught by a catch expression.

Another feature of ERLANG that is useful is the link mechanism. If one process terminates, in an application consisting of multiple processes, the application may not work correctly. By linking ERLANG processes to each other, one can ensure that all linked processes will terminate if one of them terminates. This ensures that an application that fails will be completely removed from the system. This behavior is desirable, but one may also want to restart the application. This can be done by running one process as supervisor. The supervisor process is set to "trap exits" and is then linked to all the other processes in the application. When a process terminates in the application, the supervisor will not be terminated; instead, it will be informed by a message that a process which it was linked to has terminated. When the supervisor process receives such a message, it can take appropriate actions, for example restart the application. This is a very useful feature of Erlang which makes it easier to implement fault tolerant applications.

A link between two processes is symmetric or "two-way", that is, if one of the processes terminates, both of them terminate. Both of them can of course also "trap exits". The monitor mechanism provides a way to do almost the same thing but only "one-way". A process that monitors another process will receive a message, informing that the monitored process has terminated, when the monitored process terminates. The monitored process is "unaware" of the process that monitors it and will not receive any "termination message" if the process that monitors it terminates.

2.6 Real-time

An Erlang system is required to have soft real-time properties. Real-time properties are mainly an implementation issue. On the design level of the language it's mainly a question of not introducing language constructs which make it impossible to implement a system which fulfills the desired real-time properties. Because of the implementation dependent nature of real-time properties, I will not discuss this further in this section. In section 7.5 some examples will be given of implementation constructs in the R6B-0 release which give Erlang the desired real-time properties.

2.7 Distributed Erlang

One Erlang system is called an Erlang node. On a node processes can interact as shown in the previous sections. Interaction between Erlang processes on different nodes is performed in very much the same way as if the processes were located on the same node. The most significant differences that exist are that operations which involve distribution can fail in more ways than the corresponding non-distributed operations and that Erlang port identifiers don't behave in the same way distributed as on their node of origin. Erlang port identifiers can be distributed to other nodes, but they can only be used locally on the node where the actual port exists. This is one of only a few differences that exists.

In order to identify different Erlang nodes, node identifying atoms are used. A node identifying atom is an atom of the form "`nodename@host`". Node identifying atoms are used in operations where one needs to identify a specific node that the operation should affect. For example, if one wants to spawn a process on another node, one can use the built-in function `spawn/4` with a node identifying atom as first argument.

Here are some built-in functions that use node identifying atoms:

`spawn/4` Arguments: `Node`, `Module`, `Function`, `ArgumentList`.
Spawns a process on the node identified by the atom `Node`.

`spawn_link/4` Arguments: `Node`, `Module`, `Function`, `ArgumentList`.
Spawns a process on the node identified by the atom `Node` and links to the newly created process.

`monitor_node/2` Arguments: `Node`, `Boolean`.
: Monitors the node identified by the atom `Node`. The process calling this function will be sent a `{nodedown, Node}` message if the connection to the node `Node` terminates.

`node/0`
: Returns the node identifying atom of the local node.

`node/1` Arguments: `Item`.
: Returns the node identifying atom that corresponds to `Item`'s node of origin. `Item` is either a process identifier, a port identifier, or a reference.

`nodes/0`
: Returns a list of node identifying atoms corresponding to all the nodes that are connected to the node where the call is made.

Some built-in functions for distributed as well as local interaction:

`link/1` Arguments: `Item`.
: `Item` can be either a process or a port identifier. Links the calling process together with a local or remote process or a local port.
Access transparency deviation: Failure due to linkage to a dead or non-existing process can be caught with a `catch` expression if the linkage is to local process, otherwise not. Only local ports can be linked to.

`unlink/1` Arguments: `Item`.
: Removes link between calling process and `Item`.

`monitor/2` Arguments: `Type, Item`
: `Type` can be either the atom `process` or the atom `port`. `Item` can be either a, local or remote, process or a local port identifier. Returns a reference `Ref` which later can be used to remove the monitor or to match against in the corresponding "down message" which will be received when `Item` terminates. A "down message" of the form `{'DOWN', Ref, Type, Item, Info}` message will be sent to the calling process when `Item` terminates.
Access transparency deviation: Only local ports can be monitored.

`demonitor/1` Arguments: `Ref`
: Removes the monitor corresponding to `Ref`. `Ref` is the reference returned by the corresponding `monitor/2` call.

`exit/2` Arguments: `Pid`, `Reason`
: Exits a local or a remote process, identified by `Pid`.

`!/2` Arguments: `Receiver`, `Message`
: Sends the message `Message` to `Receiver`. `Receiver` can be either a process identifier, a port identifier, an atom of a locally registered service, or a tuple consisting of an atom of a registered service and a node identifying atom. A process identifier can be either local or remote. If `Receiver` is a port identifier,

the port has to be a local port connected to the process performing the send operation.
Access transparency deviation: Only local ports can be sent messages. If `Receiver` is an atom, i.e. it is a send operation to a locally registered process, the operation will fail if there is no process registered as `Receiver`. If `Receiver` is a tuple, the operation won't fail if there is no process registered as the corresponding service, the message will just disappear.

`group_leader/2` Arguments: `LeaderPid`, `Pid`
Sets the group leader of the process referred to by `Pid` to `LeaderPid`.
Access transparency deviation: If `Pid` refers to a dead or non-existing local process a `badarg` failure will be thrown. If `Pid` refers to a remote process that is dead or non-existing this will be silently ignored.

3 Why SafeErlang?

Erlang is quite secure as it is which makes it suitable for further improvements in this area. For example, there exist no language constructs which permit the Erlang programmer to directly manipulate memory.

3.1 Access Control

Pure functional programming in Erlang is intrinsically secure as stated in [BS99]. Erlang does not only permit pure functional programming, though. There are quite a lot of operations which have side-effects. The following operations are identified in [BS99, N97]:

- Accesses to processes. This is performed by using the process identifier of a process. For example, by sending a message or linking to a process.
- Accesses to external resources. This is performed by using port identifiers to access external resources, such as files.
- Accesses to the run-time system itself. This is performed by using built-in functions such as `halt/0`, which halts the whole system, and `system_flag/2`, which sets different system properties.
- Accesses to permanent data. This is performed by using the database built-in functions, which access databases.

A process is accessed through the process identifier of the process. Anyone that has access to the process identifier of a process can access the process in any way that it can be accessed. All types of accesses from sending a message to the process, to killing the process are allowed. One can easily see the need for the ability to grant access rights in a more fine grained way.

Not only the possessor of a process identifier of a process has the ability to access the process. In Erlang a process identifier can be constructed and used by anyone. The built-in function `list_to_pid/1` takes a textual representation of a process identifier and returns a process identifier. Killing an arbitrary process is performed as follows:

```
exit(list_to_pid("<0.17.0>"), kill).
```

In order to be able to control accesses to processes, Gustaf Naeser and Dan Sahlin converted process identifiers to process identifying capabilities. This was then implemented in PoSE, see [N97]. A capability contains a number of rights which can be removed. In order for a process to access another process, a process identifying capability has to be presented to the access control manager[5]. The capability also has to contain a right corresponding to the requested access; for example, a `send` right if the requested access is sending of a message to the process. This mechanism provides the ability to only grant a subset of all possible accesses to processes less trusted.

[5]Typically the run-time system where the process which is accessed resides on.

Capabilities could still be forged, though. In order to deal with this problem, Gustaf Naeser and Dan Sahlin proposed encrypted capabilities which would make it infeasible to forge a capability. This was not implemented in PoSE, though. Lawrie Brown implemented two types of forgery protection of capabilities, HMAC-MD5 and password protection, in his SSErl implementation, see [BS99, B97].

Password protection is based on a password that is saved, together with a copy of the issued capability, by the access control manager. The password is also attached to the capability when it is distributed. When the capability is used in an access request, the capability together with the attached password are compared with the previously saved capability and password. If they match, the access request will be served; otherwise, it will be rejected.

The HMAC-MD5 protection is based on a hash-value attached to the capability. This hash-value is computed by a one-way hash-function over the capability and a secret key. HMAC protection will be described in more detail in section 8.5.

3.2 Excessive System Utilization

An Erlang process may consume any amount of memory and may spawn any number of processes. An Erlang node could easily be crashed by overconsumption of memory by a malicious process. A malicious process could also seize most of the computing power by spawning a lot of processes.

The solution to these problems that Gustaf Naeser and Dan Sahlin came up with and implemented in PoSE, was the concept of sub-nodes. Sub-nodes are hierarchically ordered as a tree. Inside a sub-node, processes execute viewing the sub-node as an ordinary node. On a sub-node, limits on memory consumption and CPU-usage can be set. The limits are enforced for a sub-node and all of its sub-node descendants. If any limit for a sub-node is exceeded, the sub-node and all of its descendants will crash, but the rest of the system will remain unaffected.

3.3 Remote Code Loading

In Erlang, code is automatically loaded locally from the node where the process which is going to execute the code, exists. Code is identified by a module name which means that code identified as the same code can be different code on different nodes.

In the beginning the work on SafeErlang was concentrated on mobile agents. In order for a mobile agent to operate on a system of SafeErlang nodes, the code which the agent executes needs to be remotely loaded. When an agent migrates from one node to another, it begins executing in a new process on the node which it migrates to. In order for the agent to operate in a predictable way, the agent's code must be loaded from the agent's node of origin.

The code loader can easily be modified to load code from other nodes instead of loading code locally; however, this introduces a new security threat which has to be dealt with. If one loads the code as byte code, one has to verify that the code adheres to a predetermined set of rules. This is how this security threat is dealt with in Java. Another way of dealing with this security threat is to load the code as source code and then compile it to byte code locally by a trusted compiler on the node which the code is going to be executed on. This was the approach which Gustaf Naeser and Dan Sahlin choose to implement[6] in PoSE [N97].

By loading the code as byte code better performance can be achieved. It is quite a lot of work to write a byte code verifier which detects all malicious byte code; therefore, the byte code verifier was left for future improvements of SafeErlang.

3.4 Secure Communication

When processes located on different nodes in an open network, such as the Internet, communicate with each other, their communication is exposed to eavesdropping. To be able to supply the SafeErlang processes with confidentiality, the communication between different SafeErlang systems needs to be encrypted.

Communication could be encrypted on a "per process pair" basis, but no security would be gained by that. Even if every process encrypted all messages itself, the run-time system would have access to the communication anyway. This would only result in a more complex communication without any security gain. A process must, in other words, trust the run-time system on which it executes not to reveal its communication or any of its data to other parties.

To be able to encrypt communication, encryption keys have to be distributed. The solution Dan Sahlin came up with was to attach a public key corresponding to the issuing node to every capability. This makes it possible to use entity identifiers as before with the additional feature that encryption keys automatically get distributed.

A related problem is authentication of entity identifiers, for example, a process identifier. How can the possessor of a process identifier be sure that the process identifier really refers to the process that the possessor thinks that it refers to? This is left for the SafeErlang programmer to solve. For some applications, an authentication server implemented in SafeErlang is appropriate; for other applications, other solutions may be more appropriate.

The need for secure communication was recognized in [N97, B97], but secure communication was not implemented in PoSE or SSErl; instead, it was left aside for future work.

3.5 Security Problems in the SafeErlang Prototypes

The SafeErlang prototypes[7] were not implemented with the intension to create secure SafeErlang systems; instead, they were implemented with the intention to test language constructs that could support the security needed. Because of this, the prototypes have some major security holes that need to be fixed before we have a SafeErlang system that is secure.

3.5.1 Access Control

The access control manager of an entity in a SafeErlang system is the run-time system of the node that issues the capability that corresponds to the entity. Access

[6]They actually choose to load the parse-tree representing the code which is almost the same thing.
[7]PoSE and SSErl.

requests to an entity can come to the run-time system in two different ways: through built-in functions[8] and through the network interface[9]. When a request to access an entity is made, the capability presented in the request has to be authenticated, so that the capability hasn't been forged, and the required access right has to be verified to be contained in the capability.

The approach used to enforce access control in both PoSE and SSErl is rewriting of function calls. The rewriting is performed by a modified compiler which injects code that performs the actual access control. Access control can in this way be enforced as long as all code that is executed is recompiled by the modified compiler. It is possible to load code that not has been recompiled and by this bypass the access control.

The network interface was left unprotected in both PoSE and SSErl. The access control which is performed when a distributed access request is made is based on the assumption that the remote node doesn't cheat. The prototypes have been implemented by modification of only the parts of the ERLANG run-time system written in ERLANG with no modifications to the parts of the ERLANG run-time system written in C. Communication through the network interface never goes through ERLANG code[10] which makes the network interface hard to protect without modifying the parts of the run-time system that have been implemented in C.

In SSErl authentication of capabilities and verification of access rights are implemented, but in PoSE validation of capabilities is missing.

All accesses to entities on a node *have* to go through either built-in functions or the network interface; there exist no other way to access entities. By performing authentication and verification of capabilities in the actual built-in functions and in the network interface, which all are implemented in the parts of the run-time system implemented in C, the access control mechanism becomes impossible to bypass.

Capabilities were in both PoSE and SSErl implemented as tuples containing the information needed. By implementing capabilities as a fundamental data type, managed by the run-time system much like any other data type, all operations involving capabilities can much easier be optimized. This was acknowledged in [B97]. I will discuss this further later in this report.

3.5.2 Secure Communication Channels

The prototypes lack the capability to communicate confidentially over open networks. It is of utmost importance for the security of the system that communication is confidential. If capabilities can be intercepted by eavesdropping the whole access control mechanism will become useless. By encrypting all communication between different SAFEERLANG systems, one can get secure communication channels.

[8]The process requesting access is located on the same node as the entity which is about to be accessed.

[9]The process requesting access is located on another node than the entity which is about to be accessed.

[10]Except for some communication performed in order to establish a connection.

3.5.3 Sub-nodes

System utilization limits are quite hard to enforce efficiently when the sub-node concept is implemented in the parts of the run-time system implemented in Erlang. It is also hard to enforce these limits in a way that is satisfactory in a pure Erlang implementation. There are no mechanisms available to control scheduling of processes from the Erlang level except for the process priority mechanism which is too coarse grained to be useful.

3.5.4 Remote Code Loading

By transferring code as parse trees and compiling the code by a trusted compiler, one can easily ensure that the byte code will behave correctly. This is quite an inefficient operation, though. Incorporating a byte code verifier in the run-time system would be more efficient.

3.6 This SafeErlang Implementation

The problems identified in the previous section require quite a lot of work. Instead of working with all of these problems, Bertil Karlsson and I concentrated our works on the parts of SafeErlang which we consider essential. Bertil Karlsson concentrated on secure communication channels between SafeErlang systems [K00], and I concentrated on access control. The implementation of sub-nodes and remote code loading have been left for future work.

4 Essential Access Control Terminology

This section introduces the essentials of access control. Access control is thoroughly discussed in [CJ97, P97].

One should distinguish between access control policies which are requirements, access control models which are formal representation of policies, and access control mechanisms which are the actual enforcement of the policies based on the models.

Objects are passive entities which can be accessed. Subjects are active entities which access objects and other subjects; in other words, subjects are sometimes viewed as objects.

At a certain point in time all subjects in a system have limitations on which objects they can access and how they can access these objects. If these limitations depend on previous accesses performed by subjects on objects, it is considered to be a complex access control policy, otherwise a simple access control policy. An example of a complex policy: if someone has read X then no-one is allowed to write to Y.

Access control models can be divided into *mandatory* and *discretionary* access control models. Discretionary models are models where the object owner decides whom to grant access to the object. In a mandatory model, access control decisions are made on a system wide basis. An example of a mandatory model is the military model where generals have access to all documents, colonels have access to all documents except for top secret documents, and privates only have access to public documents.

4.1 Access Control Matrix

Simple, discretionary access control can be modeled by an access control matrix as shown in figure 1. Subjects are indices on the rows and objects on the columns. Every element in the matrix determines which access rights a subject has on an object. For example, to determine which access rights S_2 has on S_3, one just inspects the element in row 2, column 3.

Access control could be enforced by an access control manager which stores a large access control matrix for the whole system and inspects this access control matrix in order to determine if access requests should be granted or not. Access control matrices typically become very large and sparse which doesn't make them suitable for implementation. By cutting the access control matrix into pieces, it becomes more feasible to handle.

4.2 Access Control Lists

If one cuts the access control matrix up by column, one gets *access control lists* associated with every object. All non-empty elements in a column constitute an access control list for the object that is the index of the column.

When a subject requests access to an object, it presents an access request containing type of access, object to access, and its own identity to the access control manager which grants or denies the request. The access control manager's job is to authenticate the subject requesting access to the object and then verify that the au-

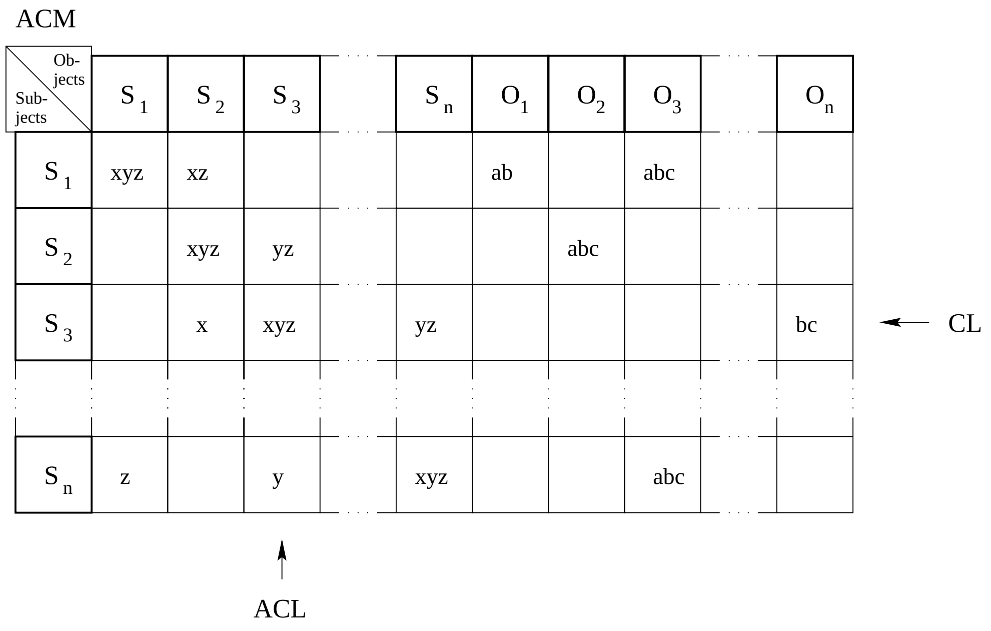

Figure 1: Access Control Matrix

thenticated subject has authorization to the object which it request access to. This is done by inspection in the access control list associated with the object.

When the access control manager authenticate a subject, the access control manager establishes the identity of the subject with great certainty.

Propagation of access rights to an object from one subject to another is performed by modification of the access control list associated with the object. The request to modify an access control list may go through the object owner, the access control manager, or some other entity which has the ability to modify the access control list.

4.3 Capability Lists

If one cuts the access control matrix up by row, one gets *capability lists* associated with every subject. All non-empty elements in a column constitute a capability list for the subject that is the index of the row.

When a subject requests an access to an object, it presents an access request containing type of access, object to access, and a capability that refers to the object to the access control manager which grants or denies the request. The access control manager's job is to authenticate the capability presented in the access request to the object and then verify that the authenticated capability give the needed authorization to the object, which it refer to, by inspection of the rights contained in the capability.

When the access control manager authenticate a capability it verifies that the capability is genuine, i.e., that it hasn't been forged.

Propagation of access rights to an object from one subject to another is performed by propagation of a capability that refers to the object. If a subject only wants to propagate a subset of the access rights contained in the capability, the subject can restrict the capability before it is propagated. When a capability is restricted, rights are removed from it. Rights can never be added to a capability. The subject restricting the capability can, of course, keep the unrestricted capability for itself.

5 Related Work

Access control issues have been dealt with in a lot of other languages. Java is probably the most well known example. Before I present how the SafeErlang access control model has been constructed in more detail, I will describe how access control has been incorporated in Java and Safe-Tcl.

5.1 Java

This short summary of the Java access control model is based on [GS98, GMPS97, JAVA99, G98].

The security model used in JDK[11] 1.0 and JDK 1.1 is known as the sand-box model. Trusted code is executed without limitations, and untrusted code is executed in a very constrained environment, i.e., a sand-box. In JDK 1.0 all remotely sourced code was considered untrusted, and all local code was considered trusted. JDK 1.1 provided the ability to digitally sign[12] code which could be used to selectively run trusted, signed, remotely sourced code outside of a sand-box. In JDK 1.2 this sand-box model has been evolved, and access control has become much more fine grained. This through the protection domain concept. A code unit, i.e. a class, is associated with a protection domain which in turn is associated with a set of access permissions. In other words, there is a mapping from every class to a set of access permissions through a protection domain. The rest of this section will describe JDK 1.2.

From a security perspective, a class is fully characterized by its origin and its digital signature. A class may be signed with one signature, multiple signatures, or no signature at all. Different security policies can be defined for different class origins and signatures. When a class is about to be loaded, the class loader consults a policy object in order to determine which access permissions to associate with the class. The class loader then associates the class with a protection domain that corresponds to the set of access permissions retrieved from the policy object.

A thread of execution is the unit which one enforces access control upon. A thread of execution is typically multiple threads performing a task. These threads will typically call methods in different classes. When an actual access is about to be performed, the call stack of methods may include methods of classes associated with a lot of different protection domains. If the access is going to be granted, a permission corresponding to the access has to be present in all of the protection domains associated with the call stack of methods. The set of access permissions granted at any point is the intersection of all permissions in all of the protection domains associated with the call stack of methods, at that point. Any code can at any time call the method `checkPermission(p)` of the `AccessController` class in order to test if the permission `p` would be granted or not at this point. If the permission `p` isn't granted at this point, a `java.security.AccessControlException` will be thrown.

[11] Java Development Kit.

[12] A class can be signed with a private key corresponding to a public key. In this way one can associate a class with, for example, the identity of a person through a digital signature. For information about digital signatures see [P97, CJ97].

When a new thread is created, it inherits all of the protection domains associated with its parents call stack up to the creation point. This because otherwise a thread of execution would be able to increase its access permissions just by creating new threads. Observe that it is a difference between a thread of execution and a thread!

Protection domains associated with a call stack can also be dropped. This when one wants code to exercise its own permissions that are not available to its callers. By enclosing code with invocations of the `beginPrivileged()` and the `endPrivileged()` methods of the `AccessController` class, the enclosed code will be executed in a privileged state. All associated protection domains up to the point where `beginPrivileged()` was invoked will be ignored except for the protection domain associated with the class where the `beginPrivileged()` invocation was performed. New protection domains entered inside the enclosing will not be ignored, though. This construct should be used with care.

If no actions what so ever are taken, i.e. no security policy is defined etc., the JDK 1.2 environment will default to the sand-box model of JDK 1.0.

5.2 Safe-Tcl

This short summary of Safe-Tcl is based on [LOW97].

Safe-Tcl is an extension of Tcl which is an interpreted scripting language. Scripts in Tcl are based on commands implemented in C or C++. Tcl provides a set of basic commands, but additional commands can also be added. These commands provide the basic building blocks of Tcl-scripts. Because the commands are written i C or C++, the commands can be implemented to perform anything that a program can be implemented to perform. Some commands access resources and the system in such ways that one would like to enforce access control upon the usage of these commands.

The Safe-Tcl security model is based on safe interpreters. An untrusted application, defined as an applet, is run in a safe interpreter which is supervised by a trusted application which runs in a master interpreter. A trusted application running in a master interpreter has complete control over the safe interpreter and the applet. One has also defined a safe base of commands which are considered safe to allow any applet to freely execute. These commands are available inside all safe interpreters.

By running an applet in a safe interpreter with only the safe base of commands available, one is ensured that the system won't be accessed by the applet in any way that may endanger the safety of the system. The applet becomes very isolated, though, which reduces the use of running it at all. In order to be able to let the applet communicate with the outside world in a controlled way, one has an alias mechanism. The application in the master interpreter can create an alias to a dangerous command, on which it wants to enforce limits, instead of supplying the command as it is. From the safe interpreter an alias is viewed as an ordinary command, but it actually is a Tcl-script inspecting the arguments passed to it and then invoking the real command in the master interpreter, *if* appropriate. For example, one can introduce an alias for the `socket` command which only can open sockets to a predetermined set of locations.

The safe interpreter and the alias mechanism offer the ability to enforce access control on applets. In order to automate this access control, *security policies* can be defined and installed through the master interpreter. A security policy is the safe base of commands together with a set of aliases. The implementation of a security policy is a set of Tcl scripts implementing the aliases of the policy.

When an applet is loaded into a safe interpreter, it only has access to the safe base of commands and an alias with which the applet can request different security policies to be loaded. The trusted application, running in the master interpreter, determine if the requested security policy should be granted or not. If a security policy is granted, it is loaded into the safe interpreter; otherwise, an error is returned to the applet which then can try to request other policies. Once a policy has been granted, other policies can never be granted to the applet. This because it would effectively compose policies even if the old aliases were removed, which is not safe. I will not go into why composition of policies isn't safe. This is discussed in [LOW97].

6 Access Control in SafeErlang

Access control is in this implementation of SafeErlang enforced by usage of capabilities. Capabilities are used in much the same way as in PoSE and SSErl, but there are some differences. This section describes how capabilities are viewed from the SafeErlang programmer's point of view and how they are used by him or her.

One can easily identify three Erlang entities that need access control protection, namely: processes, ports and nodes. The corresponding identifiers have therefore been transformed into capabilities. In Erlang, a node identifier is simply an atom, so a node identifier in SafeErlang is really a new data type. Apart from these data types there is one more Erlang data type that has been transformed into a capability, namely the reference data type. References are today globally unique identifiers often used to identify a specific request/response pair of messages. By transforming them into capabilities they also become unforgeable which is beneficial. This because the requester now can be sure that the response really is a response to the request; under the assumption that the process which sends the response is trusted. In other words, a third process cannot fake a response and get away with it.

A completely new data type called user capability has also been introduced. A user capability can, as the name implies, be used to enforce access control on any user defined resource. The SafeErlang programmer can use user capabilities to implement an access control manager for any type of resource he or she wishes. For example, a file server where access control on files is enforced by using user capabilities. User capabilities were introduced in SSErl but with a slightly different design than in this implementation.

For every type of SafeErlang capability except for user capability there is a predefined set of rights which can be contained by the capabilities. All predefined rights are listed in appendix C. For references the set is empty. The set of rights a user capability contains is defined when the capability is constructed, and the number of rights that a user capability can contain is unlimited.

In most of the operations where capabilities are used, they are explicitly supplied by the programmer, as in the send operation "`Pid ! Message`", but sometimes they are indirectly supplied. Only local node identifiers can be indirectly supplied, though, an example of this will be given in section 6.2.

6.1 SafeErlang Capabilities

Process identifiers, port identifiers, node identifiers, and references are used in much the same way as in Erlang. For example, a message is sent in SafeErlang in the same way as in Erlang. The difference is that if the process identifier doesn't contain a `send` right the message will be ignored, instead of reaching the intended receiver. The introduction of node identifiers and user capabilities is the most significant change to the Erlang programmer.

6.1.1 Restriction

Rights can only be removed from capabilities but never added. In order to remove rights from a capability, one uses the built-in function `restrict/2`. It takes a capability and a list of rights as arguments and returns a new restricted capability. The newly restricted capability contains the intersection of the rights of the supplied capability and the rights in the supplied list.

Normally operations aren't destructive[13] in ERLANG and SAFEERLANG; although, there are some destructive operations[14]. Restriction of capabilities isn't a destructive operation. This means that a restriction of a capability doesn't change anything in the capability being restricted; instead, a restricted copy of the supplied capability is returned.

`restrict/2` operates on all capabilities both local and remote capabilities of all capability sub-types. Because `restrict/2` also operates on capabilities that are issued on remote nodes, a call to `restrict/2` may block the calling process indefinitely. Only the node which issued a specific capability can create valid restrictions of this capability. This makes it necessary to communicate with the node that issued the capability in order to restrict it. In order for the SAFEERLANG programmer to be able to limit the time a process should wait for a restrict operation, the built-in function `restrict/3` has been introduced. The first two arguments are the same as for `restrict/2` and the third argument is an integer determining the timeout time.

6.1.2 Propagation

Capabilities may be propagated by processes in any way they wish; thus, propagation of access rights also implies propagation of the right to propagate these rights. This is an important concept that is essential for the security of the system. A SAFEERLANG program should only propagate capabilities:

- with security sensitive rights to other entities that are trusted not to propagate these sensitive rights to untrusted entities.
- with rights that aren't security sensitive to less trusted entities, only when really needed.

This may appear to be a disadvantage for the capability scheme compared to the access control list scheme, but actually the same requirements are required for the access control list scheme. The only difference is that it is easier to prevent that security sensitive access rights explicitly leak to untrusted subjects, but one cannot prevent that they implicitly leak. For example, a subject, say X, which has gotten security sensitive access rights to object Y could act as a proxy for untrusted subjects. The untrusted subjects would then gain security sensitive access rights to Y through X even though they don't have any access rights in the access control list associated with Y. [P97][15] discusses these issues in more detail.

[13] A destructive operation can change the content of data. An operation that isn't destructive cannot change the content of data only produce a modified copy of the data.

[14] Some operations on process dictionaries and ets-tables.

[15] Section 5.2

Capabilities are normally propagated in ordinary SafeErlang messages, but they can also be propagated by e-mail, on a floppy disc, etc. For this to be possible we have designed a textual capability format which only uses alpha-numeric characters. The textual capability format is described in appendix B. In order for the SafeErlang programmer to convert capabilities into capabilities on textual capability format and vice versa, some built-in functions have been introduced. These built-in functions are `capa_to_list/1`, `list_to_capa/1`, `write_capa/2`, and read_capa/1[16]. They are described in appendix D.

6.1.3 Equality tests

A process identifier of Erlang is either equal or not equal to another process identifier. The introduction of capabilities changes this. Two capabilities can be equal in the sense that they are exactly equal or equal in the sense that they refer to the same entity but not exactly equal because they differ in the rights they contain.

In Erlang, the comparison operator `=:=` tests for exact equality, and the comparison operator `==` tests for coarse equality. For most tests that can be performed `=:=` and `==` will give the same result, but for comparison between integers and floats there is a difference. `5.0 =:= 5` is false but `5.0 == 5` is true.

We have chosen to let the `=:=` and `==` behave like the following:

`=:=` Capabilities are compared for exact equality.

`==` Capabilities are compared for equality of reference; that is, if two capabilities refer to the same entity this comparison results in `true`, otherwise `false`.

The built-in function `node/1` takes a capability as argument and returns a node identifier without any rights corresponding to the node on which the supplied capability was issued. This node identifier will have its time-stamp set to zero[17]. A node identifier with its time-stamp set to zero is considered to be coarsely equal to another node identifier if they have the same node name and public key even if the other node identifier has a time-stamp other than zero. This is a special case; in all other cases capabilities also have to have the same time-stamp to be considered as coarsely equal.

6.2 Node Identifiers

Processes on a SafeErlang node may have different access rights to the node on which they are running; some processes may even have no access rights at all. Every process has a default node identifier associated with itself. When a process calls the built-in function `node/0`, the default node identifier is returned. This node identifier may contain all rights or less.

When a process spawns another process, the newly spawned process inherits its parent's default node identifier by default. The parent can also restrict the node

[16] `write_capa/2` and `read_capa/1` are "pseudo built-in functions" and actually use `capa_to_list/1` and `list_to_capa/1`.

[17] This because the time-stamp of the node that issued, for example, a process identifier isn't contained in the process identifier.

identifier that the child should have as default node identifier. When a process is spawned with `spawn/3`, the child inherits its parent's default node identifier assuming that it contained a `spawn` right; otherwise, the spawn operation will fail. When `spawn/4` is used, the new process will get the node identifier supplied in the call to `spawn/4` as default node identifier.

One may want to spawn a process which shouldn't be able to spawn other processes. One would then like to remove the `spawn` right of the node identifier supplied in the call to `spawn/4`. This would cause the spawn operation to fail, though, because the `spawn` right is needed for the spawn operation. By using the built-in function `spawn_opt/2` instead and supplying the option `{install_nid, MasterNid, ChildNid}` where `ChildNid` will be the default node identifier of the child and where `MasterNid` will be checked for a `spawn` right, one can spawn a process which won't be able to spawn any further processes.

Apart from this, node identifiers are used in very much the same ways as node identifying atoms are used in ERLANG.

6.3 User Capabilities

User capability is a completely new data type, not found in ERLANG. User capabilities have been introduced in SAFEERLANG in order to allow the user to protect resources that the user identifies in the system, in the same way as the system protects entities like processes, ports, etc. User capabilities can, for example, be used to enforce access control on files in a file-server implementation. An example implementation of a file-server which uses user capabilities to enforce access control is shown appendix E.

Because the system cannot possibly identify all possible resources that a user wants to enforce access control upon, the user has to implement the access control manager[18]. Access control is preferably enforced by the resource manager[19] which means that the access control manager and the resource manager preferably is one and the same. The resource manager and the access control manager don't necessarily have to be the same service, but for simplicity I will use the term resource manager to mean a resource manager combined with an access control manager. When the resource manager is implemented, it is a small effort to add the access control part to the resource manager.

For the resource manager to be able to use user capabilities to enforce access control on the resources that it manages, the resources must be referred to by user capabilities. This means that the resource manager issues user capabilities for the resources that it manages and then propagates these capabilities to clients. These capabilities are later presented to the resource manager by the clients when they are requesting access to the corresponding resources.

When a capability is presented to the resource manager, the resource manager has to be able to authenticate the capability and verify that the capability contains

[18]The entity which has the responsibility to check that access requests are permitted and based on this grant or deny accesses.

[19]The entity which performs the actual accesses on behalf of the clients to the resources that it manages.

the right corresponding to the requested access. In order for the SafeErlang programmer to perform these actions, the built-in function `has_valid_right/2` has been added. `has_valid_right/2` takes a capability and an atom representing a right as arguments and returns `true` if the capability hasn't been forged or modified and contains the right represented by the atom, otherwise `false`. By using this built-in function, the SafeErlang programmer can implement a resource manager which enforces access control by using user capabilities.

In order for the resource manager to create user capabilities the built-in function `make_capa/1` has been introduced. `make_capa/1` takes a list of atoms representing all the rights that the capability shall contain as argument and returns a user capability with all rights set. By creating the capability with the built-in function `make_capa/2`, one can also attach any arbitrary SafeErlang term to the capability. This term will also be protected by the capability protection. In order to extract the attachment from a capability, the built-in function `attachment/1` can be used. For an example of how attachments can be used, see appendix E. The built-in function `restrict/2` operates on user capabilities in the same way as on any other capability.

A resource manager typically issues user capabilities in the following way when resources are created:

1. *Reception of a message containing a creation request.* The message typically contains the process identifier of the client requesting the creation; a user capability referring to the resource manager; an atom describing the type of creation; a message tag, i.e. a reference; and some data which is used to create the resource.

2. *Master capability test.* The user capability in the request message is compared for coarse equality with the master capability referring to the resource manager. The resource manager has the original master copy of this capability. If the equality test fails the request is ignored; otherwise, the resource manager continues to the next step.

3. *Authentication and verification.* The master capability copy, from the request message, is tested for authenticity and required access right with `has_valid_right/2`. If `has_valid_right` returns `false`, the request is ignored; otherwise, the resource manager continues to the next step.

4. *Resource creation.* The actual resource is created, and a new user capability is created with `make_capa/2` and stored together with the resource.

5. *A reply message is sent to the client.* This message typically contains the newly created user capability and the message tag received in the creation request message.

An access request is typically served by the resource manager as follows:

1. *Reception of a message containing an access request.* The message typically contains the process identifier of the client requesting the access; a user capability referring to the resource which the access is requested for; an atom describing the type of access; a message tag, i.e. a reference; and maybe some other data required for the access.

2. *Resource look-up and capability test.* The resource that access is requested for is fetched. The capability in the request message is compared for coarse equality with the capability stored together with the resource. If the equality test fails the request is ignored; otherwise, the resource manager continues to the next step.

3. *Authentication and verification.* The capability is tested for authenticity and required access right with `has_valid_right/2`. If `has_valid_right` returns false the request is ignored; otherwise, the resource manager continues to the next step.

4. *Access.* The requested access is performed.

5. *A reply message is sent to the client.* This message typically contains some data showing the status of the performed access and the message tag received in the access request message.

6.4 Installation of Protection

In order for a node to be able to protect the capabilities issued on the node, a protection value is attached to every capability. These protection values enable the run-time system to be able to authenticate the capabilities issued on the node. The information that the run-time system needs to be able to produce these protection values are a key and a protection scheme to use. This key has to be kept secret. When a node comes up, a default protection scheme and a default key are set. Other protection than the default protection can be used, though. This is true for all capabilities even for remote capabilities.

Installation of protection, other than the default, for a capability is done by calling the built-in function `install_protection/2`. The call

```
install_protection(LocalNid, {Capa, PType, PKey})
```

installs protection of the type[20] `PType` with the key `PKey` for the capability `Capa`. The `LocalNid` argument is required to be a node identifier referring to the node on which the protection is installed and is required to contain an `install_protection` right.

The reason to why one should be able to install protection for remote capabilities is that it enables the SAFEERLANG programmer to be able to implement distributed services without any single point of failure. A call to `has_valid_right/[2-3]` or

[20]I.e. a protection scheme.

`validate/[1-2]` normally fails if the issuing node is down; this due to the fact that only the issuing node knows how to calculate the protection value. When the node on which the call is made cannot contact the issuing node, the operation cannot be performed. When protection has been installed locally for a remote capability, the protection value can be calculated locally; thus, there is no need for communication with the issuing remote node.

A distributed service with access control of user defined resources can by the usage of user capabilities with installed protection easily be implemented. A distributed service like this might, for example, be a replicated file server or a replicated database. When files or data are replicated, the capabilities referring to these items are also replicated. When an instance of the distributed service receives a capability together with a file or data, the service just has to install the agreed protection. Access control can then be enforced by all instances that share the file or the data.

The concept of this shared access control responsibility is based on that all entities that are part of the distributed service are trusted; otherwise, the whole concept fails. This may seem like a major drawback, but it really isn't; why would one like to have an untrusted entity as part of the service?

6.5 Reincarnation

If a capability is distributed widely in order to provide a public service and the service is replaced by another, it can be hard to distribute a new capability to all possessors of the old capability. The service could, for example, be a server process and the widely distributed capability could be a process identifier to the server with the rights `send` and `info`. In Erlang one would typically register the server process so that clients could access the service in the same way even if the server process has to be restarted. This is possible in SafeErlang too, but one would then only get the ability to propagate the `send` right. This because one is only allowed to send messages to registered processes in SafeErlang. The registration service might also not be suitable because it is public. One may want to widely provide the service but not to everybody.

By distributing capabilities instead of a name to a registration service, one gets a much more fine grained access control to the service, but a problem arises when the service needs to be restarted. The capabilities distributed will refer to a process that has terminated. If one just spawns a new process, it will get a new process identifier which one has to distribute to all clients. This is a very awkward way to restart services. To solve this problem we have introduced the ability to reuse old capabilities.

In order to reuse an old process identifier, one can spawn a new process with `spawn_opt/2` and supply the option `{install_pid, OldPid}`. The newly spawned process will then use `OldPid` as process identifier. `OldPid` has to be valid and contain all rights. If `OldPid` uses another protection than the default the protection, it can be installed by also supplying the option `{install_protection, PType, PKey}` where `PType` is the protection type to use and `PKey` is the key to use.

Node identifiers can also be reused by supplying the command line switch "`-install_nid NID_FILE PR_TYPE KEY_FILE_NAME`" where `NID_FILE` is a text file containing the node identifier in textual capability format, PR_TYPE is protection type to use, and KEY_FILE_NAME is the file name of the file containing the protection key to use. The node identifier contained in NID_FILE has to be valid under the supplied protection, has to contain all rights, and has to have a host name part of its node name that corresponds to the machine where the node is started.

The example file-server implementation shown in appendix E uses reincarnation of process identifiers in order to restart file-servers.

7 A Brief Overview of the ERLANG Run-time System

SAFEERLANG has been implemented by modification of the R6B-0[21] release of ERLANG. In order to give the reader a feeling of how the SAFEERLANG modifications fit in, I will here present a brief overview of how ERLANG's R6B-0 runtime system is constructed.

7.1 ERLANG Processes

The whole ERLANG system, i.e. an ERLANG node, runs in one operating system process. The ERLANG processes are scheduled by the ERLANG run-time system inside this operating system process.

7.1.1 Scheduling

ERLANG processes are scheduled from four different queues corresponding to the different priority classes: max, high, normal, and low. Every process belongs to one and only one priority class. As long as there are runnable processes in the max priority class, processes are only scheduled from this class. If there are no runnable processes in the max priority class and there exist runnable processes in the high priority class, only processes from the high priority class are scheduled. When there are no runnable processes in neither the max nor the high priority classes, processes are scheduled both from the normal and the low priority classes. Processes from the low priority class will be scheduled every eighth time and the rest of the time will be reserved for processes in the normal priority class. Within each priority class, processes are scheduled in a "Round Robin" fashion[22].

A context switch is performed when the executing process has performed 1000 reductions or if the executing process gets blocked. A function call is counted as one reduction. Also other things like, for example, garbage collection are counted as reductions. The number of reductions consumed during a garbage collection is determined by the size of the heap after the collection has been performed.

When 2000 reductions or more have been performed by the executing processes, the runtime system checks if there are any I/O operations that can be performed and if so performs these I/O operations.

7.1.2 Memory Management and Garbage Collection

Every ERLANG process has its own separate heap which is garbage collected independently of the other heaps of the other ERLANG processes. The garbage collector is a copying generational garbage collector using Cheney's algorithm for copying. The only references into the heap of a process that exist are from the process itself. For example, references from the stack and message queue of the process. The only references out from the heap of a process are pointers to binaries.

[21]beam emulator version 4.9.1

[22]The queue of runnable processes is managed in First-In-First-Out order.

Binaries can be shared between processes which means that the garbage collection of a binary potentially depends on every process in the system; therefore, reference counting is used to garbage collect binaries. When a process' heap has been garbage collected, the heap is searched for dead references to binaries. When such a reference is encountered, the reference count on the binary is decremented, and if the reference count reached zero the binary is deallocated.

7.2 ERLANG Data Types

ERLANG terms are stored as tagged words. A tagged word is a 32-bit word with the four least significant bits reserved for a tag. This tag determines the term's type. The interpretation of the remaining bits depends on the tag. They are either type specific data, or a pointer shifted two bits to the left. Only addresses divisible by four are addressed which means that the two least significant bits of a pointer are always zero. This makes it possible to fit a four bit tag even though the pointer is only shifted two bits. The two most significant bits are lost which means that one can only address 1/4 of all addressable addresses.

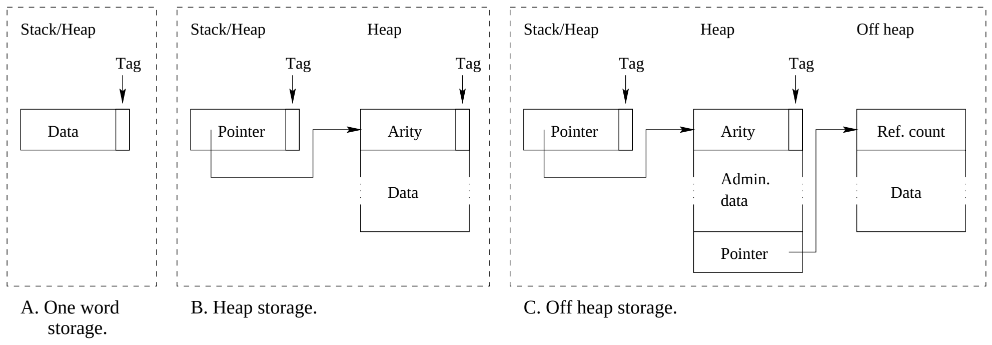

Figure 2: Data storage.

Data is stored in essentially three different ways in the system[23]:

One word storage Figure 2A.
Data of the data types which only require less than or equal to 28 bits of storage are stored in one 32-bit word. 28 bits of data and a four bit tag. The one word storage data types are small integer[24], atom, port identifier, and process identifier.

Heap storage Figure 2B.
When more than 28 bits of information have to be stored, the tagged word is used as a pointer to the actual data on the heap. The data types that use this

[23]Observe that I present a generalized view of how ERLANG terms are stored and *not* an exact description of how they are stored.

[24]Integers between -2^{27} and $2^{27}-1$ are stored internally as small integers. The internal data types small integer and big integer are viewed by the ERLANG programmer as the data type integer.

kind of storage are big integer, float, reference, and fun. The first word of the data part on the heap is tagged `THING` and contains a sub-tag[25] and an integer that determines how many words of data that follow.

Off heap storage Figure 2C.
Data of the data type binary is stored off heap; or more precise, almost all of a binary is stored off heap. A tagged pointer refer into a process' heap. A "handle" on the heap, also tagged as `THING`, refers out from the heap to the actual data. All "handles" on the heap of a process are linked together as a linked list. This enables the garbage collector to quickly locate all of a process' off heap references.

Lists and tuples are stored as special cases of the "heap storage" scheme. Lists are built up of "cons cells" which are stored on a heap. A "cons cell" consist of two words. The first word contains an ERLANG term, and the second word contains a tagged list pointer to the next "cons cell" or `NIL`, i.e., the empty list[26].

A tuple is stored almost exactly as in the ordinary "heap storage" scheme except that the heap part contains a number of ERLANG terms instead of a number of "raw" words of data. The first word has the tag `ARITYVAL` and store an integer determining how many ERLANG terms that follow.

A process identifier can only contain 28 bits of data, but the information that needs to be stored about a process require far more than 28 bits. This information is stored in a process data structure. Process data structures are referred to by a process table. Process identifiers contain an index into this table. When a process is to be accessed through a process identifier, the run-time system looks in the process table to see if there exists a process data structure at the index contained in the process identifier. If so is the case, an access can be performed, otherwise not. There are only 15 bits available for the process table index in a process identifier which sets the maximum size of the process table to 32768. The process table is a static table, i.e. an array of size 32768, which makes look-ups in the table fast.

Ports are managed in the same manner as processes, i.e. port data structures are referred to by a port table which is indexed by port identifiers, with some minor differences.

An atom refers into an atom table which contains the textual representation of the atom.

Process identifiers, port identifiers and references, also, contain an index into a distribution table. The distribution table contains distribution entries which contain information about nodes[27]. If a communication channel has been established to a node, also, information about the channel is stored in the distribution entry corresponding to the node. There is only 8 bits available for the distribution table index in a process identifier which sets the maximum size of the distribution table to 256.

[25]This sub-tag is 8 bits long and is used to determine the type of the following information.
[26]The second word can actually contain any ERLANG term if the list isn't a well formed list.
[27]One entry is used for the local node and the rest is used for remote nodes.

7.3 Network Interface

The network interface essentially consists of the `net_kernel` server, a distribution protocol, a serializer, and a network driver.

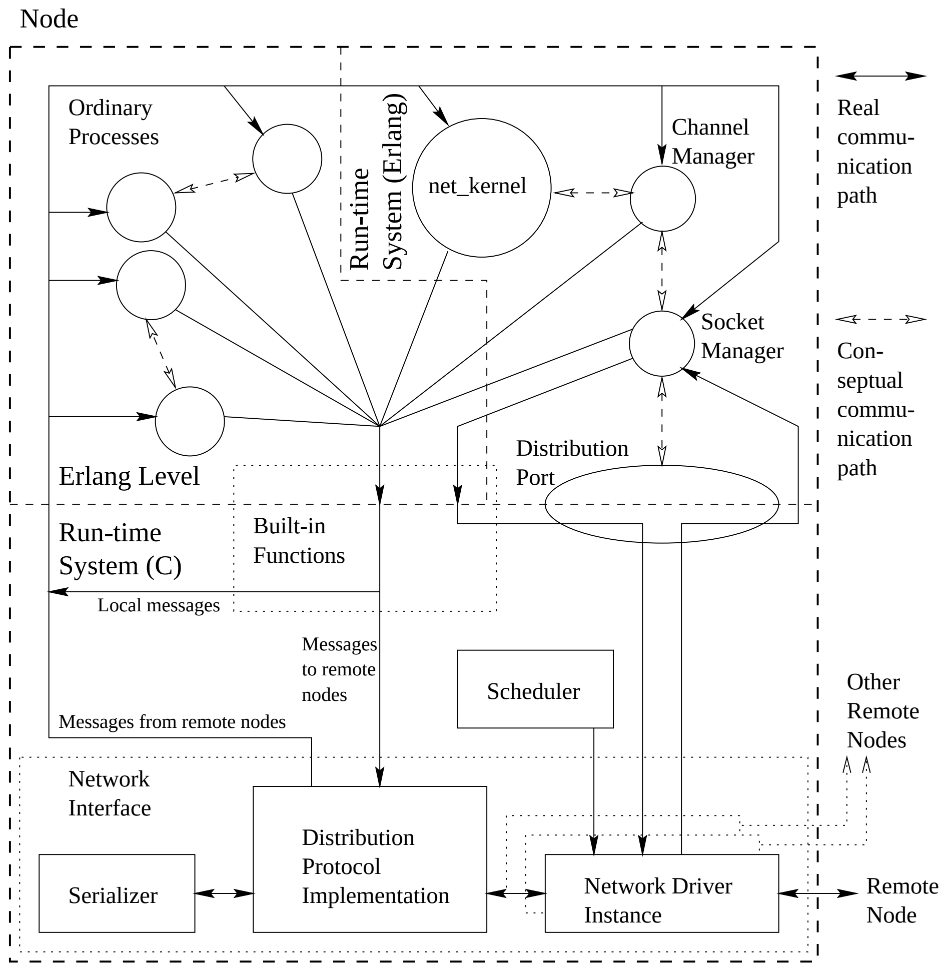

Figure 3: Schematic overview of a distributed ERLANG node.

`net_kernel`
The ERLANG server `net_kernel` runs on every distributed ERLANG node. It is responsible for establishing and some minor maintaining of connections to other ERLANG nodes.

When a new connection is to be established, the `net_kernel` spawns a connection manager process which establishes the connection. The connection manager spawns a socket manager process which opens a port. The port starts a network driver instance through which a TCP/IP connection is established. After the TCP/IP connection has been established, a handshake sequence according to a handshake protocol is performed by the connection manager and its remote counterpart.

After the connection has been established, the connection managers are responsible for monitoring each other. The monitoring is performed by sending each other messages every 15 seconds in case there hasn't been any other traffic on the connection. If a connection manager doesn't get a response from the other side in 60 seconds, the connection is torn down.

Distribution protocol
Connected ERLANG run-time systems exchange control messages according to a distribution protocol. These control messages are ERLANG tuples. The first element in a control message tuple identifies which operation the control message represents. For example, a "distributed link" operation or a "distributed send" operation. The remaining elements in a control message depend on the type of operation.

Serializer
The serializer's job is to transform ERLANG terms into byte vectors and vice versa. These byte vectors can be sent over the network or stored on disk. The serializer also contains an atom cache which is used to decrease the amount of data that is transferred over the network. Atoms that are frequently sent between two nodes are encoded as small integers instead of as strings.

Network driver
The network driver is the low level interface to the operating system's socket implementation. The network driver is controlled[28] from the ERLANG level through an ERLANG port. A read operation on a socket is invoked by the run-time system if there exists data to read from the socket. A write operation on a socket is invoked by the run-time system if there is data queued and it is possible to write to the socket. The actual read and write operations are non-blocking. Invocations of read and write operations are scheduled between scheduling of processes[29].

7.4 Message Passing

7.4.1 Local Message Passing

Message passing between ERLANG processes on the same node is, a little simplified, just a matter of copying ERLANG terms from the heap of the sending process to the heap of the receiving process. When a binary is sent in a message to another process, the stack and the heap data parts are copied, and the reference count is incremented.

[28]This involves for example setting up initialization parameters and checking that the remote node is reachable.

[29]Described in section 7.1.1.

Message passing between ERLANG processes on the same node is performed as follows:

1. A message buffer is allocated. This message buffer is like a small heap.
2. The message is copied into the message buffer.
3. The reference to the message is saved in the message queue of the receiving process, and the message buffer is saved within the receiving process. The buffer is then used as part of the heap of the receiving process until the next garbage collection. At the next garbage collection, the message will be copied onto the heap, and the message buffer will be deallocated.
4. The receiving process reaches a matching `receive` expression, and the message reference is fetched from the message queue. This may happen when the actual message is still in the buffer, i.e., before it has been moved onto the heap by the garbage collector.

7.4.2 Remote Message Passing

When there is no connection established between the node where the sending process resides and the node where the receiving process resides, the following will be performed:

1. The built-in function !/2 will detect that there is no connection to the receiving node. A "distributed send trap" is invoked which causes the sending process to load and execute a trap function.
2. The trap function sends a message containing a request to connect to the node on which the receiving process resides, to the local server `net_kernel`. The sending process is then blocked waiting for a reply from `net_kernel`.
3. `net_kernel` contacts the ERLANG Port Mapper Daemon (EPMD)[30] on the receiving node's host to get the port number on which the receiving node listens on.
4. `net_kernel` spawns another ERLANG process which establishes a connection to the receiving node. On the receiving node another process is also spawned to establish the connection.
5. `net_kernel` sends a reply message to the process which invoked the establishment of the connection, informing that the connection is up.
6. The built-in function !/2 is reinvoked, and the message is sent to the receiver as described below.

[30]EPMD runs as a separate operating system process on every host on which it exist distributed ERLANG nodes. All distributed ERLANG nodes register themselves at the local EPMD on startup.

When a connection between the participating nodes already has been established, a distributed send operation is performed as follows (1-3 on the sending node and 4-6 on the receiving node):

1. A control message containing sender, receiver, and an integer indicating that this is a send operation is created.
2. The control message and the Erlang message are serialized.
3. The message is sent on the socket.
4. The message is received on the remote socket.
5. The message is unserialized and inspected, and the integer indicating that this is a send operation is found.
6. The actual message is put into a message buffer and queued in the message queue of the receiving process in the same way as if it was a message sent locally on the node.

7.5 Real-time

An Erlang implementation is obliged to offer soft real-time properties. The whole implementation of the Erlang run-time system of the R6B-0 release is imbued with the obliged soft real-time properties in mind. Here are some examples:

- The garbage collection of memory is performed on a “per process basis” which splits the garbage collection work up into smaller pieces compared to if the whole system was to be collected at once.
- When I/O operations are scheduled, a maximum of one write and one read per file descriptor is performed.
- All write and read operations on sockets are non-blocking.
- Processes are scheduled with short time-slices.

8 Implementation of Access Control in SafeErlang

Most of the work of integrating the chosen access control mechanisms into the Erlang run-time system involves the introduction of capabilities into the system.

Information	Size
Type	3 bits
Time-stamp	8 bytes
Rights	4 bytes
Protection value	10-20 bytes
Node name	$\geq$4 bytes
Public key	100-300 bytes

Table 2: Information contained in a capability.

Table 2 shows the information needed in a capability. The size of this information is somewhere between 150 to 350 bytes depending on the public key's size.

8.1 Unique Capabilities

In order to get capabilities which uniquely identify entities, all capabilities have been attached with a time-stamp. A time-stamp is 8 bytes long which is more than enough to represent every nano-second during a century. The most significant four byte word is used to store the number of seconds since 00:00, 1 Jan., 1970 and the least significant four byte word is used to store nano-seconds of a second.

When a new capability is created, its time-stamp is set to the current time. The time-stamp resolution, therefore, sets a maximum limit on how fast new capabilities can be created. Consecutive creation of new capabilities isn't likely to happen faster than the minimum difference between time-stamps, though. If so is the case, the system will be blocked until the next change of current time which usually is quite a short period of time.

These time-stamps uniquely identify an entity on the node where the entity resides. The node name and public key are used to globally identify entities. Restrictions of a capability contain the same time-stamp, node name, and public key as the original capability; otherwise, the capability would refer to another entity.

It may seem that we are introducing our own "century-bug", but we really aren't. If a collision[31] should happen, one of the involved entities has lived for more than 100 years, and the other entity is created exactly 100 years later and, also, is of the same type as the first. This isn't likely to happen very often!

[31] Two capabilities with identical time-stamp, node names, and public keys that refer to different entities.

8.2 Internal Capability Format

A new internal format has been designed to efficiently store capabilities. A capability forms a linked list of length 4, see figure 4. Observe that this is not an ERLANG list. The different elements are `HCapa`, `OCapa`, `OOCapa` and `KeyNode` with the tagged pointer `Capa` as a reference to the list[32] . The `Capa` element may reside both on a stack and a heap. The `HCapa` element may only reside on a heap. All the rest of the elements may only reside off heap.

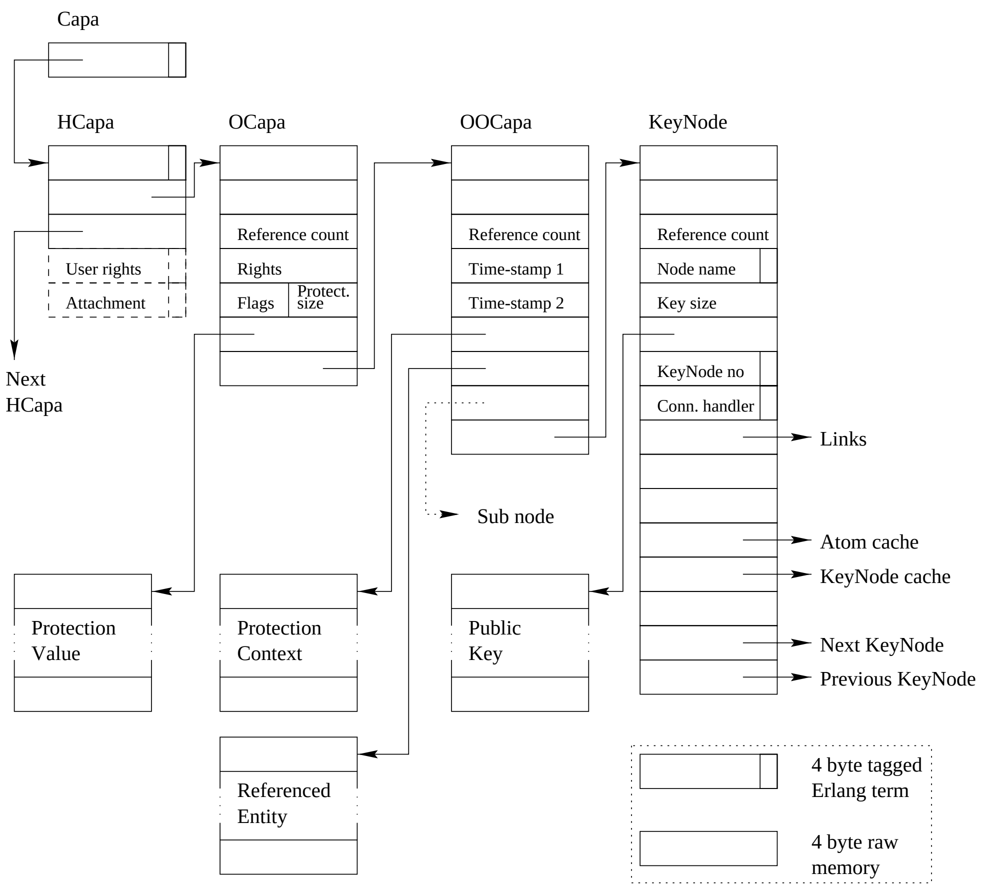

Figure 4: The internal capability format.

The idea behind this format is to duplicate as little information as possible. All capabilities that refer to entities that reside on the same node have the last element

[32]The protection value is considered to be part of the `OCapa` element, the protection context and referenced entity data structures are considered to be part of the `OOCapa` element, and the public key is considered to be part of the `KeyNode` element.

(`KeyNode`) in common. All capabilities that refer to the same entity have the two last elements (`OOCapa`, `KeyNode`) in common. And all capabilities that are equal have the three last elements (`OCapa`, `OOCapa`, `KeyNode`) in common. The following text refers to these lists as capability tails.

A user capability has some extra data. The user defined rights are stored as a tuple of atoms on the heap with a reference from the `HCapa` element. If a user capability contains an attachment, the `HCapa` element will also refer to the attachment. The `HCapa` element will be further explored in section 8.4.

There is a pointer to a sub-node in figure 4. This word isn't used in this implementation; it has been introduced as a preparation for the future sub-node implementation. The sub-node data structures should typically form a tree, and the idea is to have a pointer from every capability to the sub-node data structure which corresponds to the sub-node on which the entity referred to by the capability resides.

Some nice properties of this format:

- *Reduced memory consumption.* Capability tails are never duplicated. If the `OCapa` and `OOCapa` elements were to be stored on the heaps of the processes, memory consumption would increase with about 4-6 times.

- *Unlimited number of entities.* Because there is only one `OOCapa` element per referenced entity, one can refer directly to the data structure which stores information about the actual entity[33]. If Erlang entity identifiers would have referred directly to entity data structures, all entity identifiers in the system referring to a specific entity data structure would have had to be reset when the entity data structure was to be removed. This is a quite complex operation which would have damaged the real-time properties of the system. When the chosen capability format is used, one can reset the reference to a process data structure just by reseting *one* pointer; thus, there is no need for entity tables. There is no hard limit on the number of `OOCapa` elements in a SafeErlang system; therefore, SafeErlang have no hard limits on the number of entities in the system.

- *Simple equality tests.* All capability tails in a system are unique; therefore, equality tests are just a matter of testing for pointer equality. Two capabilities are exactly the same if they have equal pointers to their `OCapa` elements. For user capabilities, one also has to compare their user defined rights and attachments for structural equality. Two capabilities refer to the same entity if they have equal pointers to their `OOCapa` elements.

- *Garbage collection of* `KeyNode`*s and unlimited number of* `KeyNode`*s.* In Erlang, node information is stored in a distribution table consisting of distribution entries. The information stored in these distribution entries has in SafeErlang

[33]These entity data structures are only used for local processes and ports that are alive. References don't correspond to entities and do, therefore, not need to refer to any entity data structures; node identifiers store all entity information in the `KeyNode` elements (a future sub-node implementation may change this); and information about entities that are referred to by user capabilities are stored by the user.

been moved into the `KeyNode` elements, and the distribution table has, therefore, been removed. The `KeyNode` elements are garbage collected and there is no limit on the number of them.

- *Efficient copy operations.* Data is very frequently copied internally in the run-time system, in particular, when passing messages and when collecting garbage. When a capability is copied, only four machine words have to be copied which corresponds to 16 bytes in this 32-bit version of ERLANG/SAFEERLANG. These 16 bytes compare favorably with the 150-350 bytes of data that are conceptually copied when a capability is copied.

Properties which may cause problems:

- *Increased number of cache misses and page faults.* Because a capability isn't stored continuously in memory, there will probably be an increased number of cache misses and page faults compared to if capabilities had been stored continuously. This will have a negative impact on the performance. Estimating the performance loss due to this is difficult. It is possible to trade off memory consumption for decreased number of page faults and cache misses by copying frequently used information from the off heap elements to the `HCapa` element.
- *Administrative overhead.* Because capability tails need to be stored uniquely, there will be an administrative overhead to ensure that no duplicate capability tails will appear.

8.3 Capability Import Table

To avoid duplicate capability tails, the run-time system has to check every possible operation that may produce duplicate tails. These types of operations are quite limited, though. As long as capabilities are propagated and used internally on a node, no duplicates can appear. The only operations that may produce duplicates are operations which import capability elements. These import operations are:

- Reception of messages from the network.
- Creation of capabilities. Capabilities are created when entities are created with two exceptions: when references are created and when user capabilities are created. The built-in functions that create new capabilities are: `spawn/[3-4]`, `spawn_link/[3-4]`, `spawn_opt/2`, `open_port/2`, `make_ref/0` and `make_capa/[1-3]`.
- Import of capabilities through the built-in function `list_to_capa/1`.
- Restriction of capabilities by using the built-in function `restrict/2`.

In order to efficiently prevent duplicates of capability tails, a capability import table is used. This import table is implemented as three hash tables; one for each type of capability tail that exists[34]. When a capability is imported, a look-up for a fitting

[34](`KeyNode`), (`OOCapa`, `KeyNode`) and (`OCapa`, `OOCapa`, `KeyNode`).

tail in the capability import table is performed. If a tail that fits is found that one is used; otherwise, a new tail is inserted.

8.4 Garbage Collection

Garbage collection of capabilities is performed in much the same way as garbage collection of binaries. When the heap of a process has been garbage collected, the heap is searched for dead `HCapa` elements. All `HCapa` elements on the heap of process are linked to each other as a linked list, so the search is just a matter of following the list and checking if the `HCapa` elements have been moved by the garbage collector, or not. When an `HCapa` element that not has been moved, i.e. a dead `HCapa` element, is encountered, the reference count of the corresponding `OCapa` element is decremented. If the reference count of the `OCapa` element reached zero, it's deallocated after the reference count on the corresponding `OOCapa` element has been decremented. The same procedure is performed for `OOCapa` and `KeyNode` elements.

The usage of reference counts to reclaim unused memory may be controversial. Andrew W. Appel states the following arguments against usage of reference counts [A97]:

- Cycles of garbage cannot be reclaimed.
- Incrementing, decrementing and testing the reference count for every assignment to a pointer variable is expensive.
- Real-time properties may be damaged by "recursive decrementing"[35].

In SafeErlang, all cycles constructed by the run-time system will be explicitly broken by the run-time system before reclamation. The SafeErlang programmer cannot create any cycles on his own, so cycles of garbage are not a problem.

The second argument doesn't apply to SafeErlang at all. As a matter of fact, it is just the other way around! By using reference counting, there will be less to do than if the capability elements had been stored on the heaps of the processes and garbage collected in the ordinary way. When a capability is assigned to a variable, the operation is exactly the same as if the whole capability had been stored on the heap; namely, a new copy of the `Capa` word will be stored. The same occur for all other operations which produce an extra reference, from within the same process, too. When a capability is passed on the same node from one process to another, the `Capa` and the `HCapa` elements are copied and the reference count on the `OCapa` element is incremented. If all capability elements had been stored on the heaps of the processes, all of them would have had to be copied which would have been more to do.

The third argument isn't a problem in the current implementation but could be in the future. When only `OCapa`, `OOCapa`, and `KeyNode` elements are considered, the maximum depth of the recursive decrement recursion is only three which I do not

[35]When the reference count reaches zero on a memory block which refers to other memory blocks, the reference counts on the referred blocks have to be decremented. These reference counts can, in turn, reach zero which generate more decrementing of reference counts. This "recursive decrementing" can produce a lot of work for the system if a lot of reference counts reach zero.

consider a problem. But when the sub-node concept is implemented, these recursive decrements could, under certain conditions, become quite deep and potentially block the system with deallocations for quite a while. This could potentially be damaging for the real-time properties of the system. If the sub-node implementation is done so that the depth of the recursive decrements is unlimited, this problem could be solved as follows.

One sets a default limit **M** to be the maximum number of deallocations of capability elements that a process is allowed to perform under a time slice. When a process has deallocated **M** capability elements, the process just puts any further capability elements that should be deallocated into a list of capability elements to be deallocated later. When a process finishes its time slice without having reached **M** deallocations and there are elements in the list, deallocation of elements in the list is performed until **M** number of deallocations have been performed. If the length of the deallocation list becomes larger than a certain predetermined length **High**, **M** is increased by Δ at every context switch until the list shrinks to a length under **High**. The value of **M** is kept constant until the list shrinks to a length under **Low**. **M** is then decreased by Δ at every context switch until it reaches the initial value, see figure 5. Δ could be either a constant or a function of, for example, the size of the capability import table or the capability allocation rate.

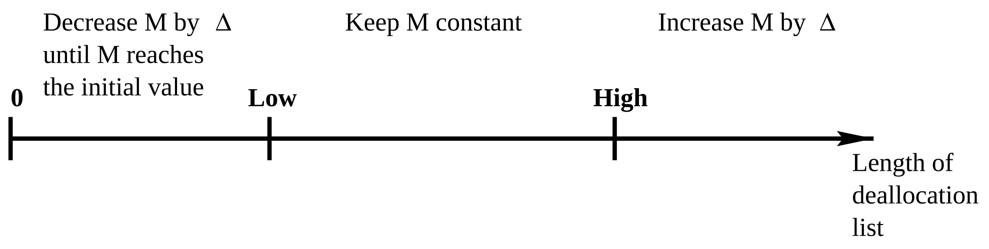
Decrease M by Δ until M reaches the initial value
Keep M constant
Increase M by Δ
0
Low
High
Length of deallocation list

Figure 5: Modified capability element deallocation approach.

By limiting the maximum number of deallocations that will be done during one time slice, we will not damage the real-time properties of the system.

8.5 Capability Protection

Capabilities are in this SAFEERLANG implementation protected by so called HMAC values. HMAC is an abbreviation for Hashed Message Authentication Codes. HMAC can be viewed as a function which takes two arguments, a secret key and data of arbitrary length, and returns a fixed size message authentication code. This value can only be computed by someone who has access to both the secret key and the data.

This secret key used to protect capabilities has *nothing* to do with the public key attached to the capability. The public key is used to encrypt communication when the capability is used, and the secret key is used to protect the integrity of the capability.

8.5.1 The HMAC Function

The HMAC function uses a cryptographic hash-function together with the data and the key in order to produce the message authentication code. Any iterative cryptographic hash-function like MD5 or SHA will do.

Algorithm 1 The HMAC function.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{MAC = H(K' \oplus opad \mid H(K' \oplus ipad \mid D)),\ where} \\
\oplus & = & Exclusive\, or \\
\mid & = & Concatenation\, function \\
H & = & Cryptographic\, hash\, function \\
B & = & Block\, size\, of\, H \\
ipad & = & The\, hexadecimal\, value\, 36\, repeated\, B\, times \\
opad & = & The\, hexadecimal\, value\, 5C\, repeated\, B\, times \\
D & = & Data\, (of\, any\, length) \\
K & = & Key\, (of\, any\, length) \\
K' & = & \begin{cases} K,\, if\, key\, size \leq B \\ H(K),\, if\, key\, size > B \end{cases}
\end{array}
$$

The HMAC function produces message authentication codes as shown in algorithm 1. I have implemented the HMAC protection schemes shown in table 3 by modification of the HMAC algorithms presented in [BCK97, CG97]. The hash-functions used for this implementation are the implementations of MD5, SHA, and SHA1 of the SSLeay[36] SSL implementation. Other implementations of hash-functions can easily be used to complement or replace these.

MAC Scheme	MAC Length (bytes)
HMAC-SHA1	20
HMAC-SHA1-96	12
HMAC-SHA	20
HMAC-SHA-96	12
HMAC-MD5	16
HMAC-MD5-96	12

Table 3: Supported capability protection.

8.5.2 Message Authentication Codes Managed by the Run-time System

When a capability is stored on the same node that issued it, no protection value is stored with it. When a capability is about to leave the node, a protection value for the

[36]SSLeay version 0.9.0b which implement SSL version 3.0. This software package was down-loaded from <http://www.cryptosoft.com>.

capability is computed and attached to it. The protection value is computed with the following arguments: secret key, known only by the issuing node, and the concatenation of type, time-stamp, rights, node name, public key and also attachment if it is a user capability with attachment. When the capability is stored on a remote node, the protection value is stored together with the capability. When the capability returns to the node where it was issued, the run-time system recomputes the capability's protection value. If the newly computed protection value matches the protection value that came along with the capability, the capability is marked as valid and the protection value is dropped; otherwise, the capability is marked as invalid, and the protection value is kept along with the capability. As long as the secret key remains secret, only the node that issued the capability can compute a message authentication code that corresponds to the capability data and the key. This means that modifications and fabrications of capabilities will be detected.

When a locally issued capability enters the system and fails the authentication, the protection value isn't dropped. This is because it may later become valid. If an old process identifier is reused, it has to be reused with the same protection algorithm and key as it was initially created with. The old protection algorithm and key are installed when the process that uses the old process identifier is created. An old process identifier that enters the system before the corresponding process has been created will be considered invalid, but when the process has been created, it should be considered valid. Therefore, there must be sufficient information to re-validate an invalid process identifier; thus, all information that initially came along with it when it entered the system has to be saved.

No protection value is stored with a locally issued capability; instead, a valid bit is used to mark if a capability is valid or not. The valid bit is part of the "flags" data of the `OCapa` element, see figure 4 on page 42. This valid bit enables the run-time system to authenticate a capability without having to recalculate the protection value every time the capability is used. This works because, as long as a capability is kept inside the run-time system and stored on the internal capability format, only the run-time system itself has access to the valid bit.

8.5.3 Protection Contexts

In SAFEERLANG, there exists a need to be able to use different protection types and/or different keys for capabilities referring to different entities. There is also a need for this to be efficiently managed. This has been solved in the following way.

When a protection algorithm and key are installed, a protection context is created from the algorithm and key which is then stored together with the capabilities it is used with. All capabilities stored on a system that refer to the same, remotely or locally located, entity have the same `OOCapa` element in common. A pointer to a protection context can be stored in this `OOCapa` element, see figure 4 on page 42. If there is no pointer to a protection context in the `OOCapa` element of a capability and the capability refers to a local entity, the default protection context will be used to protect the capability; otherwise, the installed protection context will be used. All SAFEERLANG nodes install a default protection context on start up.

A protection context contains function pointers to functions which perform the actual computation of the message authentication code and the pre-computed values $K' \oplus ipad$ and $K' \oplus opad$, see algorithm 1 on page 47. My HMAC implementation is more general and more efficient than the HMAC implementation used in SSErl, which is a slightly modified [BCK97] implementation, because my implementation pre-compute the $K' \oplus ipad$ and $K' \oplus opad$ values and can use any hash-function implementation. In SSErl, K', $K' \oplus ipad$, and $K' \oplus opad$ are recomputed every time a message authentication code is computed. If the size of K is greater than B, this means that $H(K)$ also has to be recomputed. When this is the case, my implementation reduces the computational work by about a third compared to the SSErl implementation; assuming that every call to the hash-function H requires about the same amount of computational work.

8.5.4 Other Capability Protection Schemes

In this implementation, a few different HMAC protection schemes have been implemented to protect capabilities. The password protection of capabilities, implemented in SSErl, has not been implemented. Not because it isn't a good protection scheme, but because there were other more important work to do. This implementation can easily be extended to support any protection scheme that is based on a protection value kept along with the capability when it is distributed and checked when the capability is used. For example, password protection or any MAC scheme. Protection values stored with remote capabilities are not interpreted by the run-time system; thus, a system extended with other protection schemes can communicate with systems not extended.

We have had discussions about a protection scheme that could support capability restriction without the need to contact the node which issued the capability[37]. The idea is that a capability is restricted by remote nodes by construction of a new protection value dependent on the old protection value. Protection value used at the top level, i.e. protection value attached by the issuer, could be, for example, an HMAC value. A new protection value for a remotely restricted capability would then be constructed by asymmetric encryption of the previously attached protection value together with the previously contained rights of the capability. The public key attached to the capability is used as key to asymmetrically encrypt with. This enables the issuing node to unwind the recursively created protection value and authenticate the capability when it is used.

There is one problem with this approach. Repeated restrictions of capabilities can generate capabilities which are identical but have different protection values attached to them. Equality tests may then require communication with the issuer of the involved capabilities. This because only the issuer can unwind a recursively created protection value and by this authenticate a capability. Equality tests are fundamental operations which are part of every pattern matching operation. This would mean that processes could block, potentially for ever, on every operation in which pattern matching is involved[38] which is unacceptable. We haven't yet found a scheme like

[37]Erik Klintskog at SICS initially came up with this idea.

[38]Function calls, `receive` expressions, `case` expressions, `=` expressions.

this, which doesn't require communication with the issuer when equality tests are performed, except for some naive schemes which produce enormous protection values. If one finds such a scheme, it would be very desirable to implement.

8.6 Authentication and Verification of Capabilities

There are two interfaces through which all accesses to entities have to go through, if one disregards entities protected by user capabilities, as stated in section 3.5.1. All accesses performed locally on a node have to go through the built-in functions, and all accesses performed from remote nodes have to go through the network interface.

8.6.1 Network Interface

When access requests to entities on a node come from processes located on other nodes, they all have to pass through the network interface. The natural place to intercept these requests is in the distribution protocol implementation, shown in figure 3 on page 36.

The serializer has been extended to manage capabilities. For outgoing capabilities, this includes the ability to produce protection values for capabilities and transforming capabilities of the internal capability format into capabilities of the external capability format. The external capability format is described in appendix A. For incoming capabilities, managing capabilities includes the ability to authenticate capabilities, i.e. check if they are forged, and the ability to transform capabilities of the external capability format into capabilities of the internal capability format. The protection value is dropped, and the valid bit is set to true when a local capability or a remote capability with installed protection context passes the capability authentication; otherwise, the protection value is kept with the capability, and the valid bit is set to false.

All incoming control messages pass through the distribution protocol implementation. They are also unserialized by the distribution protocol implementation by usage of the serializer. After that they are inspected, so that the run-time system can perform the requested operations. All of this makes it a perfect place to perform the actual access control. When a message has been received, it is unserialized. The serializer sets the valid bit to true on all valid capabilities, so the only thing that needs to be done by the distribution protocol implementation is to test that the valid bit is set to true and that the capability contains the right corresponding to the requested access. If the tests are positive, the requested access is performed; otherwise, it is rejected.

Rights are stored as a bit-mask in the "rights" data of the `OCapa` element, see figure 4 on page 42. Testing for a specific right is, therefore, just a matter of testing if one bit is set or not. This is true for all capabilities except for user capabilities. User capabilities are never tested in the network interface, though. Testing of rights in user capabilities will be explained in section 8.6.2.

The operations described so far fulfill the need to perform access control on access requests through the network interface. There are two more problems that need attention. The node on which a process requesting access executes on will if no further action is taken "leak access rights". For example, a monitor control message looks like this: `{19, RequestingPid, RemotePid, Ref}`. If the run-time system uses the capability with all rights of the requesting process as `RequestingPid` when it composes the control message, a malicious remote node could leak this information to its processes. The only access right which the requesting process conceptually propagates when it performs `monitor(process, RemotePid)` is the right to send "down messages"[39] to itself, no more, no less. Therefore, the run-time system should restrict the `RequestingPid` capability so that it only contains the `down` right. The `down` right of process identifiers was introduced especially for the monitor control message. I have implemented restriction of capabilities in control messages for all distributed access requests, so access rights actually propagated correspond to the access rights conceptually propagated. Except for the `down` right, no new rights had to be introduced.

The other problem is that if a malicious node sends badly formed control messages, it can crash the receiving node. Dealing with this is just a question of inspecting all control messages thoroughly and ignoring those which are badly formed. This has also been implemented.

8.6.2 Built-in Functions

All accesses to a node's local entities are performed through the built-in functions. Most of the predefined rights that exist in the different capability types have the name of a built-in function, for example `link`. This because most of the accesses that need to be controlled correspond to built-in functions. Authentication of capabilities and verification of required rights contained in capabilities used in access requests are performed in the code which implements the built-in functions.

All capabilities in the system have their valid bit set to true or false depending on whether they are valid or not, so authentication of capabilities is just a matter of testing the valid bit. Testing for a specific right in a capability is just a matter of testing the corresponding right bit with one exception, namely, when user capabilities are tested for a specific right. When an access request is made with a user capability, the user implemented access control manager tests that the required right is contained in the user capability supplied in the access request by calling the built-in function `has_valid_right/[2-3]`. This is the only place in the system where rights of user capabilities are tested. It is slightly more complicated than just testing one bit. The rights of a user capability are stored in a tuple of atoms referred to by the `HCapa` element of the capability, see figure 4 on page 42. Testing that a specific right is contained in a user capability is performed by searching for an atom in the tuple matching the atom supplied as the second argument in the call to `has_valid_right/[2-3]`[40].

[39]See description of `monitor/2` in section 2.7.

[40]This is a linear search. Capabilities are assumed to have quite few rights (normally less than 10), so this is not considered a problem.

8.7 Remote Capability Operations

Processes have to be able to restrict all capabilities that they possess. When a local capability or a remote capability with installed protection is to be restricted, the local run-time system can provide this service, but the local run-time system cannot provide this service when a remote capability without installed protection is about to be restricted. In order for the run-time system to be able to restrict remote capabilities without installed protection, it has to send the request of restriction to the node that issued the capability and then wait for a reply containing the newly restricted capability.

The actual restriction of capabilities is performed in the built-in function `restrict/2` which is implemented in C. The part of the run-time system implemented in C executes in sequence without any support for concurrency. This because it is the part of the run-time system implemented in C that provides the ability to run Erlang processes concurrently. If the implementation of `restrict/2` sent a message and then blocked waiting for a reply, the whole SafeErlang system, including all processes running on it, would be blocked waiting for this reply. This is *not* what we want. To solve this, the implementation of remote operations is implemented in the part of the run-time system implemented in SafeErlang which has support for concurrency.

A service registered as `capa_server` runs on every SafeErlang node. This service is implemented in SafeErlang. The `capa_server` serves requests, from remote nodes, to perform operations on capabilities issued on the node where the `capa_server` runs. These requests correspond to the built-in functions `restrict/2`, `has_valid_right/2` and `validate/1`. All of these built-in functions are "shadowed" behind SafeErlang code which calls the actual built-in functions implemented in C. If one of the actual built-in function cannot perform the operation because a remote capability without installed protection has been supplied, it returns the atom `remote_capability`. If the "shadowing SafeErlang code" gets a `remote_capability` atom as return value, it sends a request message, corresponding to the built-in function called, to the `capa_server` on the issuing node and then blocks waiting for a reply message. When the reply message is received, the result of the operation is extracted and then returned to the calling process; just as if the operation had been performed locally. Of course, the "shadowing code" will return directly if the actual built-in function succeeds. The SafeErlang programmer will never see any sign of this "shadowing code", and the `remote_capability` atom will never be exposed to him or her.

Because the `restrict/2`, the `has_valid_right/2`, and the `validate/1` built-in functions can potentially block the calling process forever, one can supply an optional timeout argument.

8.8 Alternative Approaches

Other approaches, than the approach with unique capability elements stored off heap and with garbage collection of capability elements by reference counting could have been chosen. The other approaches considered are presented below with short motivations to why they weren't chosen:

Alternative 1
A capability is stored as one large element on the heap except for the key which is stored as a binary. Garbage collection of capabilities as all other data located on the heaps are garbage collected.

- Increased memory consumption because information would be duplicated. According to those tests I've performed, memory consumption would increase between 4-6 times.
- Equality tests become more complex. This because one would have to test for structural equality instead of just testing for pointer equality.
- More data would have to be copied when performing garbage collection (an increase of about 200%) and node-internal message passing (an increase of about 200%).
- Another mechanism providing unlimited number of entity data structures and distribution entries would have to be implemented. Entity data structures and distribution entries cannot be referred to directly, as in the chosen alternative. One way of doing this could be to replace the static tables used in Erlang with dynamic tables, for example, hash tables. A look-up in a hash table is more expensive than following four pointers as in the chosen alternative, though. These hard limits can easily be a security problem because a node will crash or behave unpredictably when a limit is reached; therefore, they have to be removed.
- Garbage collection of distribution entries would have to be dealt with in some other way.

Alternative 2
All capability elements are unique and are fragmented as in the chosen alternative but stored on a common heap which is garbage collected just like the process heaps.

This is the most interesting alternative approach. It would be very interesting to implement this alternative and compare the performance with the chosen alternative. It has one drawback, though.

- In order for the garbage collector to be able to garbage collect this common heap, it has to construct a root-set[41] which depends on every process in the system. Some kind of table of possibly alive references to capability elements would have

[41]The root-set of a heap is the set of all references into the heap.

to be used, so the whole system wouldn't have to be searched at once. This to avoid damaging the system's real-time properties. The administration of this table would introduce an extra overhead.

Alternative 3
All `OCapa` and `OOCapa` capability elements are uniquely stored per process but stored on the heaps of the processes instead of off heap. The `KeyNode` capability elements are stored off heap as in the chosen alternative. Garbage collection of capability data on heaps as all other data located on heaps are garbage collected, and garbage collection of `KeyNode` elements as they are garbage collected in the chosen alternative.

- Increased memory consumption because information would be duplicated.
- One would need one capability import table for every process that exists. Lookups in these tables would have to be done for all message passing between processes even when messages are passed within a node. This would have a negative impact on the performance of message passing.
- More data would have to be copied when performing garbage collection (the increase wouldn't be as large as 200% but still an increase) and node-internal message passing (an increase of about 200%).
- The internal copy operation would be more complicated because it suddenly depends on source and destination of the data.
- Another mechanism providing unlimited number of entity data structures would have to be implemented. Entity data structures cannot be referred to directly as in the chosen alternative. This could be remedied as described in alternative 1 above but with worse performance than in the chosen alternative. These hard limits can easily be a security problem because a node will crash or behave unpredictably when a limit is reached; therefore, they have to be removed.

Alternative 4
A capability is stored as a tuple with protection value and public key stored as binary data. Ordinary ERLANG process identifiers, port identifiers, and node identifying atoms stored in the capability tuple to identify entities. Capabilities are, in other words, constructed as in SSErl with the exception that the public key is also stored in the capability tuple. This approach is attractive because capabilities can be constructed by combination of data of the ordinary, unmodified ERLANG data types. This approach has some major drawbacks, though.

- Protection values would always have to be stored together with capabilities even when they are located on the node that issued them. This is because, capabilities could be constructed from the SAFEERLANG level of the system. This alternative would both increase memory consumption, because of the attached protection values that cannot be dropped, and also require recalculation of protection values at every access to an entity when authentication is performed. In the chosen alternative, capabilities cannot be constructed by the SafeErlang programmer. This makes it possible for the run-time system to drop the protection value of a valid capability when it is located on the node that issued it and use the valid bit for authentication, instead.

- Equality tests become more complex. This because one would have to test for structural equality instead of just testing for pointer equality.
- More data would have to be copied when performing garbage collection (an increase of more than 200%) and node-internal message passing (an increase of more than 200%).
- Because of that the Erlang data types would be used as they are, SafeErlang would be stuck with all of Erlang's limits on maximum number of entities and maximum number of distribution entries. Distribution entries wouldn't be garbage collected either, but this is a minor problem since there only can exist 256 distribution entries. This limit of 256 distribution entries would be a *major* problem for a node running a public service which manages data from many different nodes. When the node has got in contact with node specific data[42] from 255 different nodes[43], it cannot represent any other nodes until the node is restarted. This means that it cannot create connections to any other node or receive any node-specific data from any other node apart from the 255 nodes which it already has come in contact with. These limits could be lifted; for example, expanding the data part of the process identifiers so that they can refer to a larger number of nodes and a larger number of entries in the process table. This would mean that process identifiers would have to be stored on the heaps of the processes or off heap because the word used to store a process identifier in Erlang doesn't have any unused bits. Modifications like these are possible, but why not chose another alternative[44] and do it right at once if one anyway is going to modify Erlang's data types?

8.9 Performance

To be able to evaluate what impact the modifications made to implement access control have had on the system's performance, I ran the Estone benchmark on both an Erlang R6B-0 system and a SafeErlang R6B-0 system, see table 4. The results of the benchmark are both promising and worrying.

The results of the "ets data-dictionary" tests show an enormous performance loss. I find it very unlikely that the modifications, without bugs, would cause an overhead like this. I have run the SafeErlang system through some test software which fails on a test of the "ets data-dictionary". This indicates that this performance loss probably is due to some bug. Unfortunately I haven't had the time to look into this. What is more worrying is that the time required for message passing has increased by about 100% for small messages, 70% for medium size messages, and 50% for huge

[42] Process identifiers, port identifiers and references.

[43] One distribution entry is used to represent the own node.

[44] For example, the alternative chosen!

messages. This has to be looked into and also has to be done in the future, due to lack of time.

The rest of the tests gave very good results. One could argue that tests such as list manipulation and small integer arithmetics haven't got anything to do with the modifications done to introduce access control in SAFEERLANG. This is not the case, though. All operations performed on a SAFEERLANG system are affected by the administrative overhead induced by, for example, garbage collection of heaps which is affected by the introduction of capabilities.

If one corrects the number of estones that the "ets data-dictionary" test for SAFEERLANG got, to be the number of estones it would have gotten if the number of estones would have decreased as the average decrease of estones was for all other tests, it would have gotten 138 estones. This would mean that SAFEERLANG would have gotten a total of 2153 estones. SAFEERLANG's 2153 estones compared to ERLANG's 2420 estones shows, at least in the eyes of the author[45] of the Estone benchmark, that SAFEERLANG has almost as good performance as ERLANG.

[45]No, I haven't asked him, I mean that the Estone benchmark reflects his opinion of how important the performance of different parts of the system are.

Title	**Time (ms)**		**Estone**		**Expected %**	**Real %**	
E: Erlang SE: SafeErlang	E	SE	E	SE		E	SE
List manipulation	6987	7206	217	210	7	4	0
Small messages	22628	45754	136	67	10	12	1
Medium messages	38852	66111	156	91	14	20	1
Huge messages	10295	15323	48	32	4	5	0
Pattern matching	1955	2094	396	370	5	1	0
Traverse	2450	2635	202	188	4	1	0
Port i/o	25170	26477	177	168	12	13	0
Work with large dataset	1994	1960	139	142	3	1	0
Work with large local dataset	1743	1970	160	141	3	1	0
Alloc and dealloc	2367	2420	52	51	2	1	0
Bif dispatch	11818	15280	65	50	5	6	0
Binary handling	13634	17951	36	27	4	7	0
Ets datadictionary	7176	6845154	155	0	6	4	96
Generic server (with timeout)	21048	26954	119	93	9	11	0
Small Integer arithmetics	1965	1776	141	157	3	1	0
Float arithmetics	9305	9528	3	3	1	5	0
Function calls	4658	4170	166	185	5	2	0
Timers	4677	5622	26	22	2	2	0
Links	1189	1697	26	18	1	1	0
Total time	190000	7100000					
Total Estones			2420	2015			

Table 4: Estone benchmark of Erlang R6B-0 and SafeErlang R6B-0 on a 110 MHz sparc 5 running solaris 2.6. The **Time (ms)** columns show how long time it took to perform each test. The **Expected %** column show how large part of the total time each test is predicted to take and the **Real %** columns show how large these parts actually were. The **Estone** columns show how many estones every test got. The number of estones that a test gets depends on how fast the test was performed and how important the author of the Estone benchmark thought that the test was. The more estones the better. The sum of all estones of all tests gives a value of how good the performance of the Erlang system, on a specific hardware platform and operating system, is considered to be. The reference system is an Erlang 4.3 system on a 85 MHz sparc 5 running Solaris 2.5 which gets approximately 1000 estones.

9 Comparison with Java and Safe-Tcl

Both the Java and the Safe-Tcl access control models are made for a more automated access control than the access control model implemented in SafeErlang so far. The SafeErlang access control model is extendible with more automated mechanisms, though. The mechanisms implemented, so far, can be used as building blocks in a more automated access control model. Lawrie Brown describes ways to extend SafeErlang with configurable policies in [B99]. These configurable policies have some resemblance to policies in Java and Safe-Tcl. Focus in my work has not been on these issues but on closely related issues. In section 10.5 I will describe some ideas I have had regarding future work on these issues.

9.1 SafeErlang versus Java

Subjects in Java are threads of execution. The access rights given to a thread of execution depend on which classes a specific thread and its ancestors have traversed. When a decision to grant or deny access to an entity is about to be made, the call stack has to be searched in order to determine if the access should be granted or not.

Subjects in SafeErlang are processes and the access rights given to a process correspond to all of the capabilities that the process possesses. When a decision to grant or deny an access to an entity is about to be made, the capability used in the access request is authenticated and then tested to contain the needed access right.

The operation of deciding whether or not to grant an access requires a lot less computational work in SafeErlang than in Java, at least when comparing operations performed locally on a system. In SafeErlang, it is just a matter of testing two different bits if one disregards user capabilities. When user capabilities are tested for a specific right, it is a little more work, but not much. In Java one has to go through a stack of protection domains and inspecting them one by one. This stack of protection domains could possibly be quite large and possibly cause large computations. When one considers remote accesses, it is a little harder to compare the two different approaches, but I think that the SafeErlang approach would stand a comparison. This because the remote access control enforcement in SafeErlang doesn't require very much more work than the local access control enforcement. If one disregards the information that needs to be transferred, it is only the computation of protection values that differ between local and remote access control enforcement in SafeErlang.

If a specific access is presided by a huge computation, one would very much like to prevent this huge computation if the access anyway is going to be denied, assumed that the computation only affects the access and not other things. By using user capabilities to enforce access control, this problem is easily solved in SafeErlang. One just performs the call to `has_valid_right/2` before one performs the huge computation. If the test should fail, one just ignores the request, otherwise one continues. One can do something similar in Java. One can test with `checkPermission(p)` before one performs the huge computation, but this will cause the test to be performed twice instead of just once, one time when `checkPermission(p)` is called and one time when the actual access is performed. This is not satisfactory because these tests may require quit a lot of work, at least compared to the corresponding tests in SAFEERLANG. If one skips the first call to `checkPermission(p)`, one would have to perform the huge computation even though it is useless, which is not satisfactory at all. The SAFEERLANG approach is better.

9.2 SAFEERLANG versus Safe-Tcl

The Safe-Tcl approach is quite similar to the SAFEERLANG approach. Safe-Tcl hides dangerous commands behind aliases for different applets and SAFEERLANG enables or disables dangerous built-in functions for different processes depending on which capabilities they possess. The alias mechanism provides an access control mechanism that can be tailored for a specific applet while the capability mechanism is a binary mechanism; access is either allowed or not. The SAFEERLANG approach is more coarse grained if one only looks at it from this point of view, but the same possibility to provide tailored access control exists. By implementing a resource manager[46], as described in section 6.3, one can tailor access possibilities in an arbitrary manner. In addition these services can be provided with access transparency to any process, remotely or locally located. In Safe-Tcl, the alias mechanism only provides the service for an applet running locally on an interpreter.

[46]Which could be viewed as an alias.

10 Future Work

Implementation of sub-nodes and remote code loading have to be done in the future. These issues were discussed in section 3.5.3 and 3.5.4, and I will not discuss these issues any further in this section; instead, I am going to discuss some things related to access control. Some of these things have to be done before we have a secure system and some of them don't necessarily have to be further explored.

10.1 Existing Erlang Services

There are many services implemented in Erlang, such as the `net_kernel`, which run on an Erlang system. These services are not implemented to use the access control functionality of SafeErlang. SafeErlang is sufficiently Erlang compatible so that these services can be run on a SafeErlang system, but because they are not implemented with the intention to work in a SafeErlang environment, they may introduce security problems. The only way to solve this problem is to look through all of the code for all of these services and modify the code so that no security problems arise because of these services. This is quite a lot of work, but it has to be done before we have a SafeErlang implementation which is secure.

10.2 Work Left in the Run-time System

I have in this report talked about ports and references as capabilities. The implementation of this hasn't been completely finished. All functionality has been incorporated into the system, but all accesses to ports and references internally in the run-time system[47] have to be changed to use this functionality before the transformation into capabilities will be complete.

Bertil and I haven't had the time to merge our implementations. This means that my implementation lacks the ability to communicate over secure channels with remote nodes. This merge *has* to be done before the system is secure.

10.3 `KeyNode` Cache

The `KeyNode` part of a capability is a part of a capability that a lot of capabilities will have in common. It is also the largest part of a capability. This makes the `KeyNode` part suitable for caching when it is transferred over the network. The idea is to cache `KeyNode` elements as atoms are cached in Erlang. When a `KeyNode` cache has been implemented, frequently sent `KeyNode` elements will be sent as small integers instead of as node name and public key[48]. In order to prepare for this cache, the `KeyNode` part of a capability is serialized as a separate part attached to the end of a serialized capability, see appendix A.

[47]The part of the run-time system implemented in C.

[48]The only information that is stored in the `KeyNode` element of the internal capability format that needs to be transferred between nodes are the node name and the public key.

10.4 Protection Context Management

Today, only the default protection context can be shared between capabilities referring to different entities. This means that if a service installs protection of the same type and with the same key on a lot of different capabilities referring to different entities, there will be a lot of identical protection contexts in the system. These identical protection contexts could be replaced by one shared protection context.

By implementing a protection context table similar to the capability import table, one can prevent duplicate protection contexts from appearing in the system. Today, only one `OOCapa` element can refer to a protection context which means that a protection context is deallocated when the corresponding `OOCapa` element is deallocated. If protection contexts are shared, it is a little trickier to know when to deallocate them. This is easily solved by adding a reference count to all protection contexts, and garbage collect them as off heap capability elements are garbage collected today.

10.5 Mandatory Access Control

The access control model implemented in this SAFEERLANG system is a simple, discretionary access control model. There exists a need for additional constraints on accesses to entities which cannot be enforced by a pure capability access control system, as stated in [G89]. The modified capability system proposed in [G89] doesn't fit very well into the SAFEERLANG access control model, for several reasons which I won't go into, but it addresses a problem that has to be dealt with. I would like to see some kind of mandatory access control combined with this simple, discretionary access control introduced into SAFEERLANG.

The sub-node concept was introduced with the intention to offer a mechanism with which one could enforce limits on system utilization on an application. I see the sub-node as the entity to also enforce mandatory access control upon. By running an untrusted application inside a sub-node which only allows communication with trusted processes, one can prevent the application from leaking information to other untrusted processes[49]. The constraints on the communication through a sub-node wall could be enforced in a lot of different ways. Here are two examples of different approaches which could be implemented:

- When a sub-node is created, a set of allowed sub-nodes is set. The processes running inside of this sub-node are then only allowed to communicate with processes running inside of this set of allowed sub-nodes.

- When a sub-node is created, a message check function is installed. This function is applied to all messages[50] which enter or leave the sub-node. If the function returns `true` the message is let through, otherwise not. This approach gives a more fine grained mechanism at the cost of performance. Lawrie Brown describes a very similar mechanism in [B99].

[49] If one disregards communication through covert channels. Covert channels are described in [CJ97, P97].

[50] Together with sender and receiver process identifiers.

In both approaches one could allow processes inside the sub-node to constrain the allowed communication even more than it already has been.

These approaches aren't strictly mandatory access control models rather some combination of discretionary and mandatory. They could perhaps be denominated: hierarchically ordered mandatory access control models.

When a mechanism like this has been introduced into SafeErlang, one could easily automate the usage of it. I will here give an example of how this could be done. Policies are defined as communication constraints on sub-nodes. These policies are attached to user capabilities, for example, as lists of nids representing different allowed set of nodes or as Erlang funs used as message check functions. These policy capabilities[51] could be shared between mutually trusted nodes by installation of protection. The ability to attach policy capabilities to code is introduced. The code loader[52] is modified to automatically inspect these attached policy capabilities and to load code inside a newly created sub-node with a automatically chosen policy. The code loader would only choose between policy capabilities issued locally or issued by trusted nodes, that is, policy capabilities with installed protection. If the code loader cannot find an acceptable policy capability, a default policy with very restrictive constraints is chosen. An automated mechanism like this would provide the ability to automatically load applications and run them under different constraints depending on what policy capabilities they provide.

10.6 Interface Functions

A client interface for a service implemented in Erlang is typically a set of functions. An interface function typically sends a message to a server and then waits for reception of a reply message. In order for the server to be able to send the reply message, the interface function usually incorporates the client's process identifier into the request message. This is done by calling the built-in function `self/0`.

If one uses code for such an Erlang service unmodified in SafeErlang, the client interface will leak access rights. This problem can be solved by rewriting the code of the service. This is an acceptable solution when one can control all code being loaded into the system, but when code is remotely sourced or when one wants to use services which one doesn't have the time to inspect, the problem has to be dealt with in some other way. An easy solution is to use an "interface function proxy process" which operates as a proxy between a client and a server.

Algorithm 2 implement a module called `proxy` which could be used to call untrusted interface functions with. By calling untrusted interface functions through `proxy:call/[3-4]` instead of directly, the interface function cannot leak more access rights than `send` and `info` rights[53] of the calling process and `info` right[54] of the node which the calling process executes on. If an untrusted interface function is called

[51]User capabilities with policies attached to them.

[52]The byte code verifier has, of course, already been implemented.

[53]The new built-in function `rself/0` returns a pre-restricted process identifier which only contains a `send` and an `info` right of the calling process.

[54]The new built-in function `rnode/0` returns a pre-restricted node identifier which only contains an `info` right of the default nid of the calling process.

directly, it could potentially leak all access rights of the calling process and all access rights contained in the default node identifier of the calling process.

This approach has some drawbacks:

- Processes without a `spawn` right in their default node identifier cannot use the `proxy` module.
- Services which require registered send operations to perform their services will fail unless a node identifier to the node where the service exists isn't explicitly supplied in the call.
- Overhead due to process creation in every interface function call.

My proposal to solve these problems is to introduce some kind of way to call untrusted functions securely as opposed to ordinary function calls. These secure calls could be used by processes which want to call interface functions which they don't trust. I think the semantics should be similar to the `proxy:call` but configurable and with a syntax that differs from an ordinary function call. For example, `~a_func(Arg)` would mean that the function `a_func/1` would be called securely and `a_func(Arg)` would work as usual. If one can implement this without creating new processes for every interface function call, I think it would improve performance substantially compared to when the `proxy` module is used.

Algorithm 2 proxy module, implementing secure calls to untrusted interface functions.

```
%%% File    : proxy.erl
%%% Author  : Rickard Green <rickard.green@uab.ericsson.se>
%%% Purpose : Provide secure calls to untrusted interface functions.
%%% Created : 19 May 2000 by Rickard Green <rickard.green@uab.ericsson.se>

-module(proxy).
-export([call/3, call/4, try_call/5]).
-author('rickard.green@uab.ericsson.se').

%% call/[3-4] spawns a proxy process which executes an interface function
%% and then waits for a reply from the proxy process.
call(M, F, A) ->
    call(M, F, A, infinity).

call(M, F, A, T) ->
    OTE = process_flag(trap_exit, true),
    Tag = make_ref(),
    Proxy = erlang:spawn_opt({proxy, try_call, [rself(), Tag, M, F, A]},
                             [link, {install_nid, node(), rnode()}]),
    receive
        {Tag, return, R} ->                 % Normal return
            process_flag(trap_exit, OTE),
            R;
        {Tag, throw, R} ->                  % Exception
            process_flag(trap_exit, OTE),
            throw(R);
        {'EXIT', Proxy, R} ->               % Proxy process died
            process_flag(trap_exit, OTE),
            throw({proxy_call_exception, R})
    after T ->                              % Timeout
            process_flag(trap_exit, OTE),
            throw({proxy_call_exception, timeout})
    end.

%% try_call/5 tries to perform a call to an interface function through
%% do_call. Catches failures of do_call and if so replies the failure
%% to the client.
try_call(C, T, M, F, A) ->
    case catch do_call(C, T, M, F, A) of
        {success, T} ->
            done;
        Exception ->
            catch C ! {T, throw, Exception}
    end.

%% do_call/5 executes an interface function, replies to client
%% if successful.
do_call(C, T, M, F, A) ->
    C ! {T, return, apply(M, F, A)},
    {success, T}.
```

11 Conclusions

There exists a real need for programming languages in which one can easily implement secure distributed applications. We believe that SafeErlang can fulfill this need and compete with other languages which try to fulfill the same need. SafeErlang has an access control model that is easy to understand which is essential; otherwise, programmers are more likely to introduce security flaws by mistake. Erlang, as it is, is very well suited for easy implementation of distributed applications which SafeErlang has inherited. We believe that both of these properties will make SafeErlang very competitive.

In this report I have shown that access control can be integrated into Erlang's run-time system without too much performance loss and without changing the semantics of Erlang too much. This implementation *enforce* access control which PoSE and SSErl only simulate. This was the main goal with this implementation, but almost as important as enforcement of access control is the performance. The chosen design of capabilities makes it possible for the run-time system to efficiently manage capabilities which is essential for the performance of the system. The implementation is neither flawless nor complete and it needs improvements, but it demonstrates all the important aspects of how to integrate efficient access control enforcement in Erlang.

Most of the ideas of how capabilities should be used to enforce access control have been worked on earlier when the prototypes were constructed. Some important new ideas have been introduced by this work, though; for example, process specific default node identifiers, a new design of user capabilities, reincarnation of entities and installation of protection. I believe that user capabilities should be the basic building block for building resource managers with access control in SafeErlang and that the chosen design of user capabilities supports that very well.

Most of the ideas of how access control should be integrated into Erlang's run-time system have been outlined during this work. The solutions we have come up with are good, but they could probably be improved in many ways.

Finally, as the sub-title implies, we definitely have approached a real SafeErlang implementation.

References

[A97] A. W. Appel. 1997. *Modern Compiler Implementation in C.* Chapter 13, p. 257-282. ISBN 0-521-58653-4. Cambridge: Cambridge university press.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, M. Williams. 1996. *Concurrent Programming in Erlang.* ISBN 0-13-508301-X. Hertfordshire: Prentice-Hall Europe.

[B97] L. Brown. 1997. *SSErl - Prototype of a Safer Erlang.* <http://www.adfa.edu.au/~lpb/papers/tr9704.html>. Accessed 28 October 1999.

[B99] L. Brown. 1999. *Custom Safety Policies in SafeErlang.* <http://www.adfa.edu.au/~lpb/research/sserl/sspol99.html>. Accessed 28 October 1999.

[BCK97] M. Bellare, R. Canetti, H. Krawczyk. *RFC 2104, HMAC: Keyed-Hashing for Message Authentication.* <http://www.it.kth.se/docs/rfc/rfcs/rfc2104.txt>. Accessed 12 January 2000.

[BS99] L. Brown, D. Sahlin. 1999. *Extending Erlang for Safe Mobile Code Execution.* <http://www.ericsson.se/cslab/~dan/reports/sserl99/sserl99.ps> Accessed 28 October 1999.

[CG97] P. Cheng, R. Glenn. 1997. *RFC 2202, Test Cases for HMAC-MD5 and HMAC-SHA-1.* <http://www.it.kth.se/docs/rfc/rfcs/rfc2202.txt>. Accessed 12 January 2000.

[CJ97] R. Chow, T. Johnson. 1997. *Distributed Operating Systems & Algorithms.* Chapter 8, p. 265-318. ISBN 0-20149838-3. Addison-Wesley.

[EDE97] EDE Team. 1997. *Erlang 4.4 Extensions.* <http://www.erlang.org/download/extensions-4.4.pdf>. Accessed 17 November 1999.

[G89] L. Gong. 1989. *A Secure Identity-Based Capability System.* <http://java.sun.com/people/gong/papers/cap.ps.gz>. Accessed 4 November 1999.

[G98] L. Gong. 1998. *Secure Java Class Loading.* <http://java.sun.com/people/gong/papers/ieeeic98.pdf>. Accessed 18 October 1999.

[GMPS97] L. GONG, M. MUELLER, H. PRAFULLCHANDRA, R. SCHEMERS. 1997. *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2.* <http://java.sun.com/people/gong/papers/jdk12arch.ps.gz>. Accessed 12 October 1999.

[GS98] L. GONG, R. SCHEMERS. 1998. *Implementing Protection Domains in the Java Development Kit 1.2.* <http://java.sun.com/people/gong/papers/jdk12impl.ps.gz>. Accessed 18 October 1999.

[JAVA99] *Secure Computing with Java: Now and in the Future.* <http://java.sun.com/marketing/collateral/security.html>. Accessed 5 October 1999.

[K00] B. KARLSSON. *Secure Distributed Communication in SAFEERLANG.* <http://www.ericsson.se/cslab/~bertil/thesis.html>. Accessed 27 June 2000.

[LOW97] J. Y. LEVY, J. K. OUSTERHOUT, B. B. WELCH. 1997. *The Safe-Tcl Security Model.* <http://www.sun.com/research/techrep/1997/smli_tr-97-60.ps>. Accessed 8 November 1999.

[N97] G. NAESER. 1997. *SAFEERLANG.* <http://www.erlang.se/sure/core/club/university/xjobb/0109-naeser.ps.gz>. Accessed June 1999. Now available from <http://www.erlang.se/publications/xjobb/0109-naeser.ps.gz>

[P97] C. P. PFLEEGER. 1997. *Security in Computing.* ISBN 0-13-185794-0. Upper Saddle River: Prentice-Hall.

A External Capability Format

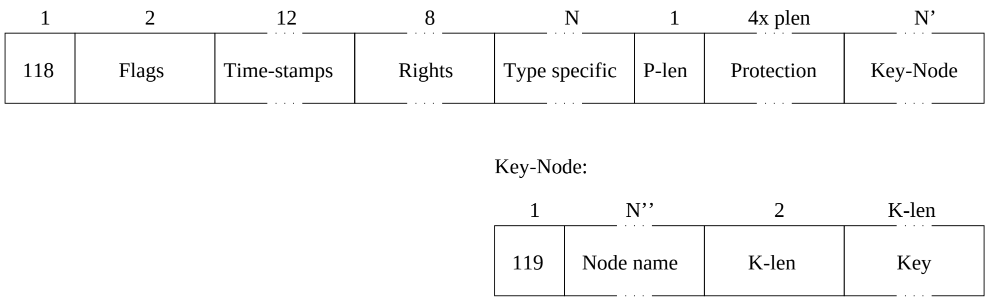

Figure 6: The external capability format.

Figure 6 shows the external capability format, i.e., the serialized format. The numbers presented above the rectangles representing the different parts of the serialized capability show how many bytes the specific parts require. A short description of every part of the format follows:

118 Integer identifying that a serialized capability follows.

Flags Flags from `OCapa` element stored as a two byte big-endian integer.

Time-stamp Time-stamp from `OOCapa` element stored as three four byte big-endian integers. The last four byte integer is currently unused and is always zero.

Rights Rights from `OCapa` element stored as two four byte big-endian integers. The last four byte integer is currently unused and is always zero.

Type specific Unused for process identifiers, port identifiers, and references. For user capabilities a serialized tuple representing the user defined rights or the serialized representation of an empty list if no rights are defined. If the user capability contains an attachment, the user defined rights are followed by a serialized attachment. For node identifiers a serialized atom representing a subnode. This is currently always the atom `'-'`.

P-len The number of four byte integers of protection value that follows.

Protection *P-len* number of four byte big-endian integers representing the protection value of a capability.

Key-Node Serialized representation of a `KeyNode` element.

119 Integer identifying that a serialized `KeyNode` follows.

Node name A serialized atom representing the node name.

K-len The number of bytes of the key that follows.

Key *K-len* number of bytes representing the public key.

B Textual Capability Format

```
<TYPE.TS1.TS2.RIGHTS.NODENAME[.attachment:ATTACHMENT]
 [.protection:PROTECTION][.key:KEY]>
```

`[.argument:VALUE]` Optional part.

`TYPE` The type of the capability, i.e., one of `Pid`, `Port`, `Nid`, `Ref` or `User capa`.

`TS1` Most significant time-stamp expressed as an integer.

`TS2` Least significant time-stamp expressed as an integer.

`RIGHTS` Rights of the capability expressed as atoms separated by white space. Rights that have been removed are marked with the atom `'-'`.

`NODENAME` The node name as an atom.

`ATTACHMENT` The external format of the attachment expressed as a hexadecimal value. This part is only used by user capabilities with attachments.

`PROTECTION` The protection value expressed as a hexadecimal value. This part is normally used, but nodes aren't required to attach protection values to the capabilities that they issue, so capabilities without protection values can exist[55].

`KEY` The public key expressed as a hexadecimal value. This part always has to be supplied when used on distributed nodes; that is, nodes that can communicate with other nodes. On non-distributed nodes this part isn't required.

All hexadecimal values are written with the most significant half byte as the left most hexadecimal character and the least significant half byte as the right most hexadecimal character.

An example, a process identifier with `send` and `info` rights:

```
<Pid.15532746.2135000.
 '-' '-' '-' '-' '-' '-' '-' send info '-' '-'.
 'rickard@tequila.du.uab.ericsson.se'.
 protection:a36 ... 4b3.key:4f6 ... 90d>
```

[55]The built-in function `node/1` takes a capability and returns a node identifier corresponding to the node on which the supplied capability was issued. This node identifier will not contain any rights nor have any protection value attached to it.

C SafeErlang Capability Rights

Pre-defined set of rights contained in process identifiers:

`link` Permission to link to the referred process.

`monitor` Permission to monitor the referred process.

`down` Permission to send "down messages" to the referred process. A "down message" is sent when a monitored process terminates.

`exit` Permission to send exit signals to the referred process. These signals can be trapped by the referred process.

`kill` Permission to send kill signals to the referred process. This signal cannot be trapped by the referred process.

`send` Permission to send messages to the referred process.

`info` Permission to get information about the referred process.

`group_leader` Permission to set group leader of the referred process.

`trace` Permission to trace the referred process.

Pre-defined set of rights contained in port identifiers:

`link` Permission to link to the referred port.

`monitor` Permission to monitor the referred port.

`close` Permission to close the referred port.

`send` Permission to send messages to the referred port.

Pre-defined set of rights contained in node identifiers:

`register` Permission to register processes on the referred node.

`unregister` Permission to unregister processes on the referred node.

`registered` Permission to view registered processes on the referred node.

`registered_send` Permission to send messages to registered processes on the referred node. One is always allowed to send messages to `net_kernel` and `capa_server` even if one hasn't got this right.

`halt` Permission to halt the referred node.

`load_module` Permission to load modules on the referred node.

`purge_module` Permission to purge modules on the referred node.

`delete_module` Permission to delete modules on the referred node.

`info` Permission to view information about the referred node.

`reincarnate` Permission to reincarnate entities on the referred node.

`install_protection` Permission to install protection of capabilities on the referred node.

`processes` Permission to inspect which processes that run on the referred node.

`spawn` Permission to spawn processes on the referred node.

`open_port` Permission to open ports on the referred node.

`priority` Permission to set scheduling priority of processes on the referred node.

`trap_exit` Permission to trap exits on the referred node. Only default nids are inspected.

`system_flag` Permission to set system flags on the referred node. Only default nids are inspected.

`db` Permission to use database functionality on the referred node. Only default nids are inspected.

Pre-defined set of rights contained in references:
Cannot contain any rights.

Pre-defined set of rights contained in user capabilities:
No pre-defined rights; the rights contained in a user capability is defined by the user on creation of the capability.

D New Built-in Functions and Guards

New built in functions:

`make_capa/[1-3]` Arguments: `RightsList`, `Attachment`, `ProtectionTuple`
: Creates a user capability. Argument `Attachment` and `ProtectionTuple` are optional. `RightsList` is a list of atoms which corresponds to the rights that the newly created capability should contain. `Attachment` is a SafeErlang term that will be attached to the newly created capability. `ProtectionTuple` is a tuple of the form `{PType, PKey}` where `PType` is the protection type to use and `PKey` is the protection key to use. If the argument `ProtectionTuple` is used, protection corresponding to the argument will be installed for the newly created capability; otherwise, the default protection will be used. Returns and newly created user capability.

`write_capa/2` Arguments: `FileName`, `Capability`.
: Writes the capability `Capability` on textual capability format to a file with the file name `FileName`.

`read_capa/1` Arguments: `FileName`.
: Reads a capability on textual capability format from a file with the file name `FileName`.

`restrict/[2-3]` Arguments: `Capability`, `RightsList`, `Timeout`.
: Returns a restricted copy of the capability `Capability. RightsList` is a list of atoms corresponding to the rights that the newly restricted capability should contain. The rights contained in the newly restricted capability will be the intersection of the rights contained in the capability `Capability` and the rights supplied in `RightsList`. `Timeout` is optional and sets a timeout time on the restrict operation.

`list_to_capa/1` Arguments: `CapabilityList`.
: Returns a capability corresponding to the capability on textual capability format in `CapabilityList`.

`capa_to_list/1` Arguments: `Capability`.
: Returns a list containing the capability `Capability` on textual capability format.

`capa_to_display_list/1` Arguments: `Capability`.
: Returns a list containing the capability `Capability` on short textual capability format.

`capa_to_verbose_display_list/1` Arguments: `Capability`.
: Returns a list containing the capability `Capability` on medium textual capability format.

`is_capa/1` Arguments: `Item`.
: Returns `true` if `Item` is a capability, otherwise `false`.

`is_nid/1` Arguments: `Item`.
Returns `true` if `Item` is a node identifier, otherwise `false`.

`is_user_capa/1` Arguments: Item.
Returns `true` if `Item` is a user capability, otherwise `false`.

`rself/0`
Returns a pre-restricted version of the process identifier of the calling process. The only rights contained in it are `send` and `info` rights.

`rnode/0`
Returns a pre-restricted version of the default nid of the calling process. If the default nid contained an `info` right it will contain an `info` right; otherwise, it will not contain any rights at all.

`attachment/1` Arguments: `UserCapability`.
`UserCapability` is a user capability. Returns the attachment of `UserCapability` if it contains an attachment, otherwise the atom `no_attachment`.

`rights/1` Arguments: `Capability`.
Returns the rights contained in the capability `Capability` as a list of atoms representing the contained rights.

`node_name/1` Arguments: `Capability`.
Returns the node name of the node of origin of the capability `Capability`.

`validate/[1-2]` Arguments: `Capability`, `Timeout`.
Returns `true` if the capability `Capability` is valid, otherwise `false`. `Timeout` is optional and sets a timeout time on the validate operation.

`has_valid_right[2-3]` Arguments: `Capability`, `Right`, `Timeout`.
Returns `true` if the capability `Capability` is valid and contains the right `Right`, otherwise `false`. `Timeout` is optional and sets a timeout time on the has valid right operation.

`install_protection/2` Arguments: `Nid`, `CapaAndProtection`.
Installs protection for a capability. `Nid` is a local node identifier to the node on which the install protection operation is performed. `Nid` is required to contain a `install_protection` right. `CapaAndProtection` is a tuple of the form `{Capa, PType, PKey}` where `Capa` is the capability for which the protection will be installed, `PType` is the type of protection to use, and `PKey` is the key to use.

`go_live/1` Arguments: `ArgList`.
Replaces `setnode/2` and is *only* used by the run-time system itself. `ArgList` is a list of tuples of size two. Every tuple contains an argument which is an atom and a value which corresponds to the argument. Valid arguments are: `node_name`, `protection_type`, `protection_key_file`, `asymmetric_keys_file`, and `installed_nid`.

New guards:

`capa/1` Arguments: `Item`.
  Succeeds if `Item` is a capability.

`nid/1` Arguments: `Item`.
  Succeeds if `Item` is a node identifier.

`user_capa/1` Arguments: `Item`.
  Succeeds if `Item` is a user capability.

E A SafeErlang Programming Example

This example is taken from the Amoeba operating system which uses capabilities to enforce access control on entities. The core of the file system in Amoeba consists of a server named bullet server which takes care of the actual storage of files. I have taken this server as an example and implemented a (quite heavily) modified version of it in SafeErlang. The bullet server stores files as immutable, i.e., one cannot modify files. Instead one has to replace a file with another if one wants to update a file. The reader doesn't have to know anything about Amoeba to understand the example, but it helps to know some Erlang and SafeErlang.

Files in this example are stored as lists, but the example could easily be modified to use real file operations instead of list operations.

```
%%% File    : bullet_server.erl
%%% Author  : Rickard Green <rickard.green@uab.ericsson.se>
%%% Purpose : To show the great capabilities of user capabilities :-).
%%% Created : 7 Feb 2000 by Rickard Green <rickard.green@uab.ericsson.se>

-module('bullet_server').
-author('rickard.green@uab.ericsson.se').

%% A file server (bullet server) is referred to by a master capability which
%% is a user capability with tailored rights for this purpose.
%%
%% All files are referred to by file capabilities which are user capabilities
%% with tailored rights for this purpose.
%%
%% Both master and file capabilities contain an attachment of the form:
%%               {FileServerPid, FileSize, FileName}
%% FileSize and FileName are unused in master capabilities.

-define(TIMEOUT, 10000).

-export([create_file/3, read_file/1, delete_file/1, file_size/1, file_name/1,
         file_status/1, shutdown/1, start/0, start/2, bullet_loop/2]).

%% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %%
%%                         Client interface functions                        %%
%%                                                                           %%

%% file_name/1
%% FileCapa has to be a file capability.
%% Returns the file name of the file referred to by FileCapa.
file_name(FileCapa) ->
    case catch attachment(FileCapa) of
        {_, _, FileName} when atom(FileName) ->
            file_name;
        _ ->
            throw({file_name, invalid_file_capa})
    end.
```

```
%% file_size/1
%% FileCapa has to be a file capability.
%% Returns the file size of the file referred to by FileCapa.
file_size(FileCapa) ->
    case catch attachment(FileCapa) of
        {_, FileSize, _} when integer(FileSize) ->
            FileSize;
        _ ->
            throw({file_size, invalid_file_capa})
    end.

%% create_file/3
%% Creates a file with the name FileName and with the content FileContent.
%% CreationCapa has to be a master capability containing a create_file right.
%% Returns a file capability referring to the newly created file.
create_file(FileName, FileContent, CreationCapa) when user_capa(CreationCapa) ->
    case catch attachment(CreationCapa) of
        {FileServer, _, _} when pid(FileServer) ->
            Tag = make_ref(),
            FileServer ! {create_file,
                          rself(),
                          Tag,
                          CreationCapa,
                          FileName,
                          FileContent},
            receive
                {new_file, Tag, FileCapa} ->
                    FileCapa;
                {error, Tag, Reason} ->
                    throw({create_file_error, Reason})
            after ?TIMEOUT ->
                    throw({create_file_error, timeout})
            end;
        _ ->
            throw({create_file, invalid_creation_capa})
    end.

%% read_file/1
%% FileCapa has to be a file capability containing a read_file right.
%% Returns the file content of the file referred to by FileCapa.
read_file(FileCapa) when user_capa(FileCapa) ->
    case catch attachment(FileCapa) of
        {FileServer, _, _} when pid(FileServer) ->
            Tag = make_ref(),
            FileServer ! {read_file, rself(), Tag, FileCapa},
            receive
                {file_content, Tag, FileContent} ->
                    FileContent
            after ?TIMEOUT ->
                    {read_file_error, timeout}
            end;
        _ ->
            throw({read_file, invalid_file_capa})
    end.
```

```
%% delete_file/1
%% FileCapa has to be a file capability containing a delete_file right.
%% Returns the atom deleted after successful deletion.
delete_file(FileCapa) when user_capa(FileCapa) ->
    case catch attachment(FileCapa) of
        {FileServer, _, _} when pid(FileServer) ->
            Tag = make_ref(),
            FileServer ! {delete_file, rself(), Tag, FileCapa},
            receive
                {deleted, Tag} ->
                    deleted
            after ?TIMEOUT ->
                    {delete_file_error, timeout}
            end;
        _ ->
            throw({delete_file, invalid_file_capa})
    end.

%% file_status/1
%% MasterCapa has to be a master capability referring to a file service and
%% has to contain a file_status right.
%% Returns all file information of the service referred to by MasterCapa.
file_status(MasterCapa)  ->
    case catch attachment(MasterCapa) of
        {FileServer, _, _} when pid(FileServer) ->
            Tag = make_ref(),
            FileServer ! {file_status, rself(), Tag, MasterCapa},
            receive
                {file_status, Tag, FileStatus} ->
                    FileStatus
            after ?TIMEOUT ->
                    {file_status_error, timeout}
            end;
        _ ->
            throw({file_status_error, invalid_capa})
    end.

%% shutdown/1
%% Shuts down a file server.
%% MasterCapa has to be a master capability referring to a file service and
%% has to contain a shutdown right.
%% Returns the atom exiting if the shutdown was successful.
shutdown(MasterCapa)  ->
    case catch attachment(MasterCapa) of
        {FileServer, _, _} when pid(FileServer) ->
            Tag = make_ref(),
            FileServer ! {shutdown, rself(), Tag, MasterCapa},
            receive
                {exiting, Tag} ->
                    exiting
            after ?TIMEOUT ->
                    {shutdown_bullet_server_error, timeout}
            end;
        _ ->
            throw({shutdown_bullet_server, invalid_capa})
    end.
```

```
%% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %% %%
%%                           Server implementation                            %%
%%                                                                            %%

%% make_pid/0
%% Auxiliary function.
%% Returns a dead process identifier (which can be reused).
make_pid() ->
    OldValue = process_flag(trap_exit, true),
    Pid = spawn_link(fun() -> bye end),
    receive
        {'EXIT',Pid,_} ->
            continue
    end,
    process_flag(trap_exit, OldValue),
    Pid.

%% start/0
%% Starts up an empty file server.
%% Returns a master capablity, with all rights set, that refers to the newly
%% created file server.
start() ->
    start([], make_capa([create_file,
                         file_status,
                         shutdown],
                        {make_pid(), 0, no_file})).

%% start/2
%% Restarts the file server referred to by MasterCapa. FileStatus contains
%% all of the file information which the server should be started with. The
%% FileStatus data has typically been fetched by a call to the function
%% file_status/1 when the service previously was up.
%% Returns MasterCapa.
start(FileStatus, MasterCapa) ->
    {Pid, _, _} = attachment(MasterCapa),
    erlang:spawn_opt({bullet_server,
                      bullet_loop,
                      [FileStatus, MasterCapa]},
                     [{install_pid, Pid}]),
    MasterCapa.
```

```
%% bullet_loop/2
%% The server loop.
%% FileStatus contain all current file information and MasterCapa is the
%% master capability referring to this service.
bullet_loop(FileStatus, MasterCapa) ->
    receive

        {create_file,
         Client,
         Tag,
         CreationCapa,
         FileName,
         FileContent} when MasterCapa == CreationCapa, pid(Client) ->
        %
        % Got a create file request. Serving this request ...
        %
            case catch has_valid_right(CreationCapa, create_file) of
                true ->
                    case lists:keysearch(FileName, 2, FileStatus) of
                        false ->
                            FileCapa =
                                make_capa([read_file, delete_file],
                                          {rself(),
                                           length(FileContent),
                                           FileName}),
                            Client ! {new_file, Tag, FileCapa},
                            bullet_loop([{FileCapa, FileName, FileContent}
                                         | FileStatus],
                                        MasterCapa);
                        _ ->
                            Client ! {error, Tag, used_file_name}
                    end;
                _ ->
                    ignore_invalid_request
            end;
```

```
{read_file, Client, Tag, FileCapa} when pid(Client) ->
%
% Got a read file request. Serving this request ...
%
    case catch attachment(FileCapa) of
        {_, _, FileName} ->
            case catch lists:keysearch(FileName, 2, FileStatus) of
                {value,
                 {Capa, _, FileContent}} when Capa == FileCapa ->
                    case catch has_valid_right(FileCapa,
                                               read_file) of
                        true ->
                            Client ! {file_content,
                                      Tag,
                                      FileContent};
                        _ ->
                            ignore_invalid_request
                    end;
                _ ->
                    ignore_invalid_request
            end;
        _ ->
            ignore_invalid_request
    end;

{delete_file, Client, Tag, FileCapa} when pid(Client) ->
%
% Got a delete file request. Serving this request ...
%
    case catch attachment(FileCapa) of
        {_, _, FileName} ->
            case catch lists:keysearch(FileName, 2, FileStatus) of
                {value,
                 {Capa, _, FileContent}} when Capa == FileCapa ->
                    case catch has_valid_right(FileCapa,
                                               delete_file) of
                        true ->
                            Client ! {deleted, Tag},
                            bullet_loop(lists:
                                        keydelete(FileName,
                                                  2,
                                                  FileStatus),
                                        MasterCapa);
                        _ ->
                            ignore_invalid_request
                    end;
                _ ->
                    ignore_invalid_request
            end;
        _ ->
            ignore_invalid_request
    end;
```

```
{file_status, Client, Tag, Capa} when MasterCapa == Capa, pid(Client) ->
        %
        % Got a file status request. Serving this request ...
        %
            case catch has_valid_right(Capa, file_status) of
                true ->
                    Client ! {file_status, Tag, FileStatus};
                _ ->
                    ignore_invalid_request
            end;


{shutdown, Client, Tag, Capa} when MasterCapa == Capa, pid(Client) ->
        %
        % Got a shutdown request. Serving this request ...
        %
            case catch has_valid_right(Capa, shutdown) of
                true ->
                    Client ! {exiting, Tag},
                    exit(normal);
                _ ->
                    ignore_invalid_request
            end;
        _ ->
            ignore_invalid_request
    end,
    bullet_loop(FileStatus, MasterCapa).
```