Security Analysis of Private Data Enquiries in Erlang

Florian Kammüller
Technische Universität Berlin
Software Engineering Group
flokam@cs.tu-berlin.de

Reiner Kammüller
Universität Siegen
Fakultät für Elektrotechnik und Informatik
reiner.kammuller@gmail.com

Abstract

Privacy is an issue of increasing concern to the Internet user. To ensure the continued success of distributed information systems, a reliable information flow must be established in certified but immediately evident ways. We begin with basic consideration of the privacy problem in the general setting of database enquiries. From there, we develop a simple solution, which we illustrate with a simple implementation in the programming language Erlang. We first provide an informal security analysis that is then developed into a formal definition of a type system for noninterference.

Index Terms

Privacy; distribution; noninterference; type systems

1. Introduction

Privacy has become an important issue in public e-business. In order to protect their customers, commercial services have to provide electronic privacy, which is approximated by anonymity using pseudonyms. However, it has been long known that chaining transactions quickly reveals the identities behind pseudonyms. Even more crucially, applications that appear secure from a superficial point of view may well contain numerous covert channels. Some of these covert channels – the ones inherent in the logic of programs – can be identified by a painstaking information flow analysis [10]. Such an analysis verifies a formal notion of security over different data domains, so-called noninterference [12], for all possible control flow of programs. Even without this classical but cumbersome method, some kind of formal language would appear to be necessary for a thorough analysis of security risks.

Several formal languages have been proposed to encode privacy policies. The Platform for Privacy Preferences (P3P) is just one example of a language that enables enterprises to communicate their privacy policies to customers. The customers may then decide whether they are willing to accept a policy prerequisite for their database enquiry. Apparently, even with means such as P3P, it is not easy to determine whether in-house enforcement policies meet their published P3P privacy promises [4].

In this paper, we first provide a simple formal specification of an obvious requirement for such privacy promises illustrating that it is virtually impossible to expect such policies to work. From this, we devise a simple idea of a different database enquiry that achieves privacy. We illustrate this solution using a prototype in the parallel programming language Erlang (Section 2). Efficiency is the price to pay for the privacy gained. We further illustrate how parallelism in Erlang helps to overcome this drawback. We then justify our claim with an informal security argument (Section 3). Once given the intuition how such an analysis of information flow works we introduce the formal approach to noninterference. We further define a type system that enables the static analysis of Erlang programs for covert channels (Section 4). Finally, we briefly contrast our Erlang solution for private data searches with alternative approaches using active objects or Java, and offer our conclusions (Section 5). This paper is an extended version of an earlier conference paper [19]: the formal approach to security by an original noninterference type system for Erlang contained in Section 4 is novel.

1.1. Privacy Policies

The enforcement of privacy policies within an enterprise constitutes an interesting problem in itself. However, if we ignore for a moment the actual implementation issue and try to establish a precise requirement specification for some of the problems involved, we can identify *data retention* as conflicting with privacy. By retention, we mean the requirement that user data provided for the identification of services only be retained a specified period after which the data must no longer be stored in the enterprise's database.

Formally, we can identify two operations *copy* and *delete* simply denoting that a data item is copied at the enterprises site and that it is deleted in order to regain some privacy. We assume the following algebraic properties of *copy*, *delete*,

and *run*, a process representing all possible behaviours.

$$\begin{aligned} copy; delete &= id \\ copy; run &= run; copy \end{aligned}$$

Using a specification formalism like CSP [14], we could now specify what is meant by the fact that a system P *does not retain data* d for any alphabet A of possible system events, as follows.

$$copy(d); run(A \setminus copy(d)); delete(d) \sqsubseteq P$$

Here, the refinement order relation $\sqsubseteq$ constrains the behaviour of P in that the specification *spec* on the left-hand side is only implemented by such processes P that implement a behaviour contained in *spec*.

The expression *run(A $\setminus$ copy(d))* specifies that between a *copy(d)* and the corresponding action *delete(d)* any sequence of events of A may happen, except another copy of d. The specification thus prohibits excessive copies and hence unauthorized retention of data d.

The interesting question is whether we can guarantee such a behaviour. In principle, the answer is yes if we can observe every sequence of actions in a server of which we require a service. Pondering this for a moment, we realize that the above retention specification is unrealistic for real-world scenarios: no service will lay open all its internal action traces.

Starting from this discouraging – but highly compelling insight – we develop a different type of database enquiry that differs from the usual service architecture model. Instead of disclosing our incentives, i.e. private data, we perform the kernel action on the data offered by a service ourselves. Clearly, there will be a loss of efficiency but we will gain security. Since the data we wish to keep private is contained in our kernel service action, the service provider has no access to it.

There are cryptographic schemes addressing similar problems. The most general, oblivious transfer, by Rabin [30], follows ideas similar to the original "Conjugate coding" by Stephen Wiesner now so popular through quantum cryptography. The scheme of private information retrieval [7] is closely related. This scheme abandons perfect secrecy for the sake of efficiency – the solution protects against attackers bound by complexity theory.

The approach we investigate here is the only one that guarantees privacy in an information theoretic sense but is deemed "practically unacceptable" because of the communication overhead [7]. We illustrate this approach in Erlang and show how massive parallelization may be used to minimize the effort.

2. An Erlang Implementation of Database Enquiries

A database enquiry is a service that is usually provided by a server through the transfer of a search key to the server, e.g. Google. In the respective service the server performs a search action on the data, e.g. the Internet, that is in its data domain. Unfortunately, this efficient standard solution implies that we must trust the server not to make unauthorized copies of our search key, i.e. the private data we wish to keep confidential. For example, we might need to input our name, address and some incentive in order to find the required services in our neighbourhood.

Instead of disclosing our personal information, we can demand access to some larger relevant data domain and perform the selection, i.e. the search corresponding to our profile or key, in our private secure domain. We will illustrate this type of database enquiry on a concrete implementation in the parallel programming language Erlang [1]. We begin with a short introduction to Erlang.

2.1. Erlang

The programming platform Erlang/OTP provides the infrastructure for programming open distributed telecommunication (OTP) systems. The language Erlang [1] was developed by the Ericsson corporation to address the complexity of developing large-scale programs within a concurrent and distributed setting. The platform Erlang/OTP consists of the functional language Erlang – with support for communication and concurrency – and the OTP middleware.

The most important features of Erlang include the following.

- Erlang variables are immutable: their value is assigned once only; no multiple assignments are allowed.
- Erlang processes do not share memory space; interaction is through explicit message passing.
- Erlang's process creation speed is much faster than the operating system's processes, much like thread creation [1][Section 8.4].

The programming style of Erlang resembles that of the ML language [28]. Recursive functions may be defined in a fairly intuitive way. For example, the factorial function is defined as follows.

```
fact 0 -> 1;
fact N -> N * fact(N-1).
```

Processes may be created by the `spawn` command, which takes the processes' function and initial arguments as parameters. The value of a `spawn` command is the process identifier `Pid` of the created process. Message passing between parallel processes is, for sending, simply written as `Pid ! message` – in our example the process identifier `Pid`. Reception of messages in processes is organized through a

mailbox in each process that can be read by the `receive` command. Using pattern matching, `receive`-statements can be written concisely and elegantly. The main data types are (untyped) lists and records, e.g. {`green, apple`}. Any lower-case name is interpreted as a constant, and higher-case names are variables. These various language features are used below when considering our database enquiry.

2.2. A Simple Database Enquiry

To simplify matters, we assume that the database is a file of already structured data. We do so to focus our attention on the communication necessary for the enquiry, leaving out the complexity of a realistic data analysis. In brief, the basic database enquiry program implements a server providing the database and our privacy-aware client that orders the data and performs a search on it. We explicitly model the server to provide a basis for the subsequent security analysis. To model a real world scenario, we provide simple programs for these two components. Later, we will see how we can improve the system through parallelization to enhance performance.

The server listens on a port for the opening of a socket and accepts the socket. After accepting it, the server closes the listening socket which does not affect the existing connection but merely prevents new connections. The Erlang package `gen_tcp` optimally supports the implementation of such distributed systems based on the tcp-protocol. We omit some parameters so as not to overload the exposition. The complete program code can be downloaded from the authors' website [1].

```
start_server() ->
  {ok, Listen} = gen_tcp:listen(2345, ...),
  {ok, Socket} = gen_tcp:accept(Listen),
  gen_tcp:close(Listen),
  loop(Socket).
```

The `loop` procedure repeatedly reads data units from the database accessible to the server. To facilitate the example, databases are simply represented as files. The socket is opened and closed by the client. The server opens the database, represented by the file specified by the client, and delegates processing of the stream transfer to the `send_stream` procedure.

```
loop(Socket) ->
   receive
     {tcp, Socket, FileB} ->
        FileS = binary_to_term(FileB),
        {ok, S} = file:open(FileS, read),
        ok = send_stream(Socket, S),
        loop(Socket);
     {tcp_closed, Socket} -> ok
   end.
```

1. http://www.swt.cs.tu-berlin.de/~flokam/research

The data is sent by the procedure `send_stream` to the socket in a repeated read action from the opened file stream `S` until end of file `eof` is reached.

```
send_stream(Socket, S) ->
  case io:read(S, '') of
    {ok, X} ->
       gen_tcp:send(Socket, term_to_binary(X)),
       send_stream(Socket, S);
    eof     ->
       file:close(S),
       gen_tcp:send(Socket, term_to_binary(eof)),
       ok
  end.
```

The client now opens the socket and transmits the database we wish to investigate, represented by a file. Here, we use a generic name `host`, representing some actual hostname. The actual reception of the file's contents is delegated to the procedure `client_receive`. The search results are returned by this procedure as a result list `Res` and are immediately output.

```
client_eval(Key, FileS) ->
 {ok, Socket} =
   gen_tcp:connect("host", 2345, ...),
 ok = gen_tcp:send(Socket, term_to_binary(FileS)),
 {eof, Res} = client_receive(Key,Socket,self(),[]),
 io:format("Client result: ~p~n", Res),
 gen_tcp:close(Socket).
```

The database's contents arrive at the client and are immediately analyzed corresponding to the search key `Key`. The actual data analysis is, for clarity's sake, reduced to a simple pattern matching on the received data items. Only matching contents are assembled in the result list `Res`.

```
client_receive(Key, Socket, From, Res) ->
 receive
  {tcp, Socket, Bin} ->
    Val = binary_to_term(Bin),
     case Val of
      eof      -> From! {eof, Res};
      {Key, X} ->
       client_receive(Key, Socket, From, [X|Res]);
      Any      ->
       client_receive(Key, Socket, From, Res)
    end
 end.
```

Two processes, one for the server and one for the client, can now be started independently by compiling the code presented above on two separate sites running Erlang. Invoking the function `start_server()` on the first, the server's site, while calling `client_eval(key,"file.dat")` on the client site has the following effect on the latter

```
Client result: "Ottostr 38, 10999 Berlin"
ok
```

where the key was `drugstore` and `file.dat` contains, amongst other arbitrarily structured data, an item

```
{drugstore, "Ottostr 38, 10999 Berlin"}.
```

The server site only reports ok after successful termination of the process.

2.3. Efficiency by Parallelization

The simple client server introduced in the previous section represents the desired security solution but it is clearly not efficient because all data has to be transferred from the server to the client before the actual selection takes place. Generally, security does not come for free, so we can see this as the price to be paid. However, the communication overhead may constitute a crucial bottleneck in an application. One of the strong points of Erlang is the possibility to create a large number of parallel processes. To show that our approach scales up to realistic application scenarios, we present below an extension to the previous basic program which significantly enhances performance. In fact, this extension is a standard way of using Erlang. We therefore only show the extensions to the basic program presented in the previous section to explain the principle, but also go on to discuss some important practical issues.

The main clue to parallelizing the server is to start a new parallel process in start_server whenever a new connection is provided by a client through gen_tcp:accept. Note, that the listening socket is, unlike the sequential server, not closed down as we accept new connections.

```
start_par_server() ->
  {ok, Listen} = gen_tcp:listen(...)
  spawn(fun() -> par_connect(Listen) end).

par_connect(Listen) ->
  {ok, Socket} = gen_tcp:accept(Listen),
  spawn(fun() -> par_connect(Listen) end),
  loop(Socket).

loop(..) -> % as above
```

On the client site, we use the same principle to make parallel client processes each communicating with a parallel server.

```
par_client(Key, FFile) ->
  {ok, S} = file:open(FFile, read),
  ok = client_par_eval(Key, S).

client_par_eval(Key, S) ->
  case io:read(S, '') of
    {ok,FileS} ->
        spawn(fun() -> client_par_eval(Key,S) end),
        client_eval(Key, FileS);
    eof -> file:close(S),
           ok
  end.
```

The input of the file names of the files to be searched is provided by an input file FFile on the client site. The gradual selection of new source files for a goal-directed search may be integrated (see Section 5).

This parallel server can potentially create thousands of connections. Performance is thus significantly enhanced, although clearly the bandwidth of the communication channels is strained. For a more sophisticated implementation, we can limit the maximum number of simultaneous connections by simply keeping count of new connections and finished ones.

3. Informal Security Analysis

3.1. Security Assumptions

A security analysis starts with a two-sided model comprising (a) the attacker and (b) the security policy, or security goals. We cannot achieve 100% security because (a) there always is the all-powerful attacker and (b) we cannot generally achieve all security goals for all involved parties because they may conflict. Usually, when investigating privacy, we use a multilateral security model [36] that enables consideration of differing protection goals of several involved parties.

Nevertheless, we analyze the privacy of the client using a typical multi-level security model (MLS) [9] because we are, in this paper, only interested in the privacy of the client's data. We therefore assume, for the security policy, that the user – or, in our case, the client process – has a higher security level than the server side, the potential attacker. Let this security level be H, or high, for the client, and L, or low, for the server. We further extend the security policy by assuming that the local host is a secure domain, i.e. that its data and internal communication are secure. All other communication channels outside the client, and all data on the server, is assumed to be visible to the attacker.

3.2. Information Flow Security

The first to formalize information flow in a program were the Dennings [10]. The most natural way to formalize confidentiality as a property of information flow have been Goguen and Meseguer by their notion of noninterference [12]. There are quite a few different definitions of noninterference [32], mainly because it is a relation over behaviours of programs (it it sometimes characterized as a bisimulation property). Thus, the underlying computation model – leading to different notions of behaviour – results in different notions of noninterference. Without giving a formal introduction to this notion, we attempt to provide a basic understanding of it. We adopt a state-based view: program behaviour is viewed as a transition between vectors of variable values.

The basis of noninterference is a relation of *indistinguishability* of program states based on a similar relation on the program variables: high variables are all indistinguishable, but low variables are only if they have equal values. Informally, the indistinguishability between states during a program run is defined extensionally over the indistinguishability of its components, the state variables.

Given an indistinguishability relation on program states, we can say that noninterference is defined as *low*-indistinguishability. In other words, given a security policy that assigns high and low to all data variables, a program is non-interfering iff any two program runs remain *low*-indistinguishable throughout the program behaviour if they have been so from the start.

The important implication of noninterference is that the attacker, who can only read low values, is thus unable to learn anything about the values of high variables, even if he can observe different runs of the same program on different – but indistinguishable – data.

To show noninterference with respect to a given security policy in practical terms, we have to analyze all control flows of a program and ensure that there are no information flows from high to low variables. In practice, this process is often supported by a static analysis with specialized noninterference type systems [18], [32] (see Section 4).

3.3. Informal Security Analysis of Privacy-Enhancing Database Search

Although we do not yet intend to provide a formal analysis according to some notion of noninterference here, we already wish to use its essential idea in an informal argument. Let our security policy be an assignment that assigns high to the variables `Key`, `Any`, and `Res`. All other data may be assigned low; most of the variables, like `Socket`, `S`, and `FileS`, must be low because they have to be communicated between the insecure server and the confidential client. To show noninterference, we have to analyze all control flows in our program and exclude all explicit and implicit information flows from `Key`, `Any` and `Res` to any other (low) variable.

An explicit flow is either given by an assignment from one variable to another, which is impossible in Erlang as it is functional (all variables are only assigned once), or it is given by a function call, whereby a value can then be assigned as the initial value of the receiving process. The `Key` variable is passed on from `client_eval` to `client_receive`, and from `client_receive` again to itself in two separate recursive calls. In both invocations, `Key` is again assigned by pattern matching to the variable `Key`, which is also high, so there is no illegal explicit flow there. These are all explicit flows from `Key` to any other variable.

An implicit information flow is given when the control flow can branch, e.g. at an `if` statement: according to the value of the first variable, the tested variable, a second variable in one of the branches receives a value, depending on the value of the first. Again, such an implicit flow should not lead from a high to a low variable. The only possible branching of the control flow is the `case` statement in `client_receive`. Here, there are implicit flows from `Key` to `Any` and `Res`: depending whether `Val` matches `Key` one of the two "non-eof" branches is selected. Consequently, `X` – containing matching data – is added to the result list `Res` if the `Key` match is successful otherwise nothing is added to `Res`. Therefore, `Res` has to be marked as H as well. In addition, the variable `Any` must be H because – if it were L – we could work out `Key`: the difference between the original file contents and those matched with `Any` gives just those data items containing `Key` as first element: Bingo!

The variable `Any` is not used in any further function calls, so there are no explicit flows from it to any other variable. Considering, finally, the variable `Res`, we see that, here too, there are no flows from it to any low variable, neither explicit nor implicit. The final output of the value of receive by the `io:format` call must be considered secure because it happens inside the secure domain of the client and has no effect on other low variables. To summarize, the privacy-enhancing database search is non-interfering with respect to our security policy.

4. Formal Security Analysis

In this section, we use the ideas already provided in the previous section in the informal security analysis and make them formal. In detail, we present the syntax and semantics of Erlang – more precisely, a small subset of the original Erlang language, Core Erlang, sufficient for many applications, including ours. Then, we provide a novel type system that encodes the legal information flows and define a notion of indistinguishability. Finally, we show that programs that are accepted by the type system are secure. This proof implies that programs, for which a type can be inferred according to our type system and over an initial security policy, do not leak information.

In order to define formal semantics, it is always advisable to use a reduced language set cutting out, on one side, syntactic sugar, while still enabling, on the other side, the full power of the language. There have been a few attempts to define a Core Erlang language that serves exactly this purpose, e.g. [6]. Several other papers, in particular those concentrating on formal aspects of the Erlang language, e.g. [17], [25], define semantics also in a more formal way. This is a prerequisite for a formal analysis.

One way to take advantage of these earlier works is to use a rigorous translation from Erlang to some formal calculus and to exploit means for a security analysis. In [19] we proposed such a strategy for a formal analysis: use a translation of Erlang to the π-calculus [25] to represent our application in a calculus that is more easily accessible to a formal analysis. This would enable the use of existing formalizations of noninterference for the π-calculus [15] to demonstrate information flow security. However, after some consideration of information flow analysis in the π-calculus we decided to follow a different but much simpler

approach based on *language based noninterference* by Volpano, Smith, and Irvine [35], [34], [33].

In this section, we base our presentation on the syntax and semantics for Erlang following Huch [17], present, then, an original noninterference type system for this language, to prove that a well-typed program does not contain illegal flows. We finally illustrate experimentally the use of our Erlang noninterference type system by showing how it is used to check our program in a process of inferring security types based on some initial security policy.

4.1. Syntax of Core Erlang

In the presentation of the syntax and semantics of Core Erlang we directly model the asynchronous communication and the message queues of Erlang. This is in difference to some models that base the formal models of Erlang on operational models of the π-calculus or CCS, unnecessarily complicating the situation by mapping the asynchronous communication of Erlang to synchronisation in π or CCS, respectively.

The syntax of Core Erlang is defined as follows where c denotes constructors and ϕ may denote any built-in or user defined function or constructor.

$$
\begin{array}{rcl}
p & ::= & f\texttt{(}X_1,\ldots,X_n\texttt{) -> } e. \mid p\ p \\
e & ::= & \phi\texttt{(}e_1,\ldots,e_n\texttt{)} \mid X \mid \mathit{pat}\texttt{=}e \mid \texttt{self} \mid \\
 & & e_1\texttt{,}e_2 \mid e_1\texttt{!}e_2 \mid \texttt{case } e \texttt{ of } m \texttt{ end} \\
 & & \texttt{receive } m \texttt{ end} \mid \texttt{spawn (}f\texttt{,}e\texttt{)} \\
m & ::= & \mathit{pat}_1\texttt{-> } e_1\texttt{; }\ldots\texttt{;}\mathit{pat}_n\texttt{-> } e_n \\
\mathit{pat} & ::= & c\texttt{(}\mathit{pat}_1\texttt{,}\ldots\texttt{,}\mathit{pat}_n\texttt{)} \mid X
\end{array}
$$

These are the known Erlang constructors. To express the semantics we need to define also constructors for pattern matching but these we treat differently as they are only needed as an "internal representation" of the most naturally expressed pattern matching expressions used in (Core) Erlang. We distinguish three different cases for matching `match`, `casematch`, and `queuematch` for simple patterns, patterns in case statements and in queues, respectively. The semantics of these patterns is given in the Appendix.

4.2. Core Erlang Semantics

Next, we introduce the operational semantics of the language Core Erlang. Usually, that is, in all the major introductory texts, e.g. [2], [1], but also in scientific descriptions of Core Erlang, e.g. [6], [25] the semantics of Erlang is just informally described. For our purposes, this is not sufficient. Hence we need to re-engineer a formal semantics. We do so closely following Huch [17] but significantly simplifying this original contribution as we do not need the same level of detail.

The semantics, we present, follows the idea of a *structural operational semantics* which means that we define a reduction relation $\longrightarrow_{Erl}$ that gives rules for all cases of possible syntax structures of Core Erlang programs. The reduction represents possible one-step evaluations of an Erlang program. The parallel program state is represented as a set of Erlang processes Π where each single process term is a triple (p, t, q) where p is the process identifier of this process, t is the sequential Erlang term representing the process in its current state of computation, and q is the current message queue of the process. The entire operational semantics is presented in Table 1. We will explain the meaning of these rules, including special notation we use in the following explanations.

We use reduction contexts E [11] to have a succinct representation of the semantic rules and enable determining a reduction strategy. Defining the reduction contexts as follows, we define the operational semantics as a leftmost innermost operational semantics.

$$
\begin{array}{rcl}
E & ::= & [] \mid \phi\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_i, E, e_{i+2},\ldots,e_n\texttt{)} \mid E\texttt{,}e \mid \mathit{pat}\texttt{=}E \\
 & & \texttt{spawn(}f\texttt{,}E\texttt{)} \mid E\texttt{!}e \mid v\texttt{!}E \mid \texttt{case } E \texttt{ of } m \texttt{ end}
\end{array}
$$

These reduction contexts are used in the rules to identify where a reduction may take place. The nonterminal [] is called the "hole" and marks the point of the next computation. We write $E[e]$ for the context E where the hole is replaced by the term e that is reduced next. In each of the reduction rules we reduce the set of current Erlang processes Π. We introduce $\Pi\dot{\cup}(p, t, q)$ as a map represented by triples: $\Pi\dot{\cup}(p, t, q) = \Pi \cup (p, t, q)$ if $\forall\ x, y.\ (p, x, y) \notin \Pi$ else Π.

The sequential reduction rules define the evaluation inside Erlang processes. The first one specifies that a sequence of terms, juxtapositioned by comma, $v\texttt{,}e$ reduces to the second argument, if the first one is a reduced value v. Next, a sequential term can be evaluated by replacing the semantics F_A of a predefined function F by its definition. The constant `self` may be replaced by the process identifier p representing the current process. Note, that – by using `self` and assigning this semantics to it – we do not need an explicit recursion in the semantics, instead iteration is mapped to invocation of processes. Finally, the sequential rules define that a function can be replaced by its function body.

The next set of three rules defines how matchings are evaluated; the functional definitions of the matching function are contained in the Appendix. Their definitions are quite complex and for the current context it suffices to anticipate the intuitive meaning of the matching results. A simple `match` leads to a pointwise match ρ of all variables in v according to pattern *pat* and can be applied to the entire sequential program $E[v]$. Similarly, a list of matches in a `casematch` results in a matching case i and a match ρ that can be applied in a `case` clause to evaluate the

SEQUENTIAL REDUCTION

$$\frac{}{\dot{\Pi\cup}(p, E[\texttt{v , } e], q) \longrightarrow_{Erl} \dot{\Pi\cup}(p, E[e], q)} \qquad \frac{F \textit{ predefined function}}{\dot{\Pi\cup}(p, E[F(v_1, \ldots, v_n)], q) \longrightarrow_{Erl} \dot{\Pi\cup}(p, E[F_A(v_1, \ldots, v_n)], q)}$$

$$\frac{}{\dot{\Pi\cup}(p, E[\texttt{self}], q) \longrightarrow_{Erl} \dot{\Pi\cup}(p, E[p], q)} \qquad \frac{f\texttt{(}X_1\texttt{, } \ldots \texttt{,}X_n\texttt{) -> } e. \in \textit{program}}{\dot{\Pi\cup}(p, E[f\texttt{(}v_1\texttt{, } \ldots \texttt{,}v_n\texttt{)}, q) \longrightarrow_{Erl} \dot{\Pi\cup}(p, E[e[v_1/X_1\texttt{, } \ldots \texttt{,}v_m/X_m]], q)}$$

MATCHING

$$\frac{\texttt{match}\textit{(pat,v)} = \rho}{\dot{\Pi\cup}(p, E[\textit{pat} = v], q) \longrightarrow_{Erl} \dot{\Pi\cup}(p, \rho(E[v]), q)} \qquad \frac{\texttt{casematch}((\textit{pat}_1, \ldots, \textit{pat}_m), v) = (i, \rho)}{\dot{\Pi\cup}(p, E[\texttt{case } v \texttt{ of } \textit{pat}_1 \texttt{ -> } e_1\texttt{; } \ldots \texttt{;}\textit{pat}_m \texttt{ -> } e_m \texttt{ end}], q) \longrightarrow_{Erl} \Pi, (p, \rho(E[e_i]), q)}$$

$$\frac{\texttt{queuematch}((\textit{pat}_1, \ldots, \textit{pat}_m), (v_1, \ldots, v_u)) = (i, j, \rho)}{\begin{array}{l}\dot{\Pi\cup}(p, E[\texttt{receive } \textit{pat}_1 \texttt{ -> } e_1\texttt{; } \ldots \texttt{; } \textit{pat}_m\texttt{-> } e_m \texttt{ end}], (v_1, \ldots, v_j, \ldots, v_u)) \\ \longrightarrow_{Erl} \dot{\Pi\cup}(p, \rho(E[e_i]), (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_u))\end{array}}$$

CONCURRENT REDUCTION

$$\frac{f\texttt{(}X_1\texttt{, } \ldots \texttt{,}X_n\texttt{) -> } e. \in \textit{program} \text{ and } p' \text{a new pid}}{\begin{array}{l}\dot{\Pi\cup}(p, E[\texttt{spawn(}f\texttt{,} [v_1, \ldots, v_n]\texttt{)}], q) \\ \longrightarrow_{Erl} \dot{\Pi\cup}(p, E[p'], q), (p', e[v_1/X_1, \ldots, v_n/X_n], ())\end{array}} \qquad \frac{v_1 = p' \in \textit{Pid}}{\dot{\Pi\cup}(p, E[v_1\texttt{!}v_2], q), (p', e, q') \longrightarrow_{Erl} \dot{\Pi\cup}(p, E[v_2], q), (p', e, q'\texttt{@}[v_2])}$$

Table 1. Operational semantics of Core Erlang

`case` construct while simultaneously replacing the matching values in e_i. The `queuematch`, finally, produces – besides the matching – the selection of the right clause i in a `receive` construct and the selection of the message v_j in the message queue of the process which is subsequently removed from it.

For the parallel semantics, we define the semantic rules CONCURRENT REDUCTION that entail a rule for spawning a new process and one for message dispatch with `!` between processes where `@` denotes list append.

4.3. Security Type System

Given the operational semantics of the Core Erlang language we can now define a type system assigning security types $\tau \in \{L, H\}$ to Core Erlang program terms. This will then enable to prove that there are no illicit information flows in well-typed Core Erlang programs. However, this statement will be dependent on the proper behaviour of the program according to the semantics. As Erlang is not a strongly typed language we would need to first provide a classical type system and prove type safety, i.e. progress and preservation for this classical type system. As this would go well beyond the scope of this paper and, moreover, such classical type systems are provided by others, e.g. [20], [27], we will just use these results without further introduction by implicitly assuming the additional hypothesis, "program p is classically well-typed".

Our security type system is a language based type system assigning L (low) and H (high) security types to terms of Core Erlang programs. We want to assign these classes to the parameters and results of Erlang processes in order to avoid explicit flows from H to L that may happen when a H process replies to a L process with `!` or when a H process spawns a process that is of class L passing H values as parameters. To forbid such flows, we can guard the corresponding program terms by type constraints to expressions, like process identifiers, and to parameters. In addition, we use the subtype relation $\leq$, i.e. $L \leq H$. To type entire terms – like a function that is being spawned – we introduce the type constructor *cmd*. Now, programs can have type H *cmd* or L *cmd*; intuitively, a program of type H *cmd* cannot transmit information to L processes. For example, if a function has type H *cmd* it can only be applied to parameters of type H. The case of a function $f : H$ *cmd* applied to an L parameter is possible because – due to subtyping $L \leq H$ – the parameter is also of type H. How do we exclude the forbidden case $f : L$ *cmd* and parameter of type H despite subtyping? In our type system, the type τ *cmd* is neither contravariant – as in other type systems [33], [5] – nor covariant; it is simply not related. Still, the flow from L to H is enabled – as pointed out above, and the flow from H to L disabled because an L *cmd* cannot be upgraded (as it would be the case with covariance). However, as Erlang is a functional language – in contrast to the simple imperative languages considered in [33], [5] – we need to be able to use arbitrary expressions as arguments. That is, terms can be supplied as arguments to functions or sent to processes. To accomodate the necessary transformation of terms of type τ *cmd* to argument terms of type τ we add a type transformation rule (C-VAL).

The typing of identifiers, like variables and simple terms,

and processes is encoded in *type maps* γ for identifiers and Γ_{proc} assigning a security type to each process. More precisely, we introduce different categories of types; the base types τ are used to assign security types to variables and constants, and the constructed types τ *cmd* to assign types to terms.

$$\begin{array}{rcl} \tau & ::= & L \mid H \\ \varsigma & ::= & \tau \mid \tau \ cmd \\ \gamma & : & Id \Rightarrow \tau \\ \Gamma_{proc} & : & Pid \Rightarrow \tau \ cmd \end{array}$$

The typing rules define inductively the typing relation $\Gamma_{proc},\ \gamma \vdash t : \rho$ where Γ_{proc}, γ are *type environments*, maps assigning types to base terms under which the actual typing $t : \rho$ is valid. Table 2 summarizes the entire type system; it is explained in the following in detail. In the type rules, we use the type variable ς as a "meta"-variable, i.e. $\varsigma \in \{L, H, L\ cmd, H\ cmd\}$. We further use the following operator $\overset{\circ}{\cup}$ that enables extension of type maps according to equalities induced by pattern matches,

$$\tau \overset{\circ}{\cup} \rho = \tau \cup (\tau \circ \rho)$$

where the operator $\circ$ is relational composition. Rules IDENT and PIDENT state that the typings encoded in the type maps Γ_{proc} and γ can be transformed into typings with $\vdash$. Simple terms may have some fixed type assigned to it – as is encoded in rule SIMPLE TERMS. Together, the first three rules provide the means to input a security policy to the type analysis. The following three rules COMPOSE, FUNCTION, and APPLICATION encode that arguments of suitable type can be plugged into the term constructors for composition, function definition and application of Erlang. The next three rules PATTERN, CASE, RECEIVE all deal with pattern matching. The assumption set ρ assigns variables to their matched terms which is included by $\overset{\circ}{\cup}$ into the type maps. If the patterns have suitable type τ, the statement using the pattern can be typed correspondingly by τ *cmd* because it will not contain a subterm lower than τ. The rules SPAWN and SEND control the communication: SPAWN demands that the arguments to a `spawn` conform to the security bound of the spawned function and SEND states that a process v_1 of level τ *cmd* can receive messages of type τ (or higher, because of rule SUBTYPE). The rule CONFIGURATION is the main rule that lifts the typing of the rules to the level of a configuration by dividing the typing to single process terms. Finally the rules BASE, REFL, C-VAL, and SUBTYPE determine the ordering on the set of types, transform *cmd*-terms to simple terms to allow terms as parameters to functions and as messages, and enable subsumption, i.e. elements of a type are also elements of its supertypes.

4.4. Indistinguishability

We provide key properties that will be used in the subsequent section to prove that well-typed programs have the noninterference property. In addition, we use this section to introduce the notion of indistinguishability that is the core of the definition of noninterference.

Lemma 4.1 (Confinement): Let $\vdash \Pi : \Gamma_{proc}$ and $(p, t, q) \in \Pi$ with $\Gamma_{proc}(p) = H cmd$. Then, $\Gamma_{proc}, \gamma \vdash t : H\ cmd$ and for all subterms t_0 and corresponding contexts E with $t = E[t_0]$ we have that $\Gamma_{proc}, \gamma \vdash t_0 : H\ cmd$, i.e. all subexpressions are H as well.

Proof: By induction over the typing rules on the structure of t. □

The following theorem is a sanity check; it shows that the types are preserved by the semantics. If they were not preserved any properties encoded in the types would be destroyed.

Theorem 1 (Subject Reduction):

$$\vdash \Pi : \Gamma_{proc} \wedge \Pi \longrightarrow_{Erl} \Pi' \Rightarrow \exists\ \Gamma'_{proc}.\ \vdash \Pi' : \Gamma'_{proc}$$

where $\Gamma_{proc} \subseteq \Gamma'_{proc}$.

Proof: The proof is by induction and a straightforward, albeit long, case analysis. □

Next, we introduce a notion of equivalence of configurations that will enable the definition of noninterference. As already explained in the previous section when introducing noninterference informally, the intuition is that an attacker cannot see any difference of H values when regarding L values. This equivalence needs to be shown over arbitrary program runs. Hence, we need to define what it means for two program states to be indistinguishable for the attacker. As we have a dynamic set of processes, represented by Pid, that needs to be compared we use a technique seen in [3] that uses typed bijections to that end.

Definition 4.2 (Typed Bijection): A typed bijection is a finite partial function σ on process identifiers p such that

$$\forall\, p : \mathrm{dom}(\Pi).\ \vdash p : T \Rightarrow \vdash \sigma(p) : T$$

(where T is given by $\Gamma_{proc}(p)$).

The intuition behind typed bijections is that $\mathrm{dom}(\sigma)$ designates all those processes that are, or have been, visible to the attacker. We cannot assume the names in different runs of programs, even for low elements, to be the same. Hence, we relate those names via a pair of bijections. These bijections are typed because they relate processes that are all of type L.

The following definition of indistinguishability uses the typed bijection in this sense. The intuitive relationship between type L and membership in $\mathrm{dom}(\sigma)$ is only later made formal by an invariant. We believe that this invariant decisively ameliorates the exposition of the proofs and the understanding of the model (compare with the proofs in Banerjee and Naumann's paper [3]).

IDENT
$$\frac{\gamma(x) = \varsigma}{\Gamma_{proc}, \gamma \vdash x : \varsigma}$$

PIDENT
$$\frac{\Gamma_{proc}(p) = \tau\ cmd}{\Gamma_{proc}, \gamma \vdash p : \tau\ cmd}$$

SIMPLE TERMS
$$\Gamma_{proc}, \gamma \vdash e : \tau$$

COMPOSE
$$\frac{\Gamma_{proc}, \gamma \vdash c_1 : \tau\ cmd\ ,\ \Gamma_{proc}, \gamma \vdash c_2 : \tau\ cmd}{\Gamma_{proc}, \gamma \vdash c_1\texttt{ , }c_2 : \tau\ cmd}$$

FUNCTION
$$\frac{\Gamma_{proc}, \gamma \cup \{X_1 : \tau, \ldots, X_n : \tau\} \vdash e : \tau\ cmd,}{\Gamma_{proc}, \gamma \vdash f\texttt{(}X_1\texttt{ , }\ldots\texttt{ ,}X_n\texttt{) -> }e. : \tau\ cmd}$$

APPLICATION
$$\frac{\Gamma_{proc}, \gamma \vdash f\texttt{(}X_1\texttt{ , }\ldots\texttt{ ,}X_n\texttt{) -> }e. : \tau\ cmd,\ \forall\ i.\ \Gamma_{proc}, \gamma \vdash v_i : \tau}{\Gamma_{proc}, \gamma \vdash e[v_1/X_1, \ldots, v_n/X_n] : \tau\ cmd}$$

PATTERN
$$\frac{\Gamma_{proc}, \gamma \vdash v : \tau,\ \Gamma_{proc}, \gamma \mathbin{\mathring{\cup}} \texttt{match}(pat, v) \vdash pat : \tau\ cmd}{\Gamma_{proc}, \gamma \vdash pat\texttt{=}v : \tau\ cmd}$$

CASE
$$\frac{\begin{array}{l}\Gamma_{proc}, \gamma \vdash v : \tau,\ \texttt{casematch}((pat_1, \ldots, pat_m), v) = (i, \rho),\\ \Gamma_{proc}, \gamma \mathbin{\mathring{\cup}} \rho \vdash pat_i\texttt{=}v : \tau\ cmd,\ \Gamma_{proc}, \gamma \mathbin{\mathring{\cup}} \rho \vdash e_i : \tau\ cmd,\ \forall i \leq m\end{array}}{\Gamma_{proc}, \gamma \vdash \texttt{case}\ v\ \texttt{of}\ pat_1\ \texttt{->}\ e_1\texttt{;}\ldots\texttt{;}pat_m\ \texttt{->}\ e_m\ \texttt{end} : \tau\ cmd}$$

RECEIVE
$$\frac{\begin{array}{l}\texttt{queuematch}((pat_1, \ldots, pat_m), (v_1, \ldots, v_u)) = (i, j, \rho),\\ \Gamma_{proc}, \gamma \mathbin{\mathring{\cup}} \rho \vdash pat_i\texttt{=}v_j : \tau\ cmd,\ \Gamma_{proc}, \gamma \mathbin{\mathring{\cup}} \rho \vdash v_j : \tau,\ \Gamma_{proc}, \gamma \mathbin{\mathring{\cup}} \rho \vdash e_i : \tau\ cmd,\ \forall i \leq m\end{array}}{\Gamma_{proc}, \gamma \vdash \texttt{receive}\ pat_1\ \texttt{->}\ e_1\ \texttt{;}\ldots\texttt{;}pat_m\texttt{->}\ e_m\ \texttt{end} : \tau\ cmd}$$

SEND
$$\frac{\Gamma_{proc}(v_1) = \tau\ cmd,\ \Gamma_{proc}, \gamma \vdash v_2 : \tau}{\Gamma_{proc}, \gamma \vdash v_1\texttt{!}v_2 : \tau\ cmd}$$

SPAWN
$$\frac{\Gamma_{proc},\ \gamma \vdash f\texttt{(}X_1\texttt{ , }\ldots\texttt{ ,}X_n\texttt{) -> }e. : \tau\ cmd,\ \forall\ i.\ \Gamma_{proc},\ \gamma \vdash v_i : \tau}{\Gamma_{proc},\ \gamma \vdash \texttt{spawn(}f, [v_1, \ldots, v_n]\texttt{)} : \tau\ cmd}$$

CONFIGURATION
$$\frac{\forall\ (p, t, q) \in \Pi.\ \Gamma_{proc}, \varnothing \vdash t : \Gamma_{proc}(p)}{\vdash \Pi : \Gamma_{proc}}$$

BASE
$$L \leq H$$

REFL
$$\varsigma \leq \varsigma$$

C-VAL
$$\frac{\Gamma_{proc}, \gamma \vdash e : \tau\ cmd}{\Gamma_{proc}, \gamma \vdash e : \tau}$$

SUBTYPE
$$\frac{\Gamma_{proc}, \gamma \vdash p : \varsigma, \varsigma \leq \varsigma'}{\Gamma_{proc}, \gamma \vdash p : \varsigma'}$$

Table 2. Security type system for Core Erlang

Definition 4.3 (Indistinguishability): An indistinguishability relation is a heterogeneous relation $\sim_\sigma$, parameterized by a typed isomorphisms σ whose differently typed subrelations are as follows.

$$\begin{array}{rcl} t \sim_\sigma t' & \equiv & t_\sigma = t' \\ p \sim_\sigma p' & \equiv & \sigma(p) = p' \\ (p, t, q) \sim_\sigma & \equiv & p \sim_\sigma p' \wedge t \sim_\sigma t' \wedge \\ (p', t', q') & & \forall\, i.q\#i \sim_\sigma q'\#i \\ \Pi_0 \sim_\sigma \Pi_1 & \equiv & \begin{array}{l}\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Pi_0) \wedge \\ \mathrm{ran}(\sigma) \subseteq \mathrm{dom}(\Pi_1) \wedge \\ \forall\, p, p'.\ p \sim_\sigma p' \Rightarrow \Pi_0(p) \sim_\sigma \Pi_1(p')\end{array} \end{array}$$

The above exploits the convention that equations involving partial functions are interpreted as false when the function is undefined. Hence, $p \sim_\sigma p'$ only if $(p, p') \in \sigma$, otherwise $\sigma(p) = \mathit{false}$.

The H part of the program is not relevant for L-indistinguishability and thus not recorded at all in the corresponding typed bijections. "H-indistinguishability" thus corresponds intuitively to "indistinguishability not defined".

The partial bijection approach is an elegant concept for specification but technically proofs become cluttered with technical detail. We explicitly mark the correspondence between type L and the domain of the isomorphism σ. The following invariant specifies this correspondence.

Definition 4.4 (Invariant):

$$p \in \mathrm{dom}(\sigma) \quad \equiv \quad \Gamma_{proc}(p) = L\ cmd$$

We write *invariant*(σ) if the configurations are clear from context.

The invariant immediately transfers to the range of σ because it is a typed bijection.

Corollary 4.5: If the invariant holds we also have the following equivalence.

$$p \in \mathrm{ran}(\sigma) \quad \equiv \quad \Gamma_{proc}(p) = L\ cmd$$

Note, that the Invariant only *specifies* this correspondence. The invariant is a tool to clarify the proof of noninterference. Its validity for given typings and pairs of configurations has to be established.

Lemma 4.6 (Initial Invariant): Given two indistinguishable initial configurations Π_0, Π_0' that are well-typed, the isomorphism σ can be constructed such that the invariant holds. These initial configurations are then indistinguishable with respect to σ, i.e. $\Pi_0 \sim_\sigma \Pi_0'$

Note, that if the initial configurations were not indistinguishable, their types could be different in which case the existence of a pair of isomorphisms fulfilling the invariant would be impossible.

4.5. A Well-typed Program is Secure

After these preparations we are able to prove the main theorem. Well-typed programs are secure. This property is a so-called *bisimulation* property, that is, a property over different

runs of a program showing that a relation is preserved by the evaluation. This property will be the indistinguishability relation. The essence of the property is that the evaluation does not change the L-indistinguishability, therefore the attacker cannot learn more by observing the program running if he could not learn anything from start.

We first prove a theorem that assumes the invariant to hold and then shows that indistinguishability is preserved. The main result is a simple corollary from this: as we can chose σ to verify the invariant for initial configurations, all reachable configurations are secure. We prove a strong version of bisimulation in which the second transition is $\longrightarrow^{01}_{\mathit{Erl}} = \mathit{id} \cup \longrightarrow_{\mathit{Erl}}$ and not just $\longrightarrow^{*}_{\mathit{Erl}}$ (as, for example in Volpano and Smith's work on noninterference of a simple multi-threaded while language [33]).

Theorem 2 (Noninterference): Let Π_0 and Π_1 be configurations such that $\Pi_0 \sim_\sigma \Pi_1$, $\vdash \Pi : \Gamma_{\mathit{proc}}$ and $\vdash \Pi' : \Gamma_{\mathit{proc}}$, and the Invariant holds. If $\Pi_0 \longrightarrow_{\mathit{Erl}} \Pi'_0$ then there exists Π'_1 such that $\Pi_1 \longrightarrow^{01}_{\mathit{Erl}} \Pi'_1$ and $\Pi'_0 \sim_{\sigma'} \Pi'_1$ such that $\sigma \subseteq \sigma'$, and the invariant remains valid for σ'.

Proof: We proceed by case analysis and induction over the semantics $\longrightarrow_{\mathit{Erl}}$. In each case, we define a new σ' based on the existing σ for which the invariant holds by assumption. The case analysis hinges on $p \in \mathrm{dom}(\sigma)$ rather than L and H as in classical proofs, e.g. [33] (however, it is important to keep in mind that this predicate corresponds to H/L-typing in form of the proof invariant).

The **high** case is proved once for all cases of the semantic reduction. Let $p_0 \in \mathrm{dom}(\Pi_0)$ and $p_0 \notin \mathrm{dom}(\sigma)$ with $\Pi_0(p_0) \neq \Pi'_0(p_0)$, i.e. this process has been reduced. Let $\sigma' = \sigma$ and $\Pi'_1 = \Pi_1$. Then, $\Pi'_0 \sim_\sigma \Pi'_1$ because $\mathrm{dom}(\sigma' = \sigma) \subseteq \mathrm{dom}(\Pi_0) \subseteq \mathrm{dom}(\Pi'_0)$. The new process that may have been introduced – in case the reduction was with rule SPAWN – is H since, by the Invariant, from $p_0 \notin \mathrm{dom}(\sigma)$ follows that $\Gamma_{\mathit{proc}}(p_0) = H$ *cmd*. In turn, by preservation, the new process has type H *cmd* whereby the Invariant remains valid and indistinguishability as well.

The other case $p \in \mathrm{dom}(\Pi_0)$ such that $p \in \mathrm{dom}(\sigma)$ and $\Pi_0(p_0) \neq \Pi'_0(p_0)$ entails the **low** cases which are proved case by case following the semantics. Generally, we know – since $p \in \mathrm{dom}(\sigma)$ – that $\Gamma_{\mathit{proc}}(p) = L$ and that $\sigma(p_0) = p_1$ for some $p_1 \in \mathrm{dom}(\Pi_1)$. Furthermore, $\Gamma_{\mathit{proc}_\sigma}(p_1) = L$ because σ preserves types. We show the case for `spawn` as it is one of the more interesting cases where a new process is added and consequently changes appear. The other cases are very similar and are omitted.

Case (SPAWN). Let f be defined in the program, $\Pi_0(p_0) = (p, E[\texttt{spawn}(f, [v_1, \ldots, v_n])], q)$, and p'_0 a new pid, then $\Pi'_0(p_0) = (p_0, E[p'_0], q_0)$ and $\Pi'_0(p'_0) = (p'_0, e[v_1/X_1, \ldots, v_n/X_n], ())$ according to the semantic rule SPAWN. Since we consider $p_0 \in \mathrm{dom}(\sigma)$, which is, due to the Invariant, equivalent to $\Gamma_{\mathit{proc}}(p_0) = L$, thus by rule CONFIGURATION $\gamma(t) = L$ *cmd*, and, consequently, by confinement, we have that $\gamma(\texttt{spawn}(f, [v_1, \ldots, v_n])) = L$ *cmd*. Since $p_0 \in \mathrm{dom}(\sigma)$, there is a unique p_1 with $p_0 \sim_\sigma p_1$ and by assumption $\Pi_0(p_0) \sim_\sigma \Pi_1(p_1)$, hence, for $\Pi_1(p_1) = (p_1, t_1, q_1)$, we have $t_0 \sim_\sigma t_1$ and thus, by definition of indistinguishability, $t_1 = E_\sigma[\texttt{spawn}(f, [v_{1_\sigma}, \ldots, v_{n_\sigma}])]$. We can select, $\Pi'_1 = \Pi_1 \dot{\cup} (p_1, E[p'_1], q_1) \dot{\cup} (p'_1, e[v_1/X_1, \ldots, v_n/X_n], ())$ where p'_1 is a fresh pid. Then $\Pi_1 \longrightarrow_{\mathit{Erl}} \Pi'_1$ according to rule SPAWN as well. According to preservation, the successor configurations are well-typed with types $\Gamma_{\mathit{proc}} = \Gamma_{\mathit{proc}} \dot{\cup} (p_0, L$ *cmd*$)$ by typing rule SPAWN. The new processes p'_0 and p'_1 have type L *cmd* by confinement and rule CONFIGURATION; The new isomorphism $\sigma' := \sigma \cup \{(p'_0, p'_1)\}$, whereby the invariant remains valid. Finally, we see that $\Pi'_0 \sim_\sigma \Pi'_1$. □

Corollary 4.7 (Reachable Configurations Security): Let Π_0 and Π_1 be configurations reachable from some initial indistinguishable configurations then there exist σ such that $\Pi_0 \sim_\sigma \Pi_1$.

The corollary follows by induction over $\longrightarrow_{\mathit{Erl}}$ from Lemma 4.6 and the Noninterference Theorem.

4.6. Experimental Evaluation

The security type system we have constructed for the Erlang language shall guarantee statically that there are no illegal information flows from H to L. How is this achieved? We finally want to illustrate here the use of the concepts developed in the current section by some simple experiments. Going back to the informal security analysis from where we started in Section 3.3, we reconsider the argument we pointed out there and show up how the Erlang type system we presented in Section 4.3 enables checking whether a given security policy is valid for the privacy concerns of our data base enquiry program.

Let us reconsider the critical case of the informal analysis. There, we found out that – if we want to keep the variable `Key` private, i.e. H – then `Res` and `Any` (at least) have to be assigned H by the security policy as well. Technically, this is realized by setting up the variable assignment γ as mapping those variables to H. As we have seen before, `Any` and `Res` had to be set to H as a consequence of an implicit flow in the program – both variables depend on `Key`. To test our Erlang noninterference type system let us see what would happen if we were to set these two variables to L.

Type inference is a process of iteratively applying the typing rules from Table 2 starting from the initial assignment given by the security policy, here

$$\gamma_{\mathit{init}} = \{\texttt{Key} \mapsto H, \texttt{Val} \mapsto L, \texttt{Any} \mapsto L, \texttt{Res} \mapsto L\}$$

reconstructing the entire program thereby constructing the security types – which in our case should be impossible.

Indeed, at some point during this reconstruction, we need to give a type for the crucial case statement in `client_eval`. The only applicable typing rule CASE imposes in its conclusion – below the line – that the case construct can only have a type τ *cmd* if for all matches this type τ *cmd* may be constructed for all variables and the branches e_i. However, as `Key` is contained in {`Key, X`} this pattern pat_2 has type H *cmd* (rule MATCH). Hence, the branch e_2 after the `Key` clause must be of type H *cmd* which can only be inferred by rules SIMPLE TERMS, FUNCTION, APPLICATION if `Res` already has type H. Equally, `Any` must be H as well. These two constraints are *not* fulfilled in our initial assumption γ_{init}. This leads to a failure of type inference with our Erlang noninterference type system; the program is rejected for the security policy γ_{init}. A straightforward implementation of the type rules would automatically detect the error in the security policy. In fact, the constraints elaborated above by the walk-through of the type rules can only be fulfilled if the initial assignment of γ_{init} already marks `Res` and `Any` as H – corresponding to our informal security consideration in Section 3.3.

Interestingly, also `Val` – as variable v at the head position of the case statement – must be H according to rule CASE. This is an additional requirement that we have not established informally in Section 3.3. In our case, it is not necessary but, in general, the head position of a case statement can also be a H variable – then forcing the other match variables to be H as well. However, this additional requirement does obstruct neither the functioning of the program nor the correctness of the type analysis. It is a well-known problem that static security analysis is often too restrictive; in fact it must be so because it is provable that an accurate noninterference analysis is undecidable [34].

To ascertain that this over-functioning of the type system does not go too far (i.e. typing everything as H), let us consider just one more variable. To function well according to our global security policy – i.e. the server is public and only the client is our trusted component – the files, that are on the server and on which we want to perform our search and variables, e.g. `Bin`, that are related to those files, need to be assigned L in the security policy γ_{init}. Now, can `Bin` be L? This seems surprising as it is used in close context, i.e. direct pattern matching, with `Val` of which we have just seen that it needs to be H. The crucial pattern matching in the `receive` statement, i.e. term `Val = binary_to_term(Bin)` represents a *legal* information flow from L to H. This is reflected in our type system as follows. As `Bin` is of type L it is – according to rule SUBTYPE – also of type H. Hence, by rule PATTERN, the corresponding pattern matching `Val = binary_to_term(Bin)` can be typed H *cmd* because `Bin` can be typed H and `Val` can be typed H *cmd* : if we assume `Val` initially – in γ_{init} – to be of type H *cmd*, it also is of type H because of rule C-VAL – thus complying to the requirements of the CASE rule seen before.[2]

The proof of type safety and the Noninterference theorem generally guarantee what we have just illustrated by a walk-through of the crucial parts of our program. Any program that is accepted by the Erlang noninterference type system will not leak information from variables designated as H in the security policy. If we want to keep, like in our example, some variable `Key` private, the type system shows us which other parts of the program need to be kept confidential as well.

5. Alternative Approaches and Conclusions

In this final section, we briefly consider alternative approaches and discuss the solution. The approach we have presented – based on the simple idea of running the security-sensitive part of the service on the client site – corresponds to the concepts of common web service implementations like Javascript, Active Server Pages and Java Applets. However, other concepts for web services, like CGI-Scripts or Java Servlets run on the server site. Compared to the presented, secure, client-sited approach, running the entire service remotely is clearly more efficient. An open question is, whether we can provide a *secure* solution to this more efficient way of running our security sensitive key-application on a remote site. We illustrate this alternative approach next using a calculus for active objects. We then sum up relevant work in the Java sector, before we reach our general conclusions.

5.1. Implementation with Active Objects

By adding the concept of objects, familiar from object-oriented programming, to the existing concepts of parallelism and distribution as in a functional – and hence relatively safe – language like Erlang, we provide functional active objects through our new calculus ASP_{fun} [16]. Active objects allow confidentiality by encapsulating local data. In contrast to data locality in a process, the idea of data encapsulation is stronger because the data is an inherent part of an *object*. Such objects are *activated* as a whole entity and become active objects. We can thus remotely activate such objects without the risk of disclosing the confidential data contained in the object.

Although we could also remotely start an Erlang function, we still have to transmit with this activation (or `spawn`) command the initial data for the process, in our case the key we wish to keep secret. Since we cannot assume that the communication channels are secure [31], confidentiality of the key, i.e. privacy, cannot be established.

We can implement a database search using active objects by starting an active object that acts as a kind of gateway

2. Flows from H to L cannot be produced: if a left hand side is a pattern of type L *cmd* this has no relation to H *cmd*.

process. Given confinement of data in an active object, we also achieve privacy. It is beyond the scope of this paper to properly introduce ASP_{fun}, but, as it is a simple and concise calculus, we still use it here to concretize this idea of a gateway object. We now give the – actually very short ASP_{fun} program – with just a very brief and informal explanation of its functionality.

Let Δ be the remote database we wish to search. The following short ASP_{fun} program, when run on a client, activates an object containing a search method σ and the key κ on a server.

```
Active([s = σ, k = κ]).s(Δ)
```

The `Active` command creates a new activity that contains our gateway object `[s = σ, k = κ]` as active object. The call of `s` with Δ as parameter to the then remotely active object – notated in the object-oriented fashion as `.s(Δ)` – initiates the search. The result of this method call is returned to the caller, here the client.

A security consideration for this program, or for ASP_{fun} in general, must assume that active objects are guaranteed to be confined. In other words, the contents of an active object can only be accessed through calls of corresponding methods. This is the principle of language based security. Although, in principle, any malicious remote run-time system can crash our active object and get access to its contents, language standards can guarantee that this will not happen. An example for such language standards is bytecode verification in virtual machines. Additionally, we need to ensure that the invocation of an active object, through the `Active` command, is secure. This means that the transfer of the object to a remote site is done in a secure fashion. Given such security measures, we can apply a security policy that permits only active objects with equal or higher security levels to call methods to an active object, thereby guaranteeing the exclusion of illicit flows.

5.2. Java Privacy Solutions

The server sided approach using active objects with ASP_{fun} we have illustrated in the previous section is also realized for the programming language Java. Andrew Myers has provided in his PhD thesis [22] an approach called Decentralized Label Model (DLM) that enables a rôle based approach to enforcing security in programs. The main idea in DLM is to have explicit labels in the program modelling the actors that have access to labelled program parts. Myers and Liskow augmented the DLM model with the idea of information flow control as described in the papers [23], [24]. Further works by Myers have been mostly practically oriented, foundations only considered later. Initially, he implemented a Java tool package called JFlow that implements the DLM and information flow control [21]. The extensions given by the DLM to the standard noninterference notions lead to undecidable typing. That is, typing cannot be completely performed at compile time but has to be partly done at run-time – which is costly and risky. From the point of view of privacy policies, an interesting experiment is the paper by Hayati and Abadi [13] in which they sketch how privacy policies can be expressed using the DLM. Hayati and Abadi use the Jif framework [26]– the new version of JFlow – to outline the Jif signatures to encode purpose and retention, two important notions for privacy policies. In more recent work, still along the same lines, Zheng and Myers [37] have gone even further in exploring the possibilities to dynamically assign security labels while still trying to arrive at static information flow control. A recent implementation of these concepts is given by the framework called Sif – for Servlet Information Flow – which basically represents a new version of Jif, i.e. Jif 3.0, but has an additional layer of Java servlets that focuses thus on the support of web-services with server-sided applications.

The impressive amount of work by Myers and his team pushes the idea of static analysis of information flow with type systems very far. The DLM concept enables even multilateral security modelling but causes troubles with decidability. However, being centred around the Java world, the work additionally has to struggle with concurrent access to data. Using active objects, as sketched on a conceptual level in the previous section, the clear separation of separate data spaces grants a simpler and more natural use of distribution in web-services.

5.3. Conclusions

Triggered by public demand, privacy in social networks – in particular internet based ones – increasingly attracts scientific interest. Leading European projects, like Primelife [29] and its predecessor Prime, bundle scientific and industrial competence to explore privacy in future networks and services. Various techniques in computing have been applied to tackle the related challenges, for example, artificial intelligence and unsupervised learning [8].

We have presented an approach to a privacy-enhancing data base enquiry that solves the problem by not disclosing the search key but by performing the search itself. The concept has been demonstrated by a simple implementation in Erlang; it's feasibility has been achieved through Erlang's high scale parallelism. A final security consideration shows, informally and formally, that the security goal of keeping the key data confidential, i.e. private, is achieved.

As briefly mentioned in Section 2.3, a sensible extension of our parallel search program is to evaluate in each step the results obtained in the previous step with respect to new goals, i.e. file names, to continue the search. This is a simple enough extension that merely needs to identify new file names, or more generally Internet sites, to continue the database enquiry. However, this kind of goal-directed search

raises new security issues. An observer could infer some information about the key from the way we continue our search because we select the file names from the matched search results. To overcome this, we have to cover up our search and load, as before, all possible files referenced in the previous round. However, for the sake of efficiency, we would only really analyze those that we know to be interesting. Here, again, a new covert channel opens up, that is not visible in the data flow based model we use in Section 3: an attacker could distinguish our two ways of treating incoming files in this "cover-up" analysis by measuring the times between new demands for connections with `gen_tcp:send(..., FileS)`.

The informal security argument we have shown is not invalidated by this consideration. In the simple implementation, we have not used information from the files. There simply is no dependency between the files being downloaded. Consequently, we cannot lose any information from them. A similar analysis of the extended "sensible" search would reveal this illegal information flow because we would use information from files analyzed in previous rounds to determine the new `FileS`.

Our informal security analysis is first informally introduced and then developed into a formal notion of information flow for Erlang. We develop a security type system and prove that when a program is well-typed according to this type system then there are no (undesired) covert channels. This approach is widely accepted for language based security analysis; a concise overview of type-based information flow control is given by Sabelfeld and Myers [32]. The special challenge we have been facing here for Erlang is that we have to address concurrency. Earlier papers on security type systems [33], [5] for concurrent systems already point out the following difficulty: the termination of processes is observable by other processes. The motivating example [5] shows that this problem is due to concurrency of shared data. By contrast – as Erlang processes have strictly separate data spaces – these incriminating examples cannot be reproduced in our case; since active objects in ASP_{fun} also have separate data spaces this security problem is neither relevant there.

The construction of a security type system for Erlang and proof of noninterference for this type system – presented in this paper – provide a general security measure that can straightforwardly be implemented in a static type checker for Erlang security. Extending these ideas to active objects is current research.

Acknowledgements. We would like to thank Jeff Sanders for helping us to get the initial understanding and the anonymous referees for constructive criticism and valuable links to related approaches.

References

[1] J. Armstrong. *Programming Erlang – Software for a Concurrent World.* The Pragmatic Bookshelf, 2007.

[2] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[3] A. Banerjee and D. A. Naumann. Stack-based Access Control for Secure Information Flow. *Journal of Functional Programming* **15**(2), 2003.

[4] A. Barth and J. C. Mitchell. Enterprise Privacy Promises and Enforcement. *WITS'05*. ACM 2005.

[5] G. Boudol, I. Castellani. Noninterference for Concurrent Programs. *ICALP'01*. LNCS:**2076**, Springer, 2001.

[6] R. Carlsson. An introduction to Core Erlang. *Proceedings of the PLI'01 Erlang Workshop*, 2001.

[7] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. *Journal of the ACM*, **45**(6): 965–982, 1998.

[8] G. Choral, E. Armengol,A. Fornells, and E. Golobardes. Data Security Analysis Using Unsupervised Learning and Explanations. *Innovations in Hybrid Intelligent Systems*. Advances in Soft Computing (ASC), **44**: 112–119, 2007.

[9] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, **19**(5): 236–242, 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communication of the ACM*, **20**(7), 1977.

[11] M. Felleisen, D. P. Friedmann, E. Kohlbecker, and B. Dulba. A syntactic theory of sequential control. *Theoretical Computer Science,* **52**(3):205–237, 1987.

[12] J. Goguen and J. Meseguer. Security Policies and Security Models. *Symposium on Security and Privacy, SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

[13] K. Hayati and M. Abadi. Language-based Enforcement of Privacy Policies. *Privacy Enhancing Techniques*. **3432**:302–313, 2005.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[15] M. Hennessy. The security pi-calculus and non-interference. *The Journal of Logic and Algebraic Programming*. **63**:3–34. Elsevier 2005.

[16] L. Henrio and F. Kammüller. Functional Active Objects: Typing and Formalisation. *8th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'09. Satellite to ICALP'09.* Proceedings to appear in ENTCS, Elsevier, 2009. Also invited for journal publication in Science of Computer Programming, Elsevier.

[17] F. Huch. Verification of Erlang Programs using Abstract Interpretation and Model Checking. *ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*. ACM Sigplan Notices, 34(9):261-272, 1999.

[18] F. Kammüller. Formalizing Non-Interference for A Small Bytecode-Language in Coq. *Formal Aspects of Computing*: **20**(3):259–275. Springer, 2008.

[19] F. Kammüller and R. Kammüller. Enhancing Privacy Implementations of Database Enquiries. *The Fourth International Conference on Internet Monitoring and Protection.* IEEE Computer Press, 2009.

[20] S. Marlow and P. Wadler. A practical subtyping system for Erlang. *ACM SIGPLAN Notices*, **32**(8):136–149, 1997.

[21] A. C. Myers. JFlow: Practical mostly-static information flow control. *26th ACM Symposium on Principles of Programming Languages, POPL'99.*

[22] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, Cambridge, 1999.

[23] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. *IEEE Symposium on Security and Privacy*. 1998.

[24] A. C. Myers and B. Liskov. Protecting Privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*. **9**:410–442, 2000.

[25] T. Noll and C. K. Roy. Modeling Erlang in the Pi-Calculus. *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang, Erlang'05.* ACM Press, 2005.

[26] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, N. Nystrom. *Jif: Java information flow.* Software release at `http://www.cs.cornell.edu/jif`. 2001.

[27] S.-O. Nyström. A soft typing system for Erlang. *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang, Erlang'03.* ACM Press, 2003.

[28] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1995.

[29] Primelife. Bringing sustainable privacy and identity management to future networks and services. EU research project in the 7th FP. `http://www.primelife.eu/`, 2009.

[30] M. O. Rabin. How to exchange secrets by oblivious transfer. *TR-81, Aiken CL, Harvard University*, 1981.

[31] K. Rikitake and K. Nakao. Application Security of Erlang Concurrent Systems. *Computer Security Symposium, CSS'08*. Okinawa, 2008.

[32] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications*, **21**:5–19. IEEE 2003.

[33] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. *POPL'98.* ACM 1998.

[34] D. Volpano and G. Smith. A Type-Based Approach to Program Security. *TAPSOFT'97.* LNCS **1214**, Springer 1996.

[35] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, **4**(3): 167–187, 1996.

[36] G. Wolf and A. Pfitzmann. Properties of Protection Goals and Their Integration into a User Interface. *Computer Networks*, **32**:(685–699), 2000.

[37] L. Zheng and A. C. Myers. Dynamic Security Labels and Static Information Flow Control. *International Journal of Information Security*. **6**(2–3), Springer, 2007.

Appendix

In the rules for pattern matching, `case`, and `receive`, we use the functions `match`, `casematch`, and `queuematch` for modelling the pattern matching mechanism of Erlang. This mechanism is more complicated than those of other programming languages because of non-linear patterns with multiple occurrences of the same variable.

$$\begin{aligned}
&\texttt{match}(X, t) := [t/X] \\
&\texttt{match}(c(pat_1, \ldots, pat_n, c(v_1, \ldots, v_n)) := \\
&\qquad \texttt{match}(pat_1, v_1) \cup \ldots \cup \texttt{match}(pat_n, v_n) \\
&\texttt{match}(_, _) := \text{Fail, otherwise}
\end{aligned}$$

Two derived substitutions can only be unified if the overlapping parts are identical. Otherwise the matching fails.

$$\begin{aligned}
\text{Fail} \cup \sigma &:= \text{Fail} \\
\sigma \cup \text{Fail} &:= \text{Fail} \\
\sigma_1 \cup \sigma_2 &:= \begin{cases} \sigma_1 \cup \sigma_2, \text{ if } \forall X \in (\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2)). \\ \qquad\qquad \sigma_1(X) = \sigma_2(X) \\ \text{Fail, otherwise} \end{cases}
\end{aligned}$$

The function `case` evaluates to the expression corresponding to the first pattern matching a given value. The function `casematch` returns a tuple containing the number of the first matching pattern and the corresponding substitution, or Fail, if none of the patterns matches.

$$\begin{aligned}
&\texttt{casematch}((pat_1, \ldots, pat_n), v) \\
&= \begin{cases} (i, \rho), \text{ if } \texttt{match}(pat_i, v) = \rho \text{ and} \\ \qquad\qquad \texttt{match}(pat_j, v) = \text{Fail } \forall j < i \\ \text{Fail, otherwise} \end{cases}
\end{aligned}$$

The constructor `receive` has the same behaviour but all values in the queue have to be considered. In Erlang, a pattern is successively matched against all values in the queue before the next pattern is matched. This is implemented in the function `queuematch` which returns the match and in addition the position of the queue value that matches.

$$\begin{aligned}
&\texttt{queuematch}((pat_1, \ldots, pat_n), (v_1, \ldots, v_n)) \\
&= \begin{cases} (i, j, p), \text{ if } \texttt{match}(pat_i, v_j) = \rho \text{ and} \\ \qquad\qquad \texttt{match}(pat_i, v_k) = \text{Fail } \forall k < j \text{ and} \\ \qquad\qquad \texttt{match}(pat_l, v_h) = \text{Fail } \forall l < i,\ h \leq n \\ \text{Fail, otherwise} \end{cases}
\end{aligned}$$