Enhancing Privacy Implementations of Database Enquiries

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, and conclude by providing an informal security analysis.

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 [4]. Such an analysis verifies a formal notion of security over different data domains, so-called non-interference [5], 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 [2].

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). Finally, we briefly contrast our solution with an alternative approach that uses active objects, and offer our conclusions (Section 4).

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 [6], 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 \ 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 [12], follows ideas similar to the original "Conjugate coding" by Stephen Wiesner now so popular through quantum cryptography. The scheme of private information retrieval [3] is closely related. This scheme adandons 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 [3]. 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 [11]. 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.
```

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.

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

```
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. The input of the file names of the files to be searched is provided by an input file on the client site. The gradual selection of new source files for a goal-directed search may be integrated (see Section 4).

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. 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 [15] 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) [4] 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 most natural way to formalize confidentiality is non-interference [5]. There are quite a few different definitions of non-interference [14], 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 non-interference. 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 non-interference 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 non-interference 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 non-interference 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 non-interference 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 non-interference type systems [9], [14].

3.3. Informal Security Analysis of Privacy-Enhancing Database Search

Although we do not intend to provide a formal analysis according to some notion of non-interference, we 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 non-interference, 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 the first invocation, `Key` is again assigned by pattern matching to the variable `Key`, which is also high. In the first recursive call, it is similarly assigned to the variable `Key`, but in the second it is assigned to `Any`. This is why our policy needs to label `Any` as high. The variable `Any` is not used in any further function calls, so there are no explicit flows from it 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 – where two of our high variables `Key` and `Any` are tested to select the branch – is the `receive` statement in `client_receive`. Here, an implicit flow is from `Key` to `Res`, which is legal as the latter is high as well. Considering, finally, the variable `Res`, we see that, here too, there are no flows from it to any low variable, either explicit or 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. Alternative Approach and Conclusions

In this final section, we briefly consider an alternative approach 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.

4.1. 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} [8]. 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 [13], 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 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 on a server an object containing a search method σ and the key κ.

```
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.

4.2. Conclusions

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, 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, though informally presented, based on a formal notion. We could make it fully formal. We have, however, adopted a state-based view of non-interference, which might seem unusual in a functional set-up. Although the approach works well on our small example, there is a general problem with this view of non-interference for parallel programs: the termination of processes is observable by other processes. Since we assume termination for the definition of non-interference, we cannot take this security problem into consideration. In future work, we plan to exploit a rigorous translation from Erlang to the π calculus [10] to represent our application in a calculus that is more easily accessible to a formal analysis: we can then use existing formalizations of non-interference for the π calculus [7] to demonstrate information flow security.

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

References

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

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

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

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

[5] J. Goguen and J. Meseguer. Security Policies and Security Models. *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

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

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

[8] L. Henrio, F. Kammüller and H. Sudhof. ASPfun: A Functional and Distributed Object Calculus: Semantics, Type-system and Formalization. INRIA Research Report N. 6353, November 2007

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

[10] T. Noll and C K. Roy. Modeling Erlang in the Pi-Calculus. *Erlang'05.* ACM Press, 2005.

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

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

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

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

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