Technical Report 171
Universitt Freiburg

Towards a Formal Analysis of a Mix Network

Burkhart Wolff*

Oliver Berthold‡
Sebastian Clauß‡
Hannes Federrath‡
Stefan Köpsell‡
Andreas Pfitzmann‡

January 31, 2002

*Institut für Informatik
Albert–Ludwigs–Universität Freiburg
Georges-Köhler-Allee 52
D-79110 Freiburg, Germany
Tel: +49 (0)761 203-8240
Fax: +49 (0)761 203-8242

{wolff}@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~{wolff}

‡Institut für Systemarchitektur
Fakultät Informatik
Technische Universität Dresden
D-01062 Dresden, Germany

{ob2,sc2,hf2,sk13,pfitza}@inf.tu-dresden.de

Abstract

We present a formal model of Mixes in an open network (following the concepts of D. Chaum) and apply well-known analysis techniques to a new type of security property, namely unlinkability. The network is composed of senders, receivers and Mix stations. The network and its components are formalized as CSP processes, and combined with a (passive) attacker. Based on the model-checker FDR, formal analyses of networks and their security properties are performed. The approach serves as a feasibility study for the analysis of anonymity and unobservability with a particular analysis technique. Moreover, it will result in the implementation of a particular analysis-testbed for the investigation of other security properties such as unobservability or more advanced protocols that might pave the way to secure Mix implementations with dynamically controlled dummy traffic.

Contents

1 Introduction **3**

2 Mix Schemes and Mix Networks **4**
2.1 A more Formal Definition of Anonymity, Unobservability and Unlinkability 5
2.2 Basic Mix Operations and Models 5
2.3 Mix Chains 7
2.4 Structure of a Mix Message 8

3 Formal Security Analysis with CSP **9**
3.1 Tools for CSP 9
3.2 Lowe's Approach to Protocol-Analysis 10

4 A CSP Model for Mix Networks and its Attacker **11**
4.1 Global Constants 11
4.2 Basic Functions 13
4.3 Basic Data Types and Operations 13
4.4 Crypt- and Decrypt Primitives 13
4.5 Abstracting the Set of Messages MSG 14
4.6 The Components of the Mix Network:
Channels, MIXes, SENDer, RECeiver 16
4.7 The Mix Network 18
4.8 A Simple Instantiation of the Generic Passive Attacker 19
4.9 An Improved Instantiation of the Generic Passive Attacker 19

5 Analysis **21**
5.1 Consistency Analysis of MIXes, SENDer and RECeiver 21
5.2 Consistency Analysis of the Network 21
5.3 Analysis of attacks against Anonymity 22

6 Conclusion **22**
6.1 Discussion 22
6.2 Future Work 23

References **23**

1 Introduction

Using Internet services means leaving digital traces. On the one hand, the technical reality makes anonymity and unobservability on the Internet highly difficult, on the other hand, there is no doubt that, as a basis of the information society, there is a fundamental need for anonymous communication. The availability of anonymous communication is considered a constitutional right in many countries, for example in voting or counselling procedures.

We distinguish *unobservability*, *unlinkability* and *anonymity* in secure systems. Systems that provide unobservability ensure that no third party, not even the underlying network, is able to recognize, if there is communication at all. As a consequence, an attacker cannot find out who communicates with whom, a property we call *unlinkability*. However, the communicating parties may know and usually authenticate each other. As an example, one might consider paying users browsing a patent data base. In contrast, systems that provide *anonymity* ensure that sender or receiver (or both) can communicate without revealing their identity to each other. We will describe anonymity and unobservability in detail in Section 2.1.

Both anonymity and unobservability can be achieved by using a Mix network.

A single Mix is a message forwarding server, which hides the relation between the incoming and outgoing messages. This is done by collecting a number of messages, applying some cryptographic operations and reorder them before forwarding them on their way.

Mixes, their networks and their security properties have risen interest in security research communities; various constructions and their properties have been investigated. An overview is given in Section 2.

Because of the complexity of Mixes and their protocols, we see a need to use formal methods to analyse unobservability and similar security properties. There are two fundamentally different approaches: *possibilistic* and *probabilistic* ones. A possibilistic framework enables to express, which runs of a system are possible. For confidentiality, for example, this means, that the framework is able to express whether there are any alternate runs which have the same behaviour on all interfaces the attacker has access to, but are different in the events which shall be kept a secret from the attacker. A probabilistic framework enables to weight runs with their probabilities. Observing statistically significant deviations from the standard probabilities of runs can pave the way for probabilitic attacks against security properties. For confidentiality, for example, this means that it is possible to express the entropy remaining for the attacker, i.e. his/her missing information with respect to the events, which shall be kept secret.

In this report we analyse the security of an anonymity providing system using the possibilistic famework.

Experience has shown that the design of protocols for systems assuring security properties is prone to sometimes spectacular errors. For instance, the story of some authentication protocols [17, 2] shows, that they revealed severe design errors that had not been discovered in decades, although they had been intensively investigated by hand. Here, a higher degree of trust into these system components could be achieved by applying

tool-supported analysis techniques developed in the field of *Formal Methods* having their roots in mathematical logics and program semantics theory.

Meanwhile a number of analysis techniques are established (at least for authentication protocols): The common ground of these approaches is to explore all possible *runs* (sequence of possible computations and communications within a system) and to check if they all fulfil a given security property. Typically, the set of runs is infinite; moreover, this set may interfere with the actions of an intelligent *attacker* that may attempt a breach of the security goals the system attempts to enforce [5, 7, 9, 10, 14, 15, 16].

In principle, three approaches to analyse the set of runs can be distinguished:

1. Automata theoretic approaches. The underlying idea is to view the set of runs as *regular language* that is generated by a finite automaton. The existence of an attack against system security is then reduced to finding a path in the finite automaton to a state representing a security breach. Among the automata-theoretic approaches, there are also process-algebras that allow for compact, abstract but nevertheless problem oriented specifications that were compiled to automatas automatically.

2. Heuristic search approaches. Here, the state-transitions are modelled and constructed explicitly; the heuristic search of critical states is made feasible by a controlled generation of the (possibly infinite) state space [1].

3. Theorem proving approaches. Here, either specialized logics such as *belief logics* [2] or *modal logics* model axiomatically the growth of knowledge of system agents about each other, or the set of possible runs is characterized inductively, allowing to prove via induction that no attack may exist [19].

Besides the theorem prover approach, which has been successfully applied using systems such as Isabelle [18] or PVS, automata theoretic approaches are most popular at present and had been implemented using systems such as SMV, SPIN etc. The process-algebraic variant of the automata theoretic approach has been implemented most successfully on FDR [12, 21, 23, 24]. It is the ultimate goal of this paper to transfer the analysis techniques developed for FDR to the problem of anonymity and to check the feasibility of the approach for this problem domain.

As main contribution of this paper, we present a formal model of a Mix network suited for a machine based analysis. It can be used to find known and potentially new attacks. Standard models and standard formal notions may improve the understanding of informal ones. Moreover, they pave the way for the formal analysis of Mix implementations. Further, we have a proof of concept of an adoption of existing protocol analysis techniques to security properties such as anonymity and unobservability.

2 Mix Schemes and Mix Networks

In this section, we first define anonymity, unobservability and unlinkability and describe their differences. Then we introduce a Mix as elementary building block for providing anonymity of sender or receiver. We describe different schemes for a single Mix, and we

explain possibilities of chaining Mixes together. At the end of this section, we show how unobservability can be achieved using Mixes.

2.1 A more Formal Definition of Anonymity, Unobservability and Unlinkability

Anonymity of a subject is the state of being not identifiable within a set of subjects, the anonymity set [20]. In the case of a Mix system, a sender of a message can be anonymous within a set of possible senders, and a receiver of a message can be anonymous within a set of possible receivers. Anonymity is stronger, the larger the respective anonymity set is and the more evenly distributed the communicated subjects within that set are.

Unobservability means that no third party is able to decide, whether a message was transmitted (sent or received) within the system, or not. Unobservability implies anonymity against third parties.

Unlinkability of two or more items (e. g. subjects, messages, events, actions, ...) means that within a run of a system, these items are no more and no less related than they are related concerning the a-priori knowledge. This means that the probability of those items being related stays the same before (a-priori knowledge) and after the run within the system (a-posteriori knowledge of the attacker). This means that two messages are unlinkable if the probability that they are sent by the same sender and/or received by the same recipient is the same as those imposed by the a-priori knowledge. We can describe anonymity in terms of unlinkability between messages and senders/receivers. *Relationship anonymity* means that it is untraceable who communicates with whom. In other words, sender and receiver are unlinkable.

In this paper we exemplify our approach with relationship anonymity; this can be easily adapted to sender or receiver anonymity. As already mentioned, we will remain in a possibilistic framework.

Regarding the possibilistic framework, a relationship anonymity system provides 2-relationship-anonymity, if there are at least two alternate runs, which have the same behaviour on all interfaces the attacker has access to, but have different senders or receivers.

Let R be the set of runs, which are indistinguishable by an attacker, and let P be the set of all sender/receiver pairs deduced from R. We can generalize the above definition to n-relationship-anonymity, if $n = card(P)$.

In the probabilistic framework, the probabilities of different runs can be taken into account. In the case of Mix networks, it can be described, which sender has sent a message with which probability.

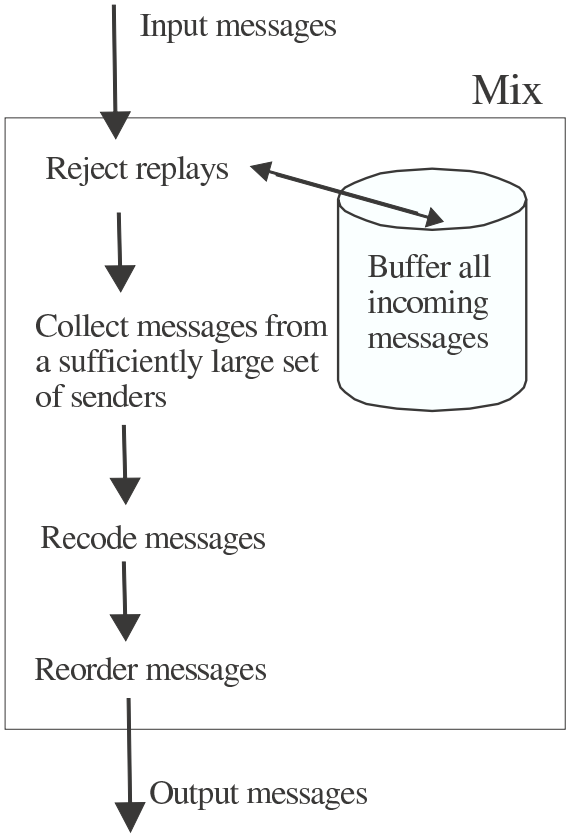
2.2 Basic Mix Operations and Models

A Mix is a special kind of router. It forwards messages but hides the relation of incoming and outgoing messages in a large set of messages. It changes order and appearance of all messages before forwarding them to their recipients or to another Mix.

The basic functions, which a Mix performs to hide the relation between incoming and outgoing messages, are shown in Figure 1. These functions are:

1. Collecting messages (from a sufficiently large set of senders). This ensures that the basic anonymity set of a message is large enough.
2. Rejecting so-called *replays* of messages. This action prevents from a special active attack used to track single messages through a Mix.
3. Cryptographic decoding of messages. This ensures, that messages cannot be tracked by their appearance.
4. Reordering of messages. This protects from tracking messages by the order in which they arrived at the Mix.

Figure 1 A Mix with its basic operations



Changing the appearance is done by means of cryptography. Inside the Mix, messages are transformed (encrypted or decrypted) and thereby change their appearance, so that an attacker can not draw any conclusions about the relation between incoming and outgoing messages based on comparing their appearance. This step is common for all

models of Mixes. The address to which the Mix should forward the message is known to the Mix, because either it is part of the encrypted message body or it is implicitly known, because there is only one subsequent Mix.

One difference of current models is their way of reordering messages. In the first Mix model, described by David Chaum in [3], it is done by collecting a batch of messages. The size of that batch is a publicly known property of a particular Mix. The incoming messages are decoded and reordered to form an outgoing batch of messages. This kind of Mix is called a *Batch Mix*. While a random ordering would suffice, Chaum proposes lexicographical ordering in order to eliminate a hidden channel through which the Mix could leak information and poses an easy way to verify that part of the Mix's work.

A *Pool Mix* keeps a number of messages in its memory. As soon as the next message arrives, it is decoded and added to the pool. A message is randomly chosen from the pool and transmitted. The pool's size is, like the batch size in the previous model, a public attribute of that Mix. The most notable problem with Pool Mixes is the lack of upper boundaries for the time a message needs to pass through. This nondeterminism makes it difficult to verify that Pool Mixes operate correctly. For more information on Pool Mixes see [4].

Another model is the *Stop-and-Go Mix*[11]. This type of Mix will not wait for a special number of messages to process them, but will delay each message by a certain amount of time, which is selected by the sender. This model, as well as the Pool Mix model, reduces the burst like stress on the network and other resources that Batch Mixes produce. But unlike Pool Mixes it maintains determinism, and by this a way of verifying part of the Mix's work. It comes with a major drawback, though. To work effectively there have to be enough messages present in the Mix at anytime. Otherwise the anonymity set will be too small to assure a desired degree of security, making the relation between ingoing and outgoing messages one-to-one and therefore insecure in the extreme case. Since the delays are chosen by the users independent from each other, this situation is possible. In order to make this scenario fairly unlikely, delays have to be chosen relatively long in comparison to the average time between message arrivals in each Mix.

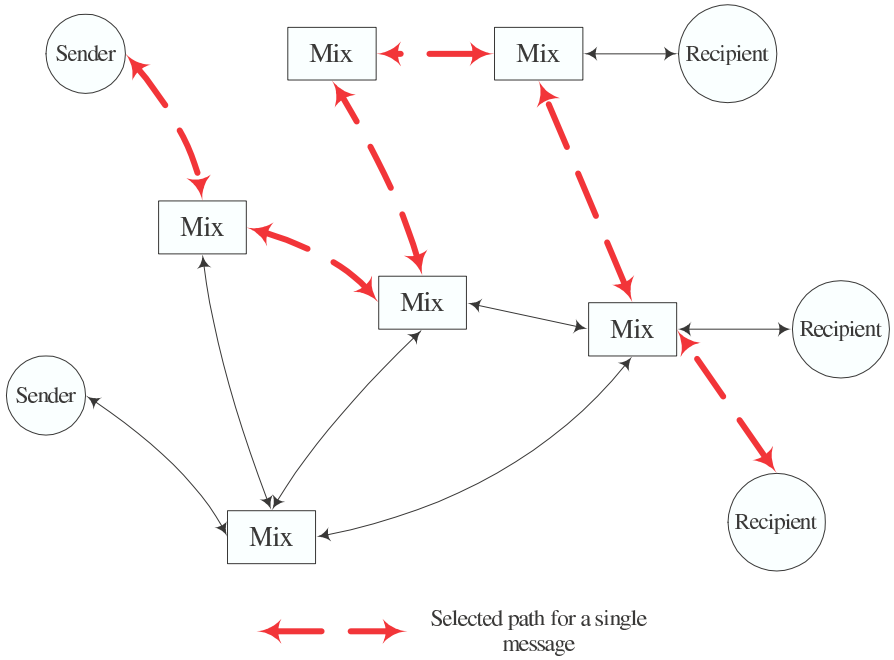
2.3 Mix Chains

A single trustworthy Mix can provide untraceability of messages even if all network connections are observed by an attacker. But if a Mix leaks its information to an attacker (no matter if it happened intentionally or due to an error) there is no protection at all. For this reason a user should use more than one Mix. By sending a message through a chain of Mixes a user can increase the chance that his message passes at least one trustworthy Mix.

In general there are two possibilities of chaining Mixes together. In a *Mix network* the user can select an individual path through the network for each message. This is illustrated in Figure 2. Every Mix can send messages to all other Mixes (the arrows in Figure 2 show some message transmissions). The thick arrows highlight the path, which one sender has selected for his message.

Another concept of chaining Mixes together is called *Mix Cascade*. A Mix Cascade is

Figure 2 Example of a Mix network



a static chain of Mixes. The user can only choose between different Mix Cascades. An example of a Mix Cascade is shown in Figure 3.

2.4 Structure of a Mix Message

A message to be sent through a Mix consists of a message body and a recipient address[1], to which the Mix sends the message after processing it. The message must be encrypted for the Mix.

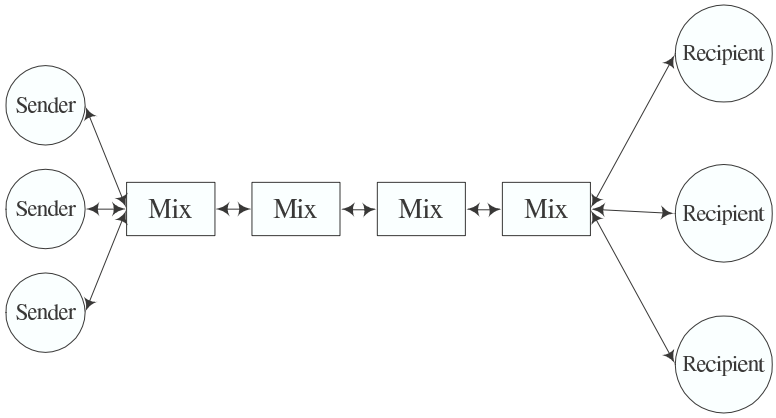
When a sender prepares a message to be sent through a chain of more than one Mix, the message and the receiver's address have to be encrypted multiple times. First they have to be encrypted for the last Mix. The result is encrypted for the last but one Mix along with the last Mixes address and so on.

An easy way to imagine this is a Russian Matrushka doll. The message is in the centre, and layers upon layers of encryption are built around it.

Every Mix along a chain only knows the previous and next station[2] of a message. Therefore an attacker has to undermine all Mixes in that chain to completely track a message.

[1] In case of a Mix Cascade the recipient address can be omitted for all but the last Mix of a cascade, because in that case the order of the Mixes is static.
[2] These stations include sender and recipient.

Figure 3 Example of a Mix Cascade consisting of four Mixes



3 Formal Security Analysis with CSP

The process algebra CSP (*Concurrent Sequential Processes*; originally developed by [8]) is a formal specification formalism that makes it possible to view a system as a composition of components (processes) that interact via synchronous events. Processes can be described by parameterised recursive equations over process operators. Events are considered as data-types as in functional programming languages, which makes it possible to compute them conveniently by pattern matching and higher-order functions. We assume that the reader is rudimentarily familiar with CSP; we refer to [22] for more details.

Since it is quite natural to view security components as concurrent systems, and since security properties are properties on all runs of them, CSP has attracted many researchers for its use in formal protocol analysis; the work of Lowe [12], Roscoe et al. [21, 23] and Schneider [24] are only some of the examples.

3.1 Tools for CSP

There are basically two tools to formally reason over CSP — one based on the model-checker FDR [13], one one based on a conservative embedding of CSP into higher-order logic called HOL-CSP [25]. The former approach offers automatic refinement proofs for finite models, the latter interactive proofs for infinite or declaratively specified models.

FDR converts specifications in an extension of CSP called CSP_M into (cleverly compressed and represented) regular automata. On this basis, FDR decides the refinement relations between a specification S and an implementation I. Two refinement relations are of importance in our context: $S \sqsubseteq_T I$ (I refines S in the *trace model*, i.e. I is more special in the sense that all runs in I must be possible in S) and $S \sqsubseteq_F I$ (S refines I in the *failures model*, i.e. I is more deterministic in the sense that all runs in I must be possible in S and they must lead to states, where I can less often refuse to engage in events than S). A deadlock in the failures model is represented by a trace leading to a state where

all possible events are refused. Thus, it is possible to express deadlock-freeness of P by asking if P refines in the failure model the process that non-deterministically chooses arbitrarily one of the possible events (see [22] for more details). FDR allows to prove refinement statements (both in trace- and failure models) as well as deadlock-freeness of processes automatically.

HOL-CSP has as basis not only a functional programming language, but also higher-order logic (Isabelle/HOL). As such, HOL-CSP is in more expressive than CSP_M. However, the main tool of reasoning over processes is interactive fixpoint-induction (which works also for proving deadlock-freeness), which requires substantially more effort.

Since it is our goal to investigate Mix networks along the more classical lines of analysis techniques, and since an analysis in HOL-CSP would profit from intermediate results of a FDR-based analysis, we chose to use model checking techniques as implemented in FDR for the current state of our work.

3.2 Lowe's Approach to Protocol-Analysis

In Garwin Lowe's [12] seminal paper for protocol analysis based on CSP and FDR, all agents of a system and the attacker are represented as processes. The attacker may be passive (only listening) or active (listening and emitting messages according to the network protocol). When the attacker reaches a success state — i.e. a security property is violated — he sends a success event. Both passive and active attackers can be formalized generically in CSP_M:

```
channel Success
channel C : . . .      -- general communication channel


ATTACK(success, incl, S) =
   if success(S)
   then Success → SKIP
   else C?x?y?z → ATTACK(success, incl, incl(x,y,z,S))

ATTACKA(success, incl, synth, incl2, S) =
   if success(S)
   then Success → SKIP
   else     C?x?y?z → ATTACKA(success, incl, synth, incl2, incl(x,y,z,S))
        □(x,y,z) : synth(S) •
            C!x!y!z → ATTACKA(success, incl, synth, incl2, incl2(x,y,z,S))
```

an operation *incl* allowing to abstract and insert a received message into its state S. The active attacker can additionally synthesize a message via *synth* and emit it via *incl*2. This message can be stored into the attackers state and marked as "generated" by the attacker.

Based on these generic attackers, a wide range of attacker models can be formalized. The question: "Does the desired security property hold in my system?" can be formalized in the form of a refinement as follows:

```
RUN = C?x?y?z → RUN

assert RUN ⊑_T (SYSTEM [| {| C |} |]
                ATTACK (success,incl,init_attack_state))
```

If the combined system reaches a success state of the attacker (i.e. the security property does not hold), then the *Success*-event may occur in some trace. In such a case, the combined system is *not* a refinement of RUN, and FDR will generate a counter-example

(i.e. an attack) for the claimed refinement. Note, however, that in this construction the deadlocking process *STOP* would trivially fulfil *any* security property; hence, it is necessary to check for each combined system its deadlock-freeness. Deadlock-analysis is hence a prerequisite, a kind of sanity check for any protocol analysis of a system in Lowe's setting.

4 A CSP Model for Mix Networks and its Attacker

In this core chapter, we will develop a model for Batch Mixes suitable for an analysis following the FDR approach. Our main requirements for a suitable model are:

- it should be deadlock free,
- represent an abstract implementation of Mixes,
- be robust under denial-of-service attacks
- and analysable for a non-trivial Mix network.

As we will see, this turns out to be a highly non-trivial task to achieve these goals at the same time. An optimal solution is even impossible.

Since model-checking yields a yes-or-no decision, it makes no sense to present our final version of our Mix model immediately, with all its limitations. Rather, we consider the techniques that lead to this version as important in order to extend or modify the existing approach, and will therefore present a sequence of versions, with their pro's and con's. Local deadlock- unfeasibility- or security problems lead to revisions of these models yielding incrementally our (currently) final version.

4.1 Global Constants

We start our model description with a collection of global constants that control the size of our *finite* Mix model and its attacker. For an analysis based on model-checking, playing with these constants is crucial: Making the model too small prevents from finding interesting attacks, making the model too large makes its check with a tool infeasible. While CSP_M offers a small functional language for computing data that is communicated between the processes and thus a fairly abstract means to describe some aspects of the model, attempts to find abstractions of the data to be computed will turn out vital for a successful use of model-checking in our problem domain.

From the informal description of MIX-stations in the introductory parts of the paper, it is straight-forward to identify some of the global parameters of our model: the number of users *userno* (both sender and receiver in the network), the number *mixno* of MIX-stations, the size of the buffer $bufsize$ inside a MIX, the number of basic messages (i. e. the elementary "raw" messages sent from user to user and not encryptions, encryptions of encryptions over them, etc). Moreover, an important measure that controls the size of our model is the "message complexity" *msgcxty* that will be discussed later in more

detail. For the moment, it suffices to know that the nesting depth of encryptions and therefore the length of a message path is controlled by this constant.

The following lines in CSP_M declare and define these constants for a suitable minimal setting. Note that the lexical conventions of CSP_M allow for two types of comments, either started by `--` to the end of the line or by `{- ... -}`.

```
userno   = 2        -- how many users
mixno    = 2        -- how many MIX-stations
msgno    = 1        -- number of different messages

msgcxty = mixno     -- complexity (nesting depth) of messages

bufsize = userno    -- size of the buffer in each MIX
```

It turns out that the number of elementary messages can be restricted to just one without loss of generality in an analysis we are heading for: since the attacker's goal is just to achieve a suitably restricted communication relation, it does not matter *what* has been communicated after all.

Based on these basic constants, we will build suitable "types" in order to characterize the essential entities *users*, *mixes* and their union *agents* abstractly:

```
userids       = {1..userno}
mixids        = {1..mixno}

datatype ids = user.userids
             | mix.mixids

users         = {user.i | i ∈ userids}
mixes         = {mix.i  | i ∈ mixids}
agents        = union(users, mixes)
```

The definition of the function *paths* that associates to each user a list of numbers representing MIX-stations follows below. This function can either be defined *individually*, i.e. different for each user ("Mix network") or *globally*, i.e. identical for each user and message ("Mix cascade").

```
{-
  paths(user.1) = {<2,1>,<1,2>}
  paths(user.2) = {<1,2>,<1,1>}
  paths(user.3) = {<1,2>,<2,2>}
  paths(i)      = paths(user.2)  -- catchall for userno > 3.
 -}

paths(_) = {<1..mixno>}
```

In our sample code, the version for individual paths is commented out; the set of path-lists in our example is assumed to be random.

Note that with respect to the choice of the concrete *paths*-function, we assume the following requirement for the *paths*-function to hold:

$$\forall u \in user, p \in paths(u) \bullet \{i | i \; in \; p\} \subseteq mixids \tag{1}$$

$$\forall u \in user, p \in paths(u) \bullet length(p) <= msgcxty \tag{2}$$

This requirement reflects the fact that the message complexity constant *msgcxty* limits — as already mentioned — the nesting depth of encryptions of messages. Since in a Mix network, each Mix removes one encryption envelope and thus decreases the nesting depth by one, *msgcxty* inherently limits the length of possible paths.

4.2 Basic Functions

Besides the CSP_M functions (see FDR-Manual), we use in our specification the usual functions front, map, filter, rev etc. known from functional programming.

```
front(<x>)       = <>
front(<x>^<y>^R) = <x>^front(<y>^R)

map(f)(S)    = <f(x) | x ∈ S>

filter(p)(S) = < x | p(x), x ∈ S >

rev(<>)   = <>
rev(<x>^R) = rev(R) ^ <x>
```

4.3 Basic Data Types and Operations

Now we are ready to define the core data structures relevant for our analysis problem: We introduce *keys* of agents as either *public* or *private* and introduce with *publicKeys* the set of all public keys of all agents in the system. A message *msg* is then either an elementary "raw" message (see above), or an address of an agent (used in packages sent to Mixes that extract the message in order to resend it), or an encryption of a message *msg* with a public key of an agent, or a data package composed of two messages *msg*, or just a dummy.

```
datatype keys= public.agents
             | private.agents

publicKeys   = {public.i | i ∈ agents}


msgs         = {1..msgno}

datatype msg = Msg.msgs
             | Addr.agents
             | K.publicKeys.msg
             | Join.msg.msg
             | Dummy
```

On messages, we define some auxiliary functions *wrap* and *unwrap*:

```
wrap(x,y)                 = Join.(Addr.(x)).y
unwrap(Join.(Addr.(x)).y) = (x,y)
```

that obey the following properties:

$$\forall x \in agents, y \in msg \bullet unwrap(wrap(x)) = x \tag{3}$$

$$\forall x \in agents \bullet wrap(unwrap(x)) = x \quad if \exists a, b \bullet Join.(Addr.(a)).b) = x \tag{4}$$

For convenience, the slightly unconventional dot-notation used in CSP_M for the application of constructors will be omitted in the sequel. Hence, we will write $Join(Addr(x), y)$ instead of FDR's output $Join.(Addr.(x)).x$ or — even more cryptic — $Join.Addr.x.x$.

4.4 Crypt- and Decrypt Primitives

Our analysis will be based on the "perfect cryptography"-assumption, i. e. we will only allow an agent to decrypt a message if he owns the private key. Thus, crypt- and decrypt functions can be modelled abstractly as constructors/destructors with suitable algebraic properties:

```
{- a : agents, m : msg -}
crypt(a,m)    = K.(public.(a)).m

{- crypted message, private keys -}
decrypt(K.(public.(mix.i )).m, private.(mix.j )) = if i≡j then {m} else {}
decrypt(K.(public.(user.i)).m, private.(user.j)) = if i≡j then {m} else {}
decrypt(_,_) = {}

cryptpath(<>,m)    = m
cryptpath(<a>^S,m)= wrap(mix.a, crypt(mix.a,cryptpath(S,m)))


strip(i,{}) = {}
strip(i,{(a,m)}) = if i ≡ a
                   then strip(i,{unwrap(x)|x ∈ decrypt(m, private.i)})
                   else ({(a,m)})
```

Note that the crypt/decrypt function pair fulfils the characterizing properties of "perfect cryptography":

$$decrypt(crypt(a, m), private(a)) = m$$

Further, we introduce the function $prefix$ that erases a prefix of package encodings with addresses to agent i. The necessity of this function will become apparent later: it will turn out to be necessary to suppress disastrous internal communication in MIX-stations and their spawned-off $SENDBUFFERS$ — see MIX1 in Section 4.6.

4.5 Abstracting the Set of Messages MSG

The size of the set of messages is a crucial factor for the size of the whole model. In a later section, we will discuss exact complexity measures that will reveal this dependency in more detail. Here, however, we will introduce several approaches to define the message set MSG. Within the sequel of attempts to define MSG, we will develop stronger and stronger abstractions of the model — making the model-checking based verification more feasible, by introducing more and more critical underlying assumptions.

A first straight-forward approach to define MSG is by enumerating the set along its inductive structure, up to a certain term-complexity $msgcxty$ which was defined in Section 4.1.

```
msgs_upto0(0) =          union(union({Msg.i   | i ∈ msgs},{Dummy}),
                               {Addr.i   | i ∈ agents})
msgs_upto0(x) = union(union(union({Msg.i   | i ∈ msgs},{Dummy}),
                               {Addr.i   | i ∈ agents}),
                      union({K.k.m     | k ∈ publicKeys, m ∈ msgs_upto0(x-1)},
                            {Join.a.b | a ∈ msgs_upto0(x-1),
                                        b ∈ msgs_upto0(x-1)}))

MSG0      = union({Dummy},msgs_upto0(msgcxty))
```

Based on $MSG0$, the analysis of the most simplest checks of the network will be unfeasible: the set contains already 4627 elements (for the setting of global constants as described in Section 4.1). This means that the automaton constructed out of the CSP_M specification will have at least a degree of 4627 !!! Depending on the version of our model, we have several thousands of states, which makes the automatic analysis for FDR unfeasible.

A first simplified version restricts $MSG0$ to the elements actually *used* in the protocols of the network. Hence, we apply an invariant that will be hidden in the sender-receiver communication relations in the network components. This abstraction is not

completely for free — it means that we restrict ourselves to "protocol- conform" messages — as if "junk-messages" will never lead to a successful attack against unobservability, which remains in fact to show. A non-protocol conform message is for example $K(public(mix(1)))(Join(Addr(user(1), K(public(user(2))(Msg(1))))))$: when it is received by $mix(1)$, it will be resent to $user(1)$ who will not be able to do anything with it since it is encrypted with the public key of $user(2)$. In our model, agents will simply ignore junk messages. These ideas lead to the following definition of $MSG1$:

```
msgs_upto1(0) = {wrap(u,K.(public.u).(Msg.m)) | m ∈ msgs,
                                                u ∈ users}
msgs_upto1(x) = union(msgs_upto1(x−1),
                      {wrap(u,K.(public.u).m) | u ∈ mixes,
                                                m ∈ msgs_upto1(x−1)})

ADRMSG1  = msgs_upto1(msgcxty)
MSG1     = union({Dummy}, {y | (x,y) ∈ {unwrap(z) | z ∈ ADRMSG1}})
```

The cardinality of this set is now just 15, which makes the analysis of many of the simpler analysis goals technically feasible.

For the more complex proof goals, it turns out that even this set is too large. Due to the fact, that only messages were used that were sent along the *user defined paths*, a number of messages is superfluous and can be suppressed. Assume, for example, $paths(_) = <1..mixno>$ and $mixno = 2$, then we can suppress from the MSG1-set constructed as below the elements 3,5,7,8,9, 11, 13, and 14:

```
MSG1 = {Dummy,                                                              (1)
        K(public(user 2))(Msg 1),                                           (2)
--      K(public(mix 1))(Join(Addr(user 1)(K(public(user 1)(Msg 1))))),     (3)
        K(public(mix 2))(Join(Addr(user 1)(K(public(user 1)(Msg 1))))),     (4)
--      K(public(mix 2))(Join(Addr(mix 2))
--                           (K(public(mix 2))
--                             (Join(Addr(user 1))
--                                  (K(public(user 1))(Msg 1))))),         (5)
        K(public(mix 1))(Join(Addr(mix 2))
                             (K(public(mix 2))
                               (Join(Addr(user 1))
                                    (K(public(user 1))(Msg 1))))),         (6)
--      K(public(mix 2))(Join(Addr(mix 1))
--                           (K(public(mix 1))
--                             (Join(Addr(user 1))
--                                  (K(public(user 1))(Msg 1))))),         (7)
--      K(public(mix 1))(Join(Addr(mix 1))
--                           (K(public(mix 1))
--                             (Join(Addr(user 1))
--                                  (K(public(user 1))(Msg 1))))),         (8)
--      K(public(mix 1))(Join(Addr(user 2))(K(public(user 2)(Msg 1)))),     (9)
        K(public(mix 2))(Join(Addr(user 2))(K(public(user 2)(Msg 1)))),     (10)
--      K(public(mix 2))(Join(Addr(mix 2))
--                           (K(public(mix 2))
--                             (Join(Addr(user 2))
--                                  (K(public(user 2))(Msg 1))))),         (11)
        K(public(mix 1))(Join(Addr(mix 2))
                             (K(public(mix 2))
                               (Join(Addr(user 2))
                                    (K(public(user 2))(Msg 1))))),         (12)
--      K(public(mix 2))(Join(Addr(mix 1))
--                           (K(public(mix 1))
--                             (Join(Addr(user 2))
--                                  (K(public(user 2))(Msg 1))))),         (13)
--      K(public(mix 1))(Join(Addr(mix 1))
--                           (K(public(mix 1))
--                             (Join(Addr(user 2))
--                                  (K(public(user 2))(Msg1))))),          (14)
        K(public(user 1))(Msg 1)                                            (15)}
```

This reduces the size of the message set by another factor 2.

Exploiting the idea, that only a limited set of paths is used in a model, we construct just the messages that can be constructed along the paths as follows:

```
MSG2 = union({Dummy},
             {snd(unwrap(cryptpath(p,wrap(u, K.(public.u).(Msg.m))))) |
                                  u1 ∈ users, u ∈ users, pp ∈ paths(u1),
                                  p ∈ suffixes(pp), m ∈ msgs })

MSG  = MSG2
```

The last line indicates that we will use this last version of the MSG-set throughout our analysis. The abstraction steps applied in this section turned out to be vital for the analysis as a whole.

4.6 The Components of the Mix Network: Channels, MIXes, SENDer, RECeiver

We are now in the position to define the general communication channel C that is used for all communications between agents in the network; it represents "the network" so to speak.

```
channel C    : agents.MSG.agents
```

Based on this channel, we introduce our first model of a MIX:

```
MIX0(i)(S) = if card(S) >= bufsize
             then SENDBUFFER(i)(S); MIX0(i)({})
             else C?x?m!i →
                     MIX0(i)(union(S,
                                   {unwrap(x) | x ∈ decrypt(m,
                                                            private(i))})
                            )
```

A MIX in CSP is a recursive process parameterised by a MIX-id i and a current state S that is a set of pairs containing the received messages in an "unwrapped" form, i. e. the topmost encryption envelope has been "ripped off" by decrypting it with the private key of the MIX. The resulting message package is decomposed into its future target address and the message content (both elements are put together in a pair). If the state contains more elements than the global constant $bufsize$, the state is transferred to a subprocess $SENDBUFFER$(to be described later); in this version, the MIX restarts sequentially its normal activity when $SENDBUFFER$ terminates; thus, the MIX is a purely sequential version. If the state S does not contain enough elements, the MIX listens to the global channel C to all communications addressed to him; if a message is sent to him, it is processed and inserted into the state. Note that multiply sent messages were ignored in this model since the state is a *set of pairs*. Note, moreover, that this sequential version of a MIX — albeit deadlock-free in itself — is deadlock prone in an environment, that does not admit the MIX to send. Such a situation occurs during a denial-of-service attack.

In order to avoid such problems, we try a MIX version that spawns off concurrently SENDBUFFER and immediately returns to listening for new messages.

```
MIX1(i)(S)  = if card(S) >= bufsize
              then SENDBUFFER(i)(S) [|SYNCS(agents,{i})|] MIX1(i)({})
              else C?x?m!i →
                      MIX1(i)(union(S,
                                    {unwrap(x) | x ∈ decrypt(m,
                                                             private.i)})
                             )
```

Unfortunately, this concurrent version produces a deadlock under a sort of denial-of-service attacks. FDR produces a counterexample that results in the creation of two

spawned off son processes SENDBUFFER that attempt to communicate with their father MIX while not agreeing in their communicating event (see 5.1;
goal `assert MIX1(mix.1)({}) :[deadlock free [F]]`)

A fix for the internal communication problem is a filter that eliminates messages that a Mix will have to send to himself: this filter — implemented by the *strip* function — erases such self-references when processing them for the internal state.

```
MIX2(i)(S)  = if card(S) >= bufsize
              then SENDBUFFER(i)(S) ||| MIX2(i)({})
              else C?x?m!i →
                      MIX2(i)(union(S,
                                    strip(i,
                                          {unwrap(x)| x∈decrypt(m,
                                                                private.i)}))
                             )
```

This concurrent version is what it should be: safe against denial-of-service and deadlock-free. Unfortunately, the state space is already infinite: a denial-of-service attack results in an unbounded spawn of SENDBUFFER's. Although any of these processes is finite and as a finite local state (S and its subsets, hence $2^{card(MSG)} * card(agents)$), the state space of the unbounded spawn is infinite, i.e. a finite, denial-of-service-safe, deadlock-free MIX does not exist.

Proof: A denial-of-service-attack consists of an arbitrarily long sequence of communications addressed to a particular MIX. This leads to the unbounded interleave:

```
SENDBUFFER(i)(S1)|||...|||SENDBUFFER(i)(Sn)|||MIX2(i)({})
```

that serves as a store that can reproduce any permutation of the sequence after the attack. Thus, the state space of the store must be infinite, although all local stores S_i are finite. However, a formal proof of deadlock-freeness could be done with HOL-CSP, on the basis of proof-rules such as

```
[| deadlock−free(P); deadlock−free(Q) |] ⇒ deadlock−free(P ||| Q)
```

Thus, it is a problem of FDR, *not* of CSP.

Therefore we return to a sequential design of Mixes (at least in our analysis). The following sequential version suppresses sendings to oneself. This avoids the internal deadlock problem. As a consequence, the network must be handled as a critical section, where the right for *sending* is owned only by one system component at a time.

```
channel grab, free : agents

SYNC  = grab?x → free!x → SYNC

-- WRITE(i:agents,m:MSG,j:agents)
WRITE(i,m,j) = grab.i → C.i.m.j → free.i   → SKIP
```

All components of the system will be syncronized with the process $SYNC$ that admits for one agent to *grab* the resource "access to C channel" and waits until the agent signalises via event $free$ that he released the resource. Writing on the C channel will only be performed via the $WRITE$-process. A final version of our MIX using channel synchronization looks as follows:

```
MIX4(i)(S)  = if card(S) >= bufsize
              then SENDBUFFER1(i)(S); MIX4(i)({})
              else C?x?m!i →
                      MIX4(i)(union(S,
                                    strip(i,
                                          {unwrap(x)| x∈decrypt(m,
                                                                private.i)}))
                             )
```

The un-synchronized and the synchronized versions of $SENDBUFFER$ reads as follows:

```
SENDBUFFER(i)(S)  = if empty(S) then SKIP
                    else ⊓ (a,m) : S • C!i!m!a
                                     → SENDBUFFER(i)(diff(S,{(a,m)}))

SENDBUFFER1(i)(S)  = grab!i → SENDBUFFER(i)(S); free!i → SKIP
```

Senders are processes indexed by user-ids (i.e. by a parameterized process). A SEND-process picks an arbitrary message, an arbitrary target user, an arbitrary path of its preconceived path-set, wraps the message along the path and sends the wrapped message to the first Mix of the path. All these decisions are represented by internal choices in CSP, and require the network to behave deadlock-free for all these choices. In the following, we present the standard version producing this behaviour and a synchronized one:

```
SEND1(i) = ⊓ m : msgs •
               ⊓ u : users •
                   ⊓ p : paths(i) •
                       let mess    = wrap(u, K.(public.u).(Msg.m))
                           (a,ms)  = unwrap(cryptpath(p,mess))
                       within  C!i!ms!a → SEND1(i)

SEND3(i) = ⊓ m : msgs •
               ⊓ u : users •
                   ⊓ p : paths(i) •
                       let mess    = wrap(u, K.(public.u).(Msg.m))
                           (a,ms)  = unwrap(cryptpath(p,mess))
                       within  WRITE(i,ms,a); SEND3(i)

REC(i)  = C?x?m!i → REC
```

The last line contains the receiver that simply reads everything addressed to him.

4.7 The Mix Network

In this subsection, we describe synchronization sets, and build a network with sender synchronization. The network will then be built as a sequence of parallel synchronization operations $P[|S|]Q$, where P and Q are forced to "communicate" (i. e. engage in one event in parallel) whenever events occur in S; otherwise, they may evolve in arbitrarily interleaved ways.

First, we introduce a shortcut for a set of communication events built over two subsets of agents S and T:

```
SYNCS(S,T)  = {C.i.m.j | i ∈ S, j ∈ T, m ∈ MSG}
```

Here we just present the variant of the network already containing sender synchronization. The core is a synchronization operator on all MIX processes, that must go in parallel whenever one MIX sends a C-event to another; every MIX starts with an initially empty buffer.

```
GEN_NET1(SEND,MIX,REC)
        =((||| i : users • SEND(i))

          [|SYNCS(users, mixes)|]

         (([|SYNCS(mixes, mixes)|] i : mixes •  MIX(i)({}))

          [|SYNCS(mixes, users)|]


          (||| i : users • REC(i))
         ))
         [|{|grab, free|}|]
```

```
        SYNC

NET1 = GEN_NET1(SEND3,MIX4,REC)
```

This core is combined with the row of senders on the one hand and the set of receivers on the other. All components in the system are then put in parallel with a semaphore represented by the $SYNC$-process that grants the exclusive right to send to some component as long as it does not release this right.

Theorem: This network is well-formed, i. e. deadlock-free.

Proof: See Section 5.1.

4.8 A Simple Instantiation of the Generic Passive Attacker

We are now ready to define the attacker that attempts to breach the security property the MIX-network is designed for, namely unobservability.

As an appetizer, we present a very simple attacker model that just stores the fact, that $user(i)$ has sent something and $user(j)$ received something. Initially, no one has sent or received a message. The attacker is successful if some user has received a message (T not empty), but there are fewer senders than required as minimal anonymity set. Required *anonymity quality* is a global constant controlling our key security requirement.

```
init_attack_state1 = ({},{})

incl1(user.i,_,_,S) = let (A,B) = S within (union({user.i},A),B)
incl1(_,_,user.i,S) = let (A,B) = S within (A,union({user.i},B))
incl1(_,_,_,S) = S

success1((S,T)) = (T≠{}) and (card(S) < anonymity)
```

Finding an attack based on this attacker model in our network is an easy game for FDR; since we have so far no control over the diversity of the senders and no dummy traffic in our model, FDR just constructs a scenario where $user(1)$ sends as the only sender several times a message to $user(2)$.

4.9 An Improved Instantiation of the Generic Passive Attacker

The previous attacker does not actually use knowledge over communication connections. In this section, we introduce a refined version of the attacker that "knows" of $bufsize$ and the sequential nature of the Mixes, and tries to keep track over possible communication relations (A communicated with B) within the network. The attacker exploits that

- received messages were retransmitted during the next shifts,
- and retransmitted messages had been received during the previous shift.

The general idea of our attacker is: The attacker keeps a *truncated log* and constructs a backchain-relation from this. During the attack, the truncated log serves as a fifo-queue (implemented as list in reverse order).

The attacker exploits the fact that if a Mix sends something, it is in sender mode (not valid in the non-sequential variant) which corresponds to a section of send-actions of this Mix in the trace. The length of this section is known to be $bufsize$. Skipping the

send-section, the previous receive-section can be constructed. Any received message in this receive-section leads to a backchain-relation, that is composed with the backchain of the previous operation.

Due to truncation, it is possible that receive-mode-sections in the trace are incomplete, in this case we assume the least information the attacker may use, i.e. the full set of participants in the net.

For the necessary finitization of the data-model, we introduce the global constant *anonymity* controlling the "size of the anonymity set" in the sense of Section 2. With *search_depth* we characterize the "memory" of the attacker — i.e. the length of its internal "truncated log" that he uses to actually construct the backchain-relation:

```
anonymity = 2

search_depth = 3
```

We will use the following auxilliary functions on truncated logs in order to define our attacker. *skip* erases all prefixing communication packages (triples of sender, message and receiver) that were not addressed to some agent m. *search_send* constructs the set of possible senders that contributed to a full buffer of a Mix. In principle, *backchain* builds the transitive closure of the direct communication relation computed by *search_send*. In all cases, whenever there is not enough information due to the truncation of the log, the algorithm assumes the worst case in order to produce a backchain-relation in any case that *contains* the communication relation, i.e. the abstraction due to finitization is in any case safe.

```
-- skip (m:mixes,<(j:agent,m:MSG,i:agent)>^R)
skip (m, <>)          = <>     -- no receiving occurrence found due to truncation
skip (m,<(j,_,i)>^R) = if m ≡ i
                       then <(j,m,i)>^R  -- first receiving
                                         -- occurrence found
                       else skip(m,R)

--search_send(m:mixes, R)
search_send(m, R) =
   let cs(0,_)     = {}
       cs(_, <>)   = {}
       cs(x,<(user.j,_,i)>^R) = if i ≡ m
                                then union({user.j}, cs(x-1, R))
                                else cs(x,R)
       cs(x,<(mix.j,_,i)>^R)  = if i ≡ m
                                then union({mix.j}, cs(x-1, R))
                                else cs(x,R)
   within cs(bufsize,R)

--backchain(a:agent,<>)
backchain(a,<>) = users
backchain(a,<(user.j,m,i)>^R) = if a ≡ i     -- something was sent to target a
                                then {user.j}
                                else backchain(a,R)
backchain(a,<(mix.j,m,i)>^R)  = if a ≡ i     -- something was sent to target a
                                then(let S = search_send(mix.j,skip(mix.j, R))
                                          -- sniff ...
                                     within (if card(S) < bufsize
                                             -- we have only partial
                                             -- information
                                             then users  -- assume the worst
                                             else (union (
                                                     { user.x | user.x ∈ S},
                                                   Union({backchain(mix.x,R)
                                                          | mix.x ∈ S})
                                                  ))   -- backchain to the
                                                       -- predecessors
                                            )
                                     )
                                else backchain(a,R)
```

Based on this machinery, it is now straight-forward to build the instance of our generic passive attacker. The inclusion operation *incl* just prepends a new communication package and truncates the log. The predicate *success* is defined by having less elements in the backchain set (i. e. the set of agents in backchain-relation to some agent a) than *anonymity*. The initial state is a truncated log filled with dummy elements of length *search_depth*.

```
incl(x,y,z,S)= <(x,Dummy,y)>^(front(S))
success(<(mix.j,_,user.i)>^R) = card(backchain(mix.j,R)) < anonymity
success(_) = false

init_attack_state = < (mix.1,Dummy,mix.1) | x ∈ <1..search_depth>>
```

Analysing our network with this attacker brings FDR to its limits. On a large machine, the analysis terminates and constructs a similar attack as the simple attacker, but the complex attacker clearly needs further improvement and further suitable abstractions.

5 Analysis

As mentioned in Section 3.2, the proof of a security property comes in two parts: First, the consistency of the system to be analysed must be assured. This boils down to a proof of deadlock freeness. Second, the system together with the attacker (incorporating the attacker model) must refine the universal process that contains all communication sequences not containing *success*.

5.1 Consistency Analysis of MIXes, SENDer and RECeiver

Here, we list the proof goals concerning constistency of our analysis. FDR displays them in an own window allowing the user to click on them — this activates the internal proof procedure, and may or may not terminate (easily).

```
assert MIX0(mix.1)({})          :[deadlock free [F]]
assert MIX1(mix.1)({})          :[deadlock free [F]]
assert MIX2(mix.1)({})          :[deadlock free [F]]
assert MIX3(mix.1)({})          :[deadlock free [F]]
assert MIX4(mix.1)({})          :[deadlock free [F]]

assert SENDBUFFER (mix.1)({}):[deadlock free [F]]
assert SENDBUFFER1(mix.1)({}):[deadlock free [F]]

assert SEND1(user.1)            :[deadlock free [F]]
assert SEND2(user.1)            :[deadlock free [F]]
assert SEND3(user.1)            :[deadlock free [F]]

assert REC(user.1)              :[deadlock free [F]]
```

MIX0 is immediately deadlock free, while MIX1 produces a counter example (see previous discussion) and MIX2 does not terminate since the model is not finite. The resulting revisions MIX3 and MIX4 of MIX0 are again deadlock free. All other processes are trivially proved deadlock free.

5.2 Consistency Analysis of the Network

The whole network (in the version presented in this paper) is also deadlock free.

```
assert NET1                    :[deadlock free [F]]
```

The analysis takes several minutes on a Pentium III with 800 Mhz.

5.3 Analysis of attacks against Anonymity

We turn now to the core of the analysis, the proof that the attacker is not successful. In our case — the senders are just interleaved, no control of sender distribution is included in our model so far — FDR constructs counter examples representing possible attacks. Note that the synchronization events *grab* and *free* must be hidden in all synchronized versions of Mix networks.

```
assert RUN ⊑T(NET1[| {| C |} |]
              ATTACK (success,incl,init_attack_state)) \ {|grab, free |}
assert RUN ⊑T(NET1[| {| C |} |]
              ATTACK (success1,incl1,init_attack_state1)) \ {|grab, free |}
```

These goals lead to counter examples such as:

```
C(user 1)(K(public(mix 1))
          (Join(Addr(mix 2))
               (K(public(mix 2))
                 (Join(Addr(user 1)(K(public(user 1)(Msg 1))))))))
 (mix 1)
C(user 1)(K(public(mix 1))
          (Join(Addr(mix 2))
               (K(public(mix 2))
                 (Join(Addr(user 2)(K(public(user 2)(Msg 1))))))))
 (mix 1)
C(mix 1)(K(public(mix 2))(Join(Addr(user 1)(K(public(user 1)(Msg 1))))))(mix 2)
C(mix 1)(K(public(mix 2))(Join(Addr(user 2)(K(public(user 2)(Msg 1))))))(mix 2)
C(mix 2)(K(public(user 1)(Msg 1)))(user 1)
Success
```

In this counter example, only one sender sends to one receiver into the network — and there can not be unlinkability in this network for this reason.

6 Conclusion

6.1 Discussion

The main result of our work is a formal model of a Mix network suitable for a machine based analysis. It can be used to find known and possibly new attacks. Standard models and standard formal notions may improve the understanding of informal ones. Moreover, they pave the way for the formal analysis of Mix implementations. Further, we have a proof of concept of an adoption of existing protocol analysis techniques to security properties such as anonymity and unobservability. We see our work as a step to turn the analysis of this type of security properties into a routine task, similarly to the analysis of authentification protocols.

In comparison to standard analysis in the field of authentification, anonymity and unobservability are particularly difficult to analyse by model-checking techniques, and the necessary simplifications and abstractions seem to be more distant to real-world assumptions. This is a consequence of the fact that anonymity and unobservability are inherently understood in terms of "large numbers", e. g. large anonymity sets, large

buffers of Mixes and large memories on the side of the attacker — this leads to large state spaces to be handled and to be abstracted.

The situation will be even worse for a probabilistic analysis — for this type of problem, we predict that purely model-checking based approaches attempting to simulate Markow automata will not be feasible; therefore formal probabilistic analysis will require theorem proving environments with theories and proof support for probabilism and Markow automata.

We believe that anonymity and unobservability analysis — based on model- checking — must be based on relatively concrete attacker models (i. e. non-deterministic "algorithms").

6.2 Future Work

We would like to distinguish two types of extensions: More or less straight-forward improvements of our approach and more general achievements for the field of formal analysis of Mixes as a whole. The following list of items belongs to the former class:

- adaption of our analysis to receiver anonymity,
- application to unobservability by introducing dummy traffic,
- simplified versions of more powerful,"intelligent" attackers,
- including assumptions on the diversity,
- using active attackers for constructing replay-attacks etc.,
- the analysis of an existing Mix protocol like Onion Routing [6]
- finding Mix models that are closely connected to (abstract) real implementations.

In the latter class, we like to name the following issues:

- finding a systematization of attackers or something like a "most general attacker" against anonymity (probably not model-checkable),
- alternative, theorem-proving based approaches for the analysis of Mixes, and
- extensions of our possibilistic framework towards a probabilistic one.

References

[1] D. Basin. Lazy infinite-state analysis of security protocols. In *Secure Networking — CQRE [Secure] '99*, LNCS 1740, pages 30–42. Springer, Berlin, 1999.

[2] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions in Computer Systems*, 8(1):18–36, 1990.

[3] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[4] L. Cottrel. Mixmaster & remailer attacks. URL: `http://www.obscura.com/ loki/remailer-essay.html`, 1995.

[5] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[6] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM*, 42(2):39–41, 1999.

[7] N. Heintze, J. Tygar, J. M. Wing, and H.-C. Wong. Model checking electronic commerce protocols. In *Proceedings of the USENIX 1996 Workshop on Electronic Commerce*. 1996.

[8] C. A. Hoare. *Communicating sequential processes*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[9] R. Kemmerer. Using formal methods to analyze encryption protocols. *IEEE Journal of Selected Areas in Communication*, 7(2):448–457, 1989.

[10] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[11] D. Kesdogan. *Vertrauenswürdige Kommunikation in offenen Umgebungen*. PhD thesis, RWTH Aachen, Mathematisch-Naturwissenschaftliche Fakultät, 1999.

[12] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocols using FDR. In T. Margaria and B. Steffen, editors, *Proceedings of TACAS'96*, LNCS 1055, pages 147–166. Springer, Berlin, 1996.

[13] F. S. E. Ltd. Failures-divergence refinement – FDR2 user manual. Available at the URL `http://www.formal.demon.co.uk/FDR2.html`.

[14] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols. In *Proceedings of the DIMACS Workshop on Design and Verification of Security Protocols*. 1997.

[15] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 19, 1994.

[16] J. K. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, 1987.

[17] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[18] L. C. Paulson. *Isabelle: a generic theorem prover*. LNCS 828. Springer, Berlin, 1994.

[19] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[20] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology. URL: `http://www.koehntopp.de/marit/pub/anon/Anon_Terminology.pdf`, 2001.

[21] A. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of 1995 IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1995.

[22] A. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, Englewood Cliffs, NJ, 1998.

[23] A. Roscoe and M. Goldsmith. The perfect spy for model-checking crypto-protocols. In *Proceedings of DIMACS workshop on the design and formal verification of cryptographic protocols*. 1997.

[24] S. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.

[25] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of FME '97*, LNCS 1313, pages 318–337. Springer, Berlin, 1997.