Robustness Modelling and Verification of a Mix Net Protocol

Efstathios Stathakidis[1], Steve Schneider[1] and James Heather[2]

[1] {e.stathakidis, s.schneider}@surrey.ac.uk
Computing Department, University of Surrey, Guildford, UK
[2] james@chiastic-security.co.uk
Chiastic Security Ltd, Guildford, UK

Abstract. Re-encryption Mix Nets are used to provide anonymity by passing encrypted messages through a collection of servers which each permute and re-encrypt messages. They are used in secure electronic voting protocols because they provide a combination of anonymity and verifiability. The use of several peers also provides for robustness, since a Mix Net can run even in the presence of a minority of dishonest or incorrectly behaving peers. However, in practice the protocols for peers to decide when to exclude a peer are complex distributed algorithms, and it is non-trivial to gain confidence that the Mix Net will be robust and live in the presence of faulty or malicious peers. In this paper we model and analyse the algorithm used by a beta version of *Ximix*, a particular Mix Net implementation, using the CSP process algebra and the FDR model checker. We model and analyse the protocol in the presence of a realistic intruder based on Roscoe and Goldsmith's perfect *Spy* [1]. We show that in the beta version of the implementation the protocol does not satisfy the robustness requirement. Finally, we propose a method of making it robust, and verify in FDR that the proposed solution is sound and provides this robustness. Along the way, we highlight the omissions and deviations from the original RPC proposal; Mix Net protocols are extremely fragile, and small and seemingly benign changes may result in security flaws. Our experimental results show that, with our modification, Ximix guarantees termination and produces a correct output in the presence of an intruder who can corrupt a minority of mix servers.

Keywords: Mix Nets, formal methods, model-checking, CSP, FDR

1 Introduction

Since ancient times, elections have been the most important aspect in ensuring democracy. A voting system should provide voters with secrecy of the ballot while assuring the integrity of the election.the voter with the assurance that her vote has been cast as intended and included in the tally without being modified, whilst guaranteeing the secrecy of the vote. Recent proposals for secure electronic voting aim to provide *end-to-end verifiability* using cryptographic techniques, and can

use *anonymising Mix Nets* to provide secrecy of the ballot (by anonymising which voter has cast any particular vote) while also providing the assurance that the votes have been decrypted correctly. The upcoming elections in the State of Victoria, Australia, will be the world's first large-scale political elections where a verifiable electronic voting system will be used. A key component in achieving this is the *Mix Net* and, of course, this should be robust and produce its required output. However, this key liveness property is generally not analysed in the literature, and it is one that the Victorian system's Mix Net (Ximix) is required to provide.

A *Mix Net* is a cryptographic protocol, which *unlinks* the correspondence between its input vector of encrypted values and the permuted vector of decrypted values given as output, thus providing anonymity to the communicating entities. The first Mix Net was introduced by Chaum [2] for constructing anonymous mail systems. In its general construction, a Mix Net consists of a sequence of servers $M_1 \dots M_n$, also called mix servers, that collectively execute a protocol. Based on the way the mix servers operate on the input ciphertexts, Mix Nets are classified as *decryption* and *re-encryption* Mix Nets. However, most of those proposed in the literature fall into the second category. We briefly explain how a re-encryption Mix Net works; for more details about decryption and re-encryption Mix Nets, we refer the reader to [2, 3].

The first re-encryption Mix Net was introduced by Park *et. al* [4]. In this type of Mix Net, a joint public-key is generated by combining the public-keys of the mix servers. The inputs are encrypted under the joint public-key and then submitted to the Mix Net. Each mix server, in turn, re-encrypts its inputs, shuffles them using its own secret permutation and fresh randomnesses and then posts them onto a publicly accessible Web Bulletin Board ($\mathcal{WBB}$). Once all the mix servers have finished their mixing, the decryption phase starts, where the final list of ciphertexts is decrypted in a distributed manner to achieve robustness and then posted on the $\mathcal{WBB}$ for public verification. To ensure correctness of the execution, each mix server produces a zero-knowledge proof, which is posted on the $\mathcal{WBB}$ alongside the mixed and decrypted messages.

Owing to their importance in providing anonymity to the communicating parties, Mix Nets play a significant role in building systems where security requirements, such as privacy, should hold. Their main application is in electronic-voting [5–8], but they have been also used in other real-life applications, such as: electronic cash payments, RFID tags and anonymous Web browsing. In electronic voting schemes, Mix Nets are used to ensure that no one can track and reveal a voter's vote, thereby guaranteeing the privacy of the vote and the anonymity of the voter. However, this is not always enough; a well constructed Mix Net should fulfill a number of security and safety requirements, such as robustness, correctness and public verifiability. A Mix Net is called *robust* if it terminates and produces a proof of the correctness of the operation in the presence of (a limited number of) faulty or malicious mix servers. *Correctness* guarantees that the output is, indeed, a valid permutation of the input ciphertexts. Additionally, it is crucial for misbehaviour to be detectable by anyone who is interested in

checking the correctness of the execution, a property called *public verifiability*. In this work we are interested in the liveness properties that a Mix Net should meet; a Mix Net which does not produce any output is of no interest.

For the first time in the literature, we construct here a formal model and present an automated verification of the beta version of the Ximix[3] Mix Net; the production version will be used in the elections in Victoria State, Australia, in November 2014. The Victorian Electoral Commission's (VEC) vVote voting system, which uses Ximix, is based on the Prêt à Voter [6] voting scheme and how it works is presented in [7, 8]. It is a requirement that Ximix be robust and be able to produce a correct result provided that a threshold number of mix servers are available and follow the protocol without deviating from it. Auditing in Ximix is performed according to the Randomised Partial Checking (RPC) auditing technique [9] and its source code is available at `http://www.cryptoworkshop.com/ximix/doku.php`.

There are numerous different Mix Net proposals existing in the literature. Each of those designs follows a different method in constructing such schemes, and it is not always clear which of the desired security requirements are met, or how best to resolve some of the implementation questions such as what to do if a mix server fails. For example, a Mix Net based on randomized partial checking for verification does not normally indicate where the random challenges come from, yet this is a subtle issue in practice and an inappropriate approach can undermine the security of the Mix Net. Standardisation of Mix Nets would address issues such as this, providing some clear direction for developers and thus confidence for users.

Once such questions are addressed, the advantages and benefits of Mix Nets being standardised will be apparent. Having a standard for Mix Nets would provide a reference point for future implementations and the properties that they provide. For example, Verificatum [10], the most advanced Mix Net implementation up to date, has been used in small-scale elections in Norway and Israel. Hence, in the case where Verificatum would become a standard, techniques like RPC verification would also become standardised and not leave up to the constructor to decide. The benefits of Mix Nets having a standard is also apparent from the subject matter of this paper: the approach taken to the development of Ximix, the Mix Net under analysis, was to combine results from a number of different research papers, and to provide additional implementation detail, resulting in a system which required further formal analysis. Its correctness relies entirely on the programmers and the way they interpret and implement the proposed techniques. In the presence of a Mix net standard, we might hope that these opacities could have been avoided.

[3] The modelling was performed on a beta version of Ximix, using a snapshot of the source code taken on 7 January 2014. The Ximix production code incorporates a number of changes since the beta version, in response to various reviews.

2 Preliminaries

Here, for convenience, names, sets, data and functions used throughout the model are detailed. The set of all mix servers is denoted by $\mathcal{P}$ and is defined to be $\mathcal{H} \cup \mathcal{D}$, where $\mathcal{H}$ (resp. $\mathcal{D}$) denotes the set of all honest (resp. dishonest) mix servers. By $\mathbb{P}\mathcal{A}$ we denote the powerset function as applied to a set $\mathcal{A}$. By $\mathbf{c}$, we denote the unmixed vector of inputs and by $M_j(m)$, we denote the vector m mixed twice using mix server j's secret permutation values, where m is either the unmixed vector $\mathbf{c}$ or some received signed mixed vector thereof. The set of all commitments to the secret permutation values is denoted by $\mathcal{C}$. When sending to a mix server, these commitments are individually hashed, thereby creating then a hash value, here $h(commit)$. Messages have the form $\mathsf{S}_j(sk_j, M_j(m))$, where sk_j is the jth mix server's secret key. The set of all messages that can be feasibly sent and received in a protocol run is denoted by $\mathcal{M}$. For example, consider the channel name $comm$ of type $\mathcal{P} \times \mathbb{P}\mathcal{P} \times \mathcal{M} \times \mathcal{C}$. In this scenario, $comm.A.\{B, C\}.\mathsf{S}_A(sk_A, M_A(m)).\mathsf{S}_A(sk_A, h(commit))$ may be an event indicating that a vector of messages, m, has been sent by the mix server A to $\{B, C\}$. Additionally, the length of a message m is denoted by $\#m$ and is calculated by counting the layers of signatures. The outer signatory of a message is verified using the corresponding public key and we use a function $outer(m)$ to return the signatory of a given message m. Similarly, the function $seq(m)$ returns the outer mix sequence of m. The function $val(id)$ returns the index of a given mix server in the shuffle plan. The function $prev(id)$ returns the identity of the preceding mix server. The set of all possible partially decrypted mixed messages, $\mathcal{L}$, is taken to be $\{\mathsf{P}_j(pt_j, seq(m), zkp) \mid m \in \mathcal{M}\}$, where pt_j is the jth mix server's share of the distributed secret key and zkp is the associated zero-knowledge proof, thus proving the correctness of the partial decryption. Finally, the set of all possible fully decrypted messages, $\mathcal{O}$, is considered to be $\{dec(m) \mid m \in \mathcal{L}\}$, where $dec(m)$ is a function that decrypts a permuted vector of ciphertexts into a permuted vector of plaintexts.

3 Ximix Mix Net Outline

Ximix is an Elliptic Curve El Gamal [11] based re-encryption Mix Net written in Java, where the main idea behind its design follows the RPC auditing technique. It consists of a number of execution phases: (i) *initialisation*; (ii) *mixing*; (iii) *checking* and; (iv) *decryption*. We will analyse each of these separately. There are two more important components of Ximix:

1. the Command Service;
2. a Transient Board for each mix server.

The *Command Service* ($\mathcal{CS}$) is the central trusted component of the system, which is responsible for talking to the mix servers and instructing them to mix, transmit their output list of ciphertexts to another mix server and to create *Transient Boards* ($\mathcal{TB}$) to host their produced data. The $\mathcal{CS}$, which is under the

control of the VEC, has a great deal of power, controls the data flow, controls the whole process and specifies the execution *plan* (also called *shuffle plan*) used to emulate the RPC pairing of mix servers. In a setting with four mix servers *A, B, C* and *D*, the shuffle plan looks like $\langle AA, BB, CC, DD \rangle$. Robustness in the Ximix implementation relies heavily on the $\mathcal{CS}$, as it is a single point of trust and a single point of failure. Any misbehaviour could potentially lead to the Mix Net's crash, thus violating the robustness requirement. Additionally, in Ximix, the mix servers exist to provide services, including board hosting. That is, as the shuffle plan progresses, a new $\mathcal{TB}$ is created on the appropriate mix server (as specified in the shuffle plan) to host the intermediate and output data (shuffled messages and commitments to the secret permutation). The *primary* mix server is responsible for sending the unmixed data to the first mix server in the shuffle plan and it is also responsible for maintaining a *Visible Board* ($\mathcal{VB}$), which differs from its internal $\mathcal{TB}$ in that all the mix servers have read access to this. At the end of the process, the contents of the last mix server's $\mathcal{TB}$, as well as all the partially decrypted messages, are posted to the $\mathcal{VB}$. How these components interact with each other, in the case where all faithfully follow the protocol, is illustrated (for the case of two mix servers) in Figure 1. One can note that actions internal to the mix servers and the $\mathcal{CS}$, such as receiving a message, mixing and then posting it onto the corresponding $\mathcal{TB}$ have been abstracted away and shown as self messages. Additionally, for clarity, all the instructions the $\mathcal{CS}$ can send and the mix servers are willing to accept, have been presented as *instruct* events.

4 Ximix Message Communication Diagram

For clarity, the diagram of Figure 1 illustrates how the data flows in a faithful run of Ximix with two mix servers.

Initialisation Phase. When the execution starts, the $\mathcal{CS}$ selects the shuffle plan and chooses the primary mix server. The original unmixed data are handed to the Mix Net by the VEC authorities and the $\mathcal{CS}$ instructs the primary mix server to transmit them to the head of the shuffle plan. The mixing phase now begins.

Mixing Phase. The mixing phase starts with the first mix server in the shuffle plan re-encrypting and shuffling its input data using fresh randomness and secret permutation values. As the shuffle progresses, each mix server is asked by the $\mathcal{CS}$ to create a $\mathcal{TB}$ to store its own shuffled messages and commitments to secret permutations. Once the data have been stored on the $\mathcal{TB}$, the $\mathcal{CS}$ instructs the mix server that is currently mixing to transmit them to the next mix server in the plan. When the mixing phase is complete, only the shuffled data of the last transient board in the plan are copied onto the $\mathcal{VB}$. In the beta version analysed here, neither the data transmitted between the mix servers nor the

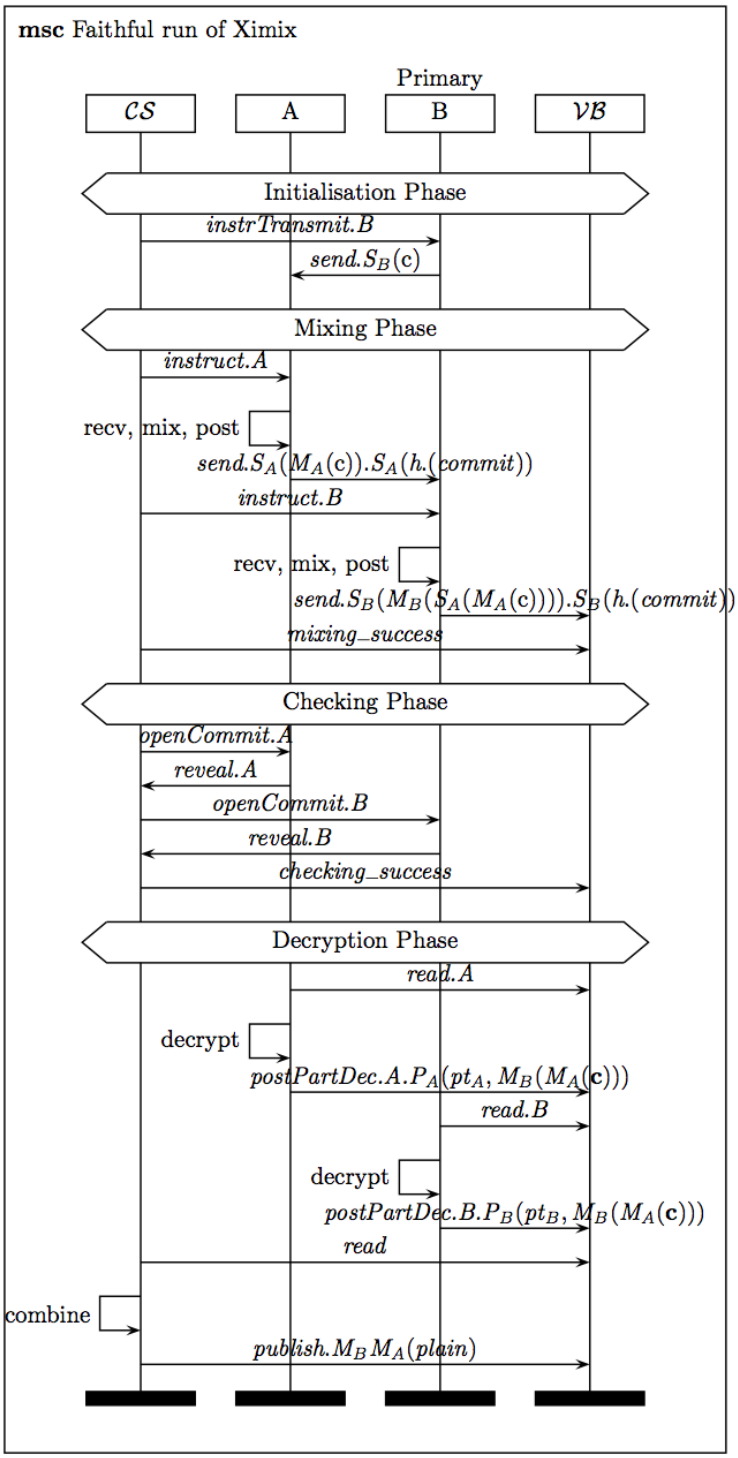

Fig. 1: Ximix protocol with two servers

final sequence of mixes is posted anywhere for public verification. Consequently, it does not provide universal verifiability[4].

Checking Phase. The Mix Net uses static verification, i.e. each mix server maintains separate sets of commitments to the secret permutation values. At the end of the mixing phase, these commitments are downloaded by the $\mathcal{CS}$ and checked for consistency. No attempt to verify anything is made during the actual mixing phase (verification *not in-phase* with mixing). This stands in contrast to the approach taken by Verificatum, where checking is *in-phase* with mixing and each mix server checks the received data before proceeding to the mixing. In Ximix, the checking is done in the order of the mixing, but after all commitments have been downloaded. The $\mathcal{CS}$ issues periodic RPC challenges as it goes, and up to half the input-output relationships are revealed for each mix server. This is done using an interactive zero-knowledge protocol [12] between the $\mathcal{CS}$, but the produced proof convinces only the verifier ($\mathcal{CS}$), which interacts with the prover (mix servers). As a consequence, the $\mathcal{CS}$ cannot prove to a third party the correctness of the execution, even if all the mix servers have faithfully followed the protocol[5]. If cheating is found, the whole process stops and an operator's intervention is required; the corrupt mix server is replaced and the whole process restarts. Otherwise, the execution proceeds with the decryption phase.

Decryption Phase. Assuming checking success, the data from the $\mathcal{VB}$ are downloaded by the $\mathcal{CS}$ and decrypted, by a quorum (threshold number) of mix servers, using El Gamal threshold decryption, whereby each is asked to provide a partial decryption of the final list of encrypted data. In Ximix, mix servers act as decryption servers, in that they partially decrypt what has been output by the last mix server, but they send their partial decryptions only to the $\mathcal{CS}$. Once the $\mathcal{CS}$ has received a threshold number of partial decryptions for each message, it assembles them into the plaintext messages and publishes the fully decrypted message(s). As part of the decryption process, periodic challenges are issued against the partial decryptions, using an interactive zero-knowledge proof protocol (IZKP) between the $\mathcal{CS}$ and the mix servers.

5 Modelling and Formal Analysis in CSP

In the preceding section, Ximix was described based on how it operates and processes the input messages. In this section, we present the processes modelling the individual components of Ximix, and how these are composed into models to be checked for robustness. Before proceeding to the modelling, it is pertinent to discuss the modelling decisions and assumptions we will make.

[4] The production version of Ximix does provide transcripts of the stages of the mix and hence provides universal verifiability.

[5] The production version of Ximix does provide a non-interactive proof, appropriate as a proof to a third party.

Modelling Decisions and Assumptions. We follow the typical approach to modelling of security properties, treating cryptographic primitives, such as digital signatures, encryption and decryption, as *symbolic operations* with the appropriate algebraic properties. Each component is modelled as an individual process; the communications between them are *synchronous* and over *authenticated* channels. We assume that the *checking* phase will always succeed. No participant wishes to be expelled from the protocol, and for this reason they provide the checker with valid commitments to secret permutations when asked to reveal a subset of them.

Faithful Model of Ximix. Ximix will be used under the strong assumption that all its components faithfully follow the protocol without deviating from it. Additionally, VEC can tolerate failure of one mix server, which provides us with our *threshold* value. Before the execution starts, the shuffle plan is fed to the mix servers so they know their position and the neighbouring mix servers. The complete script for this model can be found at `http://www.tvsproject.org/csp/Ximix_faithful.csp`.

Honest mix servers. An honest mix server waits for an instruction, sent by the $\mathcal{CS}$, to be received and then starts to operate. It is willing to receive any structurally correct message signed by the previous mix server in the plan and then posts the received message on its $\mathcal{TB}$. Obviously, the head of the plan is willing to receive the unmixed data signed by the primary mix server. This is represented in *HON_SRVR* by the external choice over the set of the messages signed by the sender and the set of the hashed commitments. Then, $\mathcal{CS}$ instructs the first mix server to mix the received data and the execution proceeds to the mixing phase.

$$
\begin{array}{l}
HON\_SRVR(id) = \\
\quad \textbf{if } id == primaryServer \textbf{ then} \\
\qquad \Box_{\substack{m \in \{m' \in \mathcal{M} \mid \#m' = 0\}, \\ commit \in \mathcal{C}}} \begin{pmatrix} send.\mathsf{S}_{id}(sk_{id}, m).\mathsf{S}_{id}(sk_{id}, h(commit)) \rightarrow \\ MIX(id, 1, m) \end{pmatrix} \\
\textbf{else} \\
\qquad CS\_instructsToCopy.id \rightarrow \\
\qquad\quad \Box_{\substack{m \in \{m' \in \mathcal{M} \mid outer(m') = prev(id)\}, \\ commit \in \mathcal{C}}} \begin{pmatrix} recv.m.commit \rightarrow \\ postInData!id!m!commit \rightarrow \\ CS\_instructsToShuffle.id \rightarrow \\ MIX(id, val(id), m) \end{pmatrix}
\end{array}
$$

The mix server mixes the received data twice (in order to emulate the RPC pairing) and the $\mathcal{CS}$ then instructs it to create a $\mathcal{TB}$ to store the output data for future verification. Once this has been done, the $\mathcal{CS}$ requests the mix server to transmit only the output message and the commitments to the secret permutations to the next mix server in the shuffle plan. This is modelled in the *MIX* process below. Here, the *toBeMixed* value indicates the message received from the previous mix server, which will be mixed twice by the current one. The *rnd* symbolises the round number of the mixing; a mix server proceeds to the mixing phase only when this value equals its own number in the sequence.

Command Service. The $\mathcal{CS}$ sends instructions to the mix servers and announces the phase's success or failure. The instructions are sent sequentially and at the end of the execution of each phase and a *done* event indicates its successful completion.

$$CS(\langle\rangle) = mixing\_done \rightarrow STOP$$
$$CS(\langle id\rangle \frown ids) = \left(\begin{array}{l} CS\_instructsToCopy.id \rightarrow \\ CS\_instructsToShuffle.id \rightarrow \\ CS\_instructsToCreateTB.id \rightarrow \\ CS(ids) \end{array}\right)$$

$$MIX(id, rnd, toBeMixed) = \left(\begin{array}{l} CS\_instructsToCreateTB.id \rightarrow \\ postInterData!id!interData!interCommit \rightarrow \\ postOutData!id!outData!finalCommit \rightarrow \\ send.\mathsf{S}_{id}(sk_{id}, outData).\mathsf{S}_{id}(sk_{id}, h(finalCommit)) \rightarrow \\ SKIP \end{array}\right)$$

Transient and Visible Board. As we have described earlier, each mix server maintains a $\mathcal{TB}$, which always allows *post* and *read* requests from the owner mix server. The other mix servers are blind on what has been posted to the other $\mathcal{TB}$s and when the execution starts, they are both empty.

$$TB(id, inData, interData, outData, interCommit, outCommit) = \Box_{\substack{inD, interD, outD \in \mathcal{M} \\ inC, outC \in \mathcal{C}}} \left(\begin{array}{l} postInData!id.inD.inC \rightarrow \\ postInterData!id.interD.interC \rightarrow \\ postOutData!id.outD.outC \rightarrow \\ TB(id, inData \bigcup inD, interD, outD, interC, outCommit \bigcup inC) \end{array}\right)$$

The $\mathcal{VB}$ is visible to all mix servers as well as to the $\mathcal{CS}$. The mix servers *read* what has been posted there and *post* their partial decryptions. The $\mathcal{CS}$ reads all the partially decrypted messages, combines them and *outputs* (posts) the fully decrypted messages on the $\mathcal{VB}$. (For brevity, the partial decryption and the combination processes are run in the $\mathcal{CS}$'s side and are not presented here; we refer the reader to the full script.)

$$VB(primaryServer, outData, outCommit, partDec) = \Box_{\substack{id \in \mathcal{P}, outD \in \mathcal{M}, \\ outC \in \mathcal{C}, newPartDec \in \mathcal{L}}} \left(\begin{array}{l} postDataToVB.id!outD!outC \rightarrow VB(\ldots, outD, outC, \ldots) \\ \Box\ readFromVBData.id!outData \rightarrow VB(\ldots) \\ \Box\ postPartDec.id!newPartDec \rightarrow VB(\ldots, partDec \bigcup newPartDec) \\ \Box\ CSreadFromVBPartDec!partDec \rightarrow VB(\ldots) \end{array}\right)$$

Putting the Network Together. Based on the modelling assumptions presented in Section 5, and connecting the channels of the various processes so they can synchronise, we can produce our final $SYSTEM_{Ximix}$ process, which is defined in terms of the parallel composition of all the following processes, synchronised on their common events. The *MIXING* process is defined as the parallel composition of the *HON_SRVR* and the *TB* processes.

$$SYSTEM_{Ximix} = MIXING \,||\, CHECKING \,||\, DECRYPT \,||\, COMBINE \,||\, VB(\ldots) \,||\, CS(ShufflePlan)$$

We should now verify whether our system satisfies the robustness requirement or not and to check this, we need to determine whether a fully decrypted message is always output. In [13], the output of the system was modelled as a synchronous *agree* event among the majority of the mix servers, whilst here, the $\mathcal{CS}$ is responsible for combining all the partial decryptions and publishing the fully decrypted message(s). No consensus among the mix servers is provided

for. For this purpose, we create the specification process $RBST$, which always performs an *output* event, and use the Failures/Divergences refinement model ($\mathcal{FD}$), to check in FDR that the following assertion holds:

$$RBST = output \rightarrow RBST, \quad RBST \sqsubseteq_{FD} (SYSTEM_{Ximix} \setminus \Sigma \backslash \{| \, publish \, |\}) [\![{}^{output}/_{publish.m} \, |_{m \in \mathcal{O}}]\!]$$

To perform a rigorous analysis of the system, we must include an intruder model. To this end, in the next section, we introduce our threat model and place it in parallel with the $SYSTEM_{Ximix}$ process defined here.

6 Adapting the Intruder

In the previous section, we modelled and verified Ximix under the assumption that all the components are honest. However, this is a strong assumption; a system consisting of honest participants is of little interest. In this section, we use Roscoe and Goldsmith's perfect *Spy* [1, 14] as the basis of our threat model and investigate whether Ximix still meets the robustness requirement. The description of the *Spy* presented in the first half of this section it will be used as well in the second, subject to some minor modifications, in order to accommodate the behaviour of the proposed scheme. However, our Spy is not as powerful as in the original Roscoe and Goldsmith version, which is in complete control of the whole network. Obviously, that would be pointless in the current case, as it would clearly violate robustness.

Here, the *Spy* plays the role of a mix server that can receive ingoing messages over a *learn* channel, *infer* events based on received messages and its initial knowledge and then *say* messages that it has inferred. This intruder model provides active attacks against the system, by blocking outgoing messages and sending those that deviate from the protocol. The Spy is constructed using the same approach as taken by Roscoe and Goldsmith, with respect to the messages of the Ximix system.

The initial knowledge of the intruder consists of all the mix servers' identities and public-keys, the initial unmixed vector of values and, of course, its own secret key as well as the assigned share of the secret key.

Apart from the channels *learn*, *say* and *infer*, that allow messages to be received, sent and internally inferred, the *Spy* has in its alphabet all the events that an honest mix server can perform, so it can communicate with the other components. Furthermore, the *Spy* can ignore the instructions sent by the $\mathcal{CS}$ by *absorbing* them and carrying on its operation.

In this model of Ximix, the intruder *learns* messages sent to him only from the previous mix server and *says* to the next one in the shuffle plan. In the case where he is the last mix server in the plan, an additional *say* event is renamed to *postDataToVB* and can post any message that has been learnt. In the same fashion, a *say* event is renamed to *postPartDec*, which means that he can post partial decryptions of messages that he has received and, of course, these are of length larger than or equal to the threshold. In the next section, we shall show that the intruder *hears* all the messages sent to any of the honest mix servers and

he can choose to send different messages to different mix servers, so each message can be potentially be sent to an individual mix server, that is, to the singleton $\{x\}$, where x is the identity of some honest mix server. The new $SYSTEM_{Spy}$ process is now defined in terms of the parallel composition of the $SYSTEM_{Ximix}$ and the *renSpy* processes, as:

$$SYSTEM_{Spy} = SYSTEM_{Ximix} \; {}_{\alpha SYSTEM_{Ximix}}\|_{\alpha renSpy} \; renSpy$$

where, $\alpha SYSTEM_{Ximix}$ and $\alpha renSpy$, are the alphabets containing all the events these processes can engage in. Using the assertion presented in Section 5, we check that $SYSTEM_{Spy}$ does not satisfy its liveness property and in Section 7 we present some traces illustrating this behaviour.

The *Spy* can also mount a Denial of Service (*DoS*) attack, by perpetually posting messages to the $\mathcal{VB}$, thus leading the system to an unstable condition (divergence, in CSP terms) and one defence against this is to add an extra constraint that allows only one post (per event) for each mix server. This is modelled as:

$$DIVERGE = VB(primaryServer, empty, empty, empty) \; ||| \; (post.x?\_ \rightarrow STOP \mid x \in \mathcal{D})$$

where *post* is an abbreviation for the *postDataToVB* and *postPartDec* events the intruder can perform and "_" allows any message of the appropriate type.

A misbehaving $\mathcal{CS}$ can also break robustness. For example, it can refuse to send instructions to a specific mix server, making it wait indefinitely, and causing deadlock. We describe this attack in Section 7. Additionally, upon receiving a threshold number of partially decrypted data, it can either refuse to combine them or output an unrelated message.

So far, we have seen that the proposed Ximix is not resilient in the current implementation, for it is vulnerable to attacks carried out by an intruder and a dishonest $\mathcal{CS}$. FDR confirms that the *RBST* assertion specified in Section 5 does not hold and the complete script for this model can be found at `http://www.tvsproject.org/csp/Ximix_Spy.csp`. In the following section we show how to make the system robust.

Robust Ximix. Here, we describe the changes required to guarantee successful termination in the presence of the intruder introduced above. In our modified Ximix, upon termination, a valid and fully decrypted message is published and any external party interested in verifying its correctness can do so.

One of the purposes of a Mix Net is to *distribute the trust* among the mix servers, so that the whole system does not rely on the integrity of a single component. However, Ximix relies critically on the availability and honesty of the $\mathcal{CS}$. Hence, the first step in making Ximix robust is to remove the $\mathcal{CS}$ and instead, allow the mix servers to broadcast their messages. In this context, an honest mix server sends the same message to all the mix servers, while a dishonest one may send different messages to different mix servers, or refuse to send to some of them. We also allow any external party interested in checking and combining the partial decryptions to do so, in order to check the fully decrypted

messages. A few changes to the previous model are needed to accommodate these decisions, which we describe below and the complete script can be found at `http://www.tvsproject.org/csp/Ximix_robust.csp`.

Honest mix servers. An honest mix server is willing to receive messages from another mix server; he can, however, *timeout* before or after receiving it. Adding this behaviour in the system, we allow the execution to continue even when a dishonest mix server refuses to send messages to some of the honest ones (simply times out without performing any action). In this case, the honest mix servers will absorb the timeout and continue to operate. At the end of the mixing phase, the honest mix servers will have posted on their $\mathcal{TB}$s at least one message of threshold length, mixed by the majority of the mix servers. The requisite changes in the *HON_SRVR* and *renHON_SRVR* processes are shown below. Additionally, an extra *timeout* is added after the *send* event in the *MIX* process presented in Section 5.

$$
\begin{array}{l}
HON\_SRVR(id) = \\
\quad \textbf{if } id == primaryServer \textbf{ then} \\
\qquad \Box_{\substack{m \in \{m' \in \mathcal{M} \mid \#m' = 0\}, \\ commit \in \mathcal{C}}} \left(\begin{array}{l} send.\mathsf{S}_{id}(sk_{id}, m).\mathsf{S}_{id}(sk_{id}, h(commit)) \rightarrow \\ timeout \rightarrow \\ MIX(id, 1, m) \end{array} \right) \\
\textbf{else} \\
\qquad timeout \rightarrow MIX(id, val(id), m) \\
\qquad \Box \quad \Box_{\substack{m \in \{m' \in \mathcal{M} \mid outer(m') = prev(id)\}, \\ commit \in \mathcal{C}}} \left(\begin{array}{l} recv.m.commit \rightarrow \\ postInData!id!m!commit \rightarrow \\ timeout \rightarrow \\ MIX(id, val(id), m) \end{array} \right)
\end{array}
$$

The (non-replicated) external choice in the *HON_SRVR* process models the ability of an honest mix server to timeout before receiving a message. This will only occur where a dishonest mix server refuses to send anything. Minor changes are required when renaming the *HON_SRVR* process:

$$
\begin{array}{l}
renHON\_SRVR(id) = HON\_SRVR(id)[\![{}^{comm.id.\mathcal{P}\setminus\{id\}.m.c}/_{send.m.c} \,|\, m \in \mathcal{M}, c \in \mathcal{C}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![{}^{comm.\mathcal{P}\setminus\{id\}.z.m.c}/_{recv.m.c} \,|\, m \in \mathcal{M}, c \in \mathcal{C}, z \in \mathcal{P}]\!]
\end{array}
$$

Here, each *send* is renamed to a $comm.id.\mathcal{P}\setminus\{id\}$, broadcasting a message to all mix servers other than *id*. Each *recv* is renamed to an incoming $comm.\mathcal{P}\setminus\{id\}.z$, where z can be either a singleton containing only *id* or a set of some mix servers.

Dishonest mix servers. A dishonest mix server learns all messages sent to any of the honest mix servers, so *learn* events are renamed to incoming $comm.x.\mathcal{P}\setminus\{x\}$, where x is an honest mix server. The renaming of *say* events is slightly more complicated in that we allow the *Spy* to send messages only to individual servers (singleton sets). This simplifies the modelling, but without loss of generality: if he wants to send different messages to different mix servers then he can now do so, and if he wishes to send the same message to all mix servers, he can send it to each of them separately.

$$
\begin{array}{l}
renSpy = Spy[\![{}^{comm.x.\mathcal{P}\setminus\{x\}.m.c}/_{learn.m.c} \,|\, m \in \mathcal{M}, x \in \mathcal{H}, c \in \mathcal{C}]\!] \\
\qquad\qquad [\![{}^{comm.y.\{x\}.m.c}/_{say.m.c} \,|\, m \in \mathcal{M}, x \in \mathcal{H}, y \in \mathcal{D}, c \in \mathcal{C}]\!] \\
\qquad\qquad [\![{}^{postDataToVB.y.m.c}/_{say.m.c} \,|\, m \in \mathcal{M}, y \in \mathcal{D}, c \in \mathcal{C}]\!] \\
\qquad\qquad [\![{}^{postPartDec.y.m}/_{say.m.c} \,|\, m \in \mathcal{M}, y \in \mathcal{D}, c \in \mathcal{C}, \#m >= threhold]\!]
\end{array}
$$

Putting the Network Together. Our final $SYSTEM_{XimixNoCS}$ process is now defined in terms of the parallel composition of the new *renSpy* and the processes defined in the earlier sections, as:

$$SYSTEM_{XimixNoCS} = MIXING \,||\, CHECKING \,||\, DECRYPT \,||\, COMBINE \,||\, DIVERGE \,||\, renSpy$$

Eventually, using the *RBST* assertion defined in Section 5, we check that the robustness of the system is maintained and the traces described in Section 7 support this argument.

7 Results and Analysis

In this section, we verify the protocols against the liveness property and we present interesting traces illustrating their behaviour. Each of the traces discussed in this section were obtained via simulating the models in ProBE, a CSP animator (built into FDR3), which allows the user to explore the behaviour of a process. Due to the number of events occurring in the traces, we keep only those highlighting the importance of the trace. For clarity, throughout this section, we give traces using three mix servers; the shuffle plan will always be $\langle AA, BB, CC\rangle$ and B the primary mix server. There are numerous possible corruption scenarios that can be modelled and analysed and we have chosen the most representative ones. We start with the $SYSTEM_{Ximix}$ process, which is a faithful model of Ximix where all the components are honest.

$\langle CS\_instructsToCopy.A,$
$comm.B.\{A\}.\mathsf{S}_B(\mathbf{c}).\mathsf{S}_B(h(commit)),$
$postInData.A.\mathsf{S}_B(\mathbf{c}).\mathsf{S}_B(h(commit)),$
$CS\_instructsToShuffle.A,$
$CS\_instructsToCreateTB.A,$
$CS\_instructsToCopy.B,$
$comm.A.\{B\}.\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})).\mathsf{S}_A(h(commit)),$
$\ldots$
$comm.B.\{C\}.\mathsf{S}_B(\mathsf{M}_B(\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})))).\mathsf{S}_B(h(commit)),$
$\ldots$
$postDataToVB.C.\mathsf{S}_C(\mathsf{M}_C(\mathsf{S}_B(\mathsf{M}_B(\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})))))).\mathsf{S}_C(h(commit)),$
$mixing\_done, checking\_done,$
$readFromVBData.A.\mathsf{S}_C(\mathsf{M}_C(\mathsf{S}_B(\mathsf{M}_B(\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})))))),$
$postPartDec.A.\mathsf{P}_A(pt_A, \mathsf{M}_C(\mathsf{M}_B(\mathsf{M}_A(\mathbf{c})))),$
$\ldots$
$decryption\_done,$
$CSreadFromVBPartDec.\{\mathsf{P}_A(pt_A, \mathsf{M}_C(\mathsf{M}_B(\mathsf{M}_A(\mathbf{c})))), \ldots\},$
$publish.\mathsf{M}_C(\mathsf{M}_B(\mathsf{M}_A(\mathbf{p})))\ \rangle$

In general, after the *decryption_done* event, the $\mathcal{CS}$ combines all the partial decryptions of length greater than or equal to the threshold with preference given to longer messages. Hence, $\mathsf{M}_C(\mathsf{M}_B(\mathsf{M}_A(\mathbf{p})))$ will be considered the published output as it is of length greater than the threshold. Of course, more interesting traces result from checking the protocol under the existence of our Spy.

In $SYSTEM_{Spy}$, consider the case where the intruder is the primary mix server, B; he starts by sending the unmixed data to A. A is honest, mixes twice the original data, produces its own mixed messages and communicates them to B. Now,

B decides to act dishonestly and instead of sending to C that received by A, he sends the unmixed data. At this point, C, being unable to distinguish which of the preceding mix servers misbehaved, accepts what B sent and operates on them. As C is the last in the shuffle plan, he posts a mixed message of length 1 (mixed only by C) to the $\mathcal{VB}$. Each server is now instructed by the $\mathcal{CS}$ to read the posted message and partially decrypt it. However, none of them will post any partial decryption of the posted message because it is of length being less than the threshold. Subsequently, the $\mathcal{CS}$ cannot read or combine any partial decryptions and the whole process stops without publishing anything. Clearly, this violates the robustness requirement. This is demonstrated in the left trace below.

$\langle \ldots$
$comm.B.\{C\}.\mathsf{S}_B(\mathbf{c}).\mathsf{S}_B(h(commit))$,
$\ldots$
$postDataToVB.C.\mathsf{S}_C(\mathsf{M}_C(\mathbf{c})).\mathsf{S}_C(h(commit))$,
$readFromVBData.A.\mathsf{S}_C(\mathsf{M}_C(\mathbf{c}))$,
$postPartDec.A.nothing$,
$\ldots$
$CSreadFromVBPartDec.empty$,
$STOP\rangle$

$\langle \ldots$
$comm.A.\{B\}.\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})).\mathsf{S}_A(h(commit))$,
$\ldots$
$CS\_instructsToCopy.B$,
$\ldots$
$CS\_instructsToCopy.C$,
$\times\rangle$

A similar behaviour arises when C is the spy. Upon receiving a message from B, he ignores it, mixes the initial data and posts them on the $\mathcal{VB}$. No one can now partially decrypt the posted message and the $\mathcal{CS}$ cannot publish anything. In another scenario, (trace above on the right), the intruder, B, sends the unmixed data to A and absorbs the instructions sent by the $\mathcal{CS}$, but refuses to transmit the received messages to C, thus resulting in a deadlock. This is because C is willing to receive a message from B, but it never arrives.

However, as we have seen in Section 6, the $\mathcal{CS}$ can also break robustness. The empty trace $\langle\times\rangle$ illustrates the scenario in which the $\mathcal{CS}$ does not send a *copy* instruction to A. Although the mix server is always willing to receive and execute it, such an instruction never arrives and the $SYSTEM_{Spy}$ deadlocks without performing even one event.

In our revised system, in the trace below, the intruder, acting as dishonest mix server A, times out without sending any message. However, this does not prevent the other mix servers from continuing to operate.

$\langle comm.B.\{A, C\}.\mathsf{S}_B(\mathbf{c}).\mathsf{S}_B(h(commit)), timeout,$
$timeout,$
$comm.B.\{A, C\}.\mathsf{S}_B(\mathsf{M}_B(\mathbf{c})).\mathsf{S}_B(h(commit)), timeout,$
$\ldots$
$comm.C.\{A, B\}.\mathsf{S}_C(\mathsf{M}_C(\mathsf{S}_B(\mathsf{M}_B(\mathbf{c})))).\mathsf{S}_C(h(commit)), timeout$
$\ldots$
$timeout,$
$postPartDec.B.\mathsf{P}_B(pt_B, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c}))),$
$postPartDec.C.\mathsf{P}_C(pt_C, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c}))),$
$readFromVBPartDec.\{\mathsf{P}_C(pt_B, \mathsf{M}_B(\mathsf{M}_B(\mathbf{c}))), \mathsf{P}_C(pt_C, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c})))\},$
$publish.\mathsf{M}_C(\mathsf{M}_B(\mathbf{p}))\rangle$

Having received nothing from A, mix server B mixes the original data (which are visible to everyone on the $\mathcal{VB}$) and broadcasts them to $\{A, C\}$. Honest C operates similar to B and posts its own mixed data onto the $\mathcal{VB}$. All the mix servers proceed now to the partial decryption of the messages that they have received

during the mixing phase and post them onto the $\mathcal{VB}$. The intruder can choose either to post a valid partial decryption of a message of correct length or not to post anything. Neither of these can violate the robustness of the system: enough partial decryptions of the same message (posted by the honest mix servers) appear on the $\mathcal{VB}$. Any external party can now combine them and publish the fully decrypted message.

A more interesting behaviour occurs when the intruder sends different messages to different mix servers, or does not send to some of them, in order to cause a dispute among them. More specifically, the intruder acting as dishonest mix server B, refuses to communicate the initial data to A and times out. Honest A reads the initial data from the $\mathcal{VB}$, mixes and broadcasts them to $\{B,C\}$. Now, dishonest B chooses to send different mixed messages to A and C: he sends a mix of the received messages to A and a mix of the initial data to C. At this point, C cannot work out which of A or B has misbehaved. When receiving $\mathsf{S}_B(\mathsf{M}_B(\mathbf{c}))$ from B, he might think that A did not mix the initial output and simply forwarded them to B (or timed out). On the other hand, he might think that B ignored message from A and that B mixed the initial data and transmitted to him. This is shown in the following trace.

$\langle timeout,$
$\ldots$
$comm.A.\{B, C\}.\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})).\mathsf{S}_A(h(commit)), timeout,$
$\ldots$
$comm.B.\{A\}.\mathsf{S}_B(\mathsf{M}_B(\mathsf{S}_A(\mathsf{M}_A(\mathbf{c})))).\mathsf{S}_B(h(commit)),$
$comm.B.\{C\}.\mathsf{S}_B(\mathsf{M}_B(\mathbf{c})).\mathsf{S}_B(h(commit)), timeout,$
$\ldots$
$comm.C.\{A, B\}.\mathsf{S}_C(\mathsf{M}_C(\mathsf{S}_B(\mathsf{M}_B(\mathbf{c})))).\mathsf{S}_C(h(commit)), timeout,$
$\ldots$
$postPartDec.A.\mathsf{P}_A(pt_A, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c}))),$
$postPartDec.B.\mathsf{P}_B(pt_B, \mathsf{M}_B(\mathsf{M}_A(\mathbf{c}))),$
$postPartDec.C.\mathsf{P}_C(pt_C, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c}))),$
$readFromVBPartDec.\{\mathsf{P}_A(pt_A, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c}))), \mathsf{P}_B(pt_B, \mathsf{M}_B(\mathsf{M}_A(\mathbf{c}))), \mathsf{P}_C(pt_C, \mathsf{M}_C(\mathsf{M}_B(\mathbf{c})))\},$
$publish.\mathsf{M}_C(\mathsf{M}_B(\mathbf{p}))\rangle$

In the partial decryption phase, dishonest B is able to post two different partially decrypted messages, both of threshold length: $\mathsf{P}_B(pt_B, \mathsf{M}_B(\mathsf{M}_A(\mathbf{c})))$ and $\mathsf{P}_B(pt_B, \mathsf{M}_C(\mathsf{M}_A(\mathbf{c})))$. A has seen two different messages, both of threshold length; he partially decrypts the latest to arrive. C has enough information to partially decrypt only one message and finally, the output is $\mathsf{M}_C(\mathsf{M}_B(\mathbf{p}))$.

All these traces describe some of the possible behaviours of the system, and might not have been appreciated without this analysis. We saw that it is easy for the intruder to break the robustness of the original protocol by choosing not to perform some specific actions. To recognise our contributions in making the system robust, the above traces show that whatever actions an intruder is willing to perform, the robustness of the protocol remains intact.

Moreover, in order to make sure that our changes for making Ximix robust are sound, we further analysed the system in the presence of a stronger intruder, who controls more than the threshold number of mix servers (that is, two). As expected, and verified in FDR, the modified Ximix is not robust in this setting, and FDR provides a trace where two dishonest servers (out of three) can prevent the Mix Net from completing.

The automated verification of the models, with three mix servers, completes in a matter of minutes in a modern laptop. However, when adding an extra mix server, the state space escalates quickly. Table 1 in Appendix B shows this problem.

8 Previous Work

In this context, the first formal model of a re-encryption Mix Net was provided by Stathakidis *et. al* [13]. In their work, they verify the robustness and privacy properties of a typical $\mathcal{WBB}$-based re-encryption Mix Net in the presence of a realistic intruder, using the FDR model-checker and show the modifications that are needed in order to make such a Mix Net robust. The Mix Net verified in their work was inspired by Verificatum [10], and it has few similarities with Ximix. For example, the partial decryption phase is absent from [13] and the mix servers check the received messages before proceeding to their mixing. As we saw in Section 5, in Ximix, the mix servers are willing to accept any *structurally correct* message and proceed to their mixing without checking their validity. Obviously, this may lead to an incorrect output. Furthermore, a different notion of a $\mathcal{WBB}$ is presented in [13], where it acts as a broadcast channel and keeps a consistent record of all messages being sent between the mix servers. In Ximix (beta version), only the messages produced by the last mix server are posted on the $\mathcal{WBB}$ and each of them maintains a local board, called a *Transient Board* ($\mathcal{TB}$). Küsters *et. al* [15], provide a formal security analysis of Chaumian RPC Mix Nets. They propose a new security definition, called *accountability*, which allows one to measure the level of privacy and verifiability of such a Mix Net precisely. Their analysis is interesting, but it is not automated.

9 Conclusion

We have described and conducted a formal modelling and verification of a beta version of the Ximix Mix Net. The production version, which addresses the key issues raised in this paper, will be used in real large-scale elections in Victoria, Australia, in November 2014. It was our aim to be explicit about all the subtleties of the protocol and to apply sufficient rigour in ensuring its robustness. We demonstrated that the beta version is not robust in the presence of an intruder, based on Roscoe and Goldsmith's perfect Spy, and described the modifications needed in order to satisfy this liveness requirement. In our revised Ximix, the election does not rely on the integrity of a single component, but instead distributes the trust among them. Our analysis demonstrates that Ximix guarantees completion and produces a valid output in the presence of a dishonest component. Additionally, we explained the impact on the lack of standardisation in Mix Nets and in what extent they can be standardised. Although this standardisation is difficult to be achieved in practice, it would be useful and solve issues occurring when Mix Nets are used in real world applications, such as in constructing trustworthy electronic voting systems.

Acknowledgements. This work was supported by the EPSRC project Trustworthy Voting Systems, project EP/G025797/1. We would like to thank Dave Hook for responding to our questions on Ximix. Thanks also to Chris Culnane and Chris Mitchell and the anonymous reviewers for their pertinent comments.

A CSP

Communicating Sequential Processes (CSP) is a process algebra designed for describing processes that interact with each other. It was introduced by Hoare [16] in 1978 and since then it has been extensively used for applying the theory of concurrency in practice. The core of the CSP algebra is a *process*, which is described by the way it communicates with its environment. Processes proceed from one state to another by engaging in *events*. An event describes a particular action that can be performed or refused by a process and the set of all possible events is denoted by Σ. In CSP, all the communication events are instantaneous and they happen only when both the processes and the environment agree on their occurrence (*handshaken* communication). At the construction of a process in CSP, the alphabet of a process P, denoted αP, is the set of all visible communication events that this process may perform. For a detailed explanation of CSP and its associated FDR model-checker, we refer the reader to [17–19].

$STOP$ is the simplest CSP process, which does nothing and $SKIP$ is the process indicating successful termination. The process $a \rightarrow P$ is initially willing to communicate a and then behaves like P. $P \,\Box\, Q$ can act either as P or Q, the choice of which is in the hands of the environment. Replicated external choice replicates the choice over the set $\mathcal{A}$, and is denoted by $\Box_{x \in \mathcal{A}} P(x)$. By $P \underset{\mathcal{A}}{\parallel} Q$ we denote generalised parallel, which synchronises P and Q on events lying in the set $\mathcal{A}$. Alphabetised parallel is denoted by $P \,{}_{\alpha P}\|_{\alpha Q}\, Q$ and synchronises P and Q on events lying in the intersection of αP and αQ. We write $\|_{i \in I}\, [\alpha P]\, P(i)$ for the replicated alphabetised parallel composition of processes $P(i)$ indexed over I, where each $P(i)$ is allowed to perform events from αP and the processes are synchronised on the common events. In hiding, $P \setminus \mathcal{A}$, the internal events from $\mathcal{A}$ are hidden from the environment. In renaming, $[\![{}^{a}/_{b}]\!]$, the events b occurring in the process are replaced by the events a. $P \,|||\, Q$ is the interleaving process, where the processes P and Q run independently of each other without synchronising on any event.

B State-space Explosion Problem

Individual checks were performed for each possible instantiation of the models, with a minority of mix servers, testing the inevitability of an output being produced. For the case where three mix servers where used, six models were checked (three for $SYSTEM_{\text{Spy}}$ and three for $SYSTEM_{\text{XimixNoCS}}$). Similarly, in the case where four mix servers were participating in the process, eight checks, in total, were performed. All checks verified the inevitability of a *publish* event, regardless

of the behaviour of the dishonest mix servers. All checks were performed using the refinement checker FDR3 beta 7 on a desktop with an Intel i7 Quad-Core CPU @ 3.6GHz with 8GiB of memory running 64-bit Ubuntu 12.04.

Looking at $SYSTEM_{\text{XimixNoCS}}$, one can see how the state-space grows when the intruder is the last in the shuffle plan. In this case, he *learns*, *infers* and is able to *deduce* and finally *say* many more messages due to the fact that he receives many messages broadcast from all the previous honest mix servers.

System / $\mathbb{P}\mathcal{H}, \mathbb{P}\mathcal{D}$	$SYSTEM_{\text{Ximix}}$		$SYSTEM_{\text{Spy}}$		$SYSTEM_{\text{XimixNoCS}}$	
	RBST	States	RBST	States	RBST	States
{A,B,C}	✓	45	-	-	-	-
{A,B,C,D}	✓	56	-	-	-	-
{B,C}, {A}	-	-	×	5680	✓	289
{A,C}, {B}	-	-	×	13544	✓	2679
{A,B}, {C}	-	-	×	11882	✓	2060
{A}, {B,C}	-	-	×	57869	×	1695
{B,C,D}, {A}	-	-	NA	NA	NA	NA
{A,C,D}, {B}	-	-	NA	NA	NA	NA
{A,B,D}, {C}	-	-	NA	NA	NA	NA
{A,B,C}, {D}	-	-	NA	NA	NA	NA
{A,B}, {C, D}	-	-	NA	NA	NA	NA

Table 1: The FDR verification results for our models of Ximix. As the state-space increases quickly with the number of mix servers, it was not possible for FDR to handle such huge states. These are denoted as "NA" in the table.

However, FDR cannot handle the state-space produced when four mix servers are taking part in the process. An honest mix server is required to post onto (resp. read from) its internal $\mathcal{TB}$ all the received messages, as well as to post (resp. read) the intermediate and ouputted messages. In a similar fashion, the mix servers post their partially decrypted messages onto the $\mathcal{VB}$. Internally, the *Spy*, does not need to maintain a $\mathcal{TB}$, as he is able to say any received (or mixed by him) message. All these communications between the mix servers and the boards are computationally expensive and increase the overall state-space.

In the way the models are constructed, *COMBINE* is the most demanding process. That is, the combiner (the $\mathcal{CS}$ in $SYSTEM_{\text{Spy}}$ and any interested party in $SYSTEM_{\text{XimixNoCS}}$), is responsible for reading all the partial decryptions from the $\mathcal{VB}$, checking the associated generated zero-knowledge proofs, combining them into a fully decrypted plaintext message and publishing it. Hence, it is willing to read *all* the possible sets of partially decrypted messages and the correct way for implementing it in CSP involves the use of powersets. In the case with four mix servers, each of them is able to partially decrypt any received message of length greater than or equal to the threshold (here, three). Using the assumption that messages are in strictly increasing order, i.e. a mix server

can mix only a message signed by a preceding mix server, the cardinality of the set containing all the possible partially decrypted messages is 20. Under this circumstance, the combiner is willing to read any possible combination of these 20 messages, that is, $\mathbb{P}(20) = 2^{20} \simeq 1M$, but FDR struggles when performing such calculations.

References

1. Roscoe, A.W., Goldsmith, M.: The perfect spy for model-checking crypto-protocols. In: Proceedings of DIMACS workshop on the design and formal verification of crypto-protocols, Rutgers University, (September 1997)
2. Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Commun. ACM **24**(2) (1981) 84–88
3. Golle, P., Jakobsson, M., Juels, A., Syverson, P.F.: Universal re-encryption for mixnets. In Okamoto, T., ed.: CT-RSA. Volume 2964 of Lecture Notes in Computer Science., Springer (2004) 163–178
4. Park, C., Itoh, K., Kurosawa, K.: Efficient anonymous channel and all/nothing election scheme. In Helleseth, T., ed.: EUROCRYPT. Volume 765 of Lecture Notes in Computer Science., Springer (1993) 248–259
5. Adida, B.: Helios: Web-based open-audit voting. In: Proceedings of the 17th USENIX Security Symposium (Security '08). (2008)
6. Ryan, P.Y.A., Bismark, D., Heather, J., Schneider, S., Xia, Z.: Prêt à voter: a voter-verifiable voting system. IEEE Transactions on Information Forensics and Security **4**(4) (2009) 662–673
7. Burton, C., Culnane, C., Heather, J., Peacock, T., Ryan, P.Y.A., Schneider, S., Srinivasan, S., Teague, V., Wen, R., Xia, Z.: A supervised verifiable voting protocol for the victorian electoral commission. In Kripp, M.J., Volkamer, M., Grimm, R., eds.: Electronic Voting. Volume 205 of LNI., GI (2012) 81–94
8. Burton, C., Culnane, C., Heather, J., Peacock, T., Ryan, P.Y.A., Schneider, S., Srinivasan, S., Teague, V., Wen, R., Xia, Z.: Using prêt à voter in victorian state elections. In: Proceedings of the 2012 International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections. EVT/WOTE'12, Berkeley, CA, USA, USENIX Association (2012) 1–1
9. Jakobsson, M., Juels, A., Rivest, R.L.: Making mix nets robust for electronic voting by randomized partial checking. In Boneh, D., ed.: USENIX Security Symposium, USENIX (2002) 339–353
10. Wikström, D.: Verificatum. Website (2014) `http://www.verificatum.org/verificatum/index.htm`.
11. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory **31**(4) (1985) 469–472
12. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Advances in Cryptology, CRYPTO '92, Springer-Verlag (1992) 89–105
13. Stathakidis, E., Williams, D.M., Heather, J.: Verifying a mix net in csp. In: Proceedings of the 13th International Workshop on Automated Verification of Critical Systems (AVoCS 2013). Volume 66 of Electronic Communications of the EASST., European Association of Software Science and Technology (2013)
14. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall (1998)
15. Küsters, R., Truderung, T., Vogt, A.: Formal analysis of chaumian mix nets with randomized partial checking. IACR Cryptology ePrint Archive **2014** (2014) 341

16. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8) (1978) 666–677
17. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. 1st edn. John Wiley & Sons, Inc., New York, NY, USA (1999)
18. Roscoe, A.: Understanding Concurrent Systems. 1st edn. Springer-Verlag New York, Inc., New York, NY, USA (2010)
19. Gardiner, P., Goldsmith, M., Hulance, J., Jackson, D., Roscoe, B., Scattergood, B., Armstrong, P.: Fdr2 user manual (2010) `http://www.fsel.com/fdr2_manual.html`.