Eindhoven University of Technology

MASTER

Probabilistic analysis of network anonymity using PRISM

Adithia, M.T.

Award date:
2008

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
MASTER’S THESIS

Probabilistic Analysis
of Network Anonymity using PRISM

by

M.T. Adithia

Supervisor

dr. E.P. de Vink

Eindhoven, August 2006
ACKNOWLEDGEMENTS

Time goes by and now I am in the last phase of my study in Holland. Two years ago when I arrived here at the first time, I did not have any idea of what I would face. Being really far away from my family and friends really scared me.

The same thing happened six months ago when I started this master's thesis. I did not know what to do exactly and whether I could make progress as expected or not. I also was not confident that I could finish it since my background is Mathematics and many Computer Science methods and terms were really new for me. But all my doubts and worries are gone now I made it. I have finished my Master's thesis.

Firstly, I would like to thank God for always giving me His blessing. I would not have the courage to move on without it.

Furthermore, I wish to express my sincere gratitude to all the people, who in one way or another helped me in this work.

Erik de Vink, I did not expect to get a really nice supervisor as you. Thank you for your support, time for discussions almost every week, patience to listen to my difficulties during the process, and encouragement for me to learn new things. I really appreciate it.

Natalia Sidorova and Simona Orzan, thank you for being the committee members of my final examination.

Henk van Tilborg, thank you for your support during my study here in Holland. You are not only my lecturer but also my friend that always believe that I can make it.

Ichsan, really thank you to be a great fiancé who always give me your time, heart, love and patience. Thank you for being there for me to listen all those complaints and cries. I would not make it without you.

All my family back there in Indonesia, I would never be here and finish my study without all of your support and love. I miss you all.

All my friends, thank you guys for cheering me up everyday with all those jokes and funny stories! Those helped me a lot to forget all nightmares.

Mariskha Adithia
Eindhoven, 15 August 2006
SUMMARY

This report discusses probabilistic analysis of the anonymity provided by anonymity networks, namely Crowds, Adithia, Onion Routing, and Tarzan, by using a model-checker tool called PRISM. Previously, Shmatikov has analyzed the anonymity properties provided by the Crowds network with PRISM. In this project, we investigate the applicability of the approach used by Shmatikov to other anonymous networks, in particular Onion Routing and Tarzan. An additional network, baptised Adithia, is made to complete the analysis. These networks are the improvements of others. Regarding the encryption method, Onion Routing and Tarzan improve Crowds and Adithia networks, by changing the link encryption employed into layered encryption. Similarly, based on the forwarder selection, Adithia and Tarzan improved upon Crowds and Onion Routing networks by adding the concept of mimics.

As preliminary steps, we first make models of two well-known anonymity problems, namely the Dining Cryptographers and the Three Door problems. These steps are taken to better understand how to model protocols with PRISM and how to analyze them. For the first model, we check the functional correctness of the protocol by computing some probabilistic properties of it. As for the second model, we try to find the best strategy to win the prize by means of probabilistic analysis.

The next step, we remodel Crowds based on Shmatikov’s model to make it easier to understand. This model is also more organised and describes the protocol more precisely. Furthermore, we extend the approach to model Adithia, Onion Routing, and Tarzan networks. The models cover the member’s behavior only on the path setup protocol, ignoring all subsequent communication conducted along an established static path. Moreover, since the models do not contain any non-deterministic choice, they can be built as a discrete-time Markov chain. As for the adversaries, we model them as a coalition of corrupt members which is able to observe a communication within a path. Note that the observations can be done only if the adversary is selected to join the path. An important assumption we use for the models is that the corrupt members are able to link paths.

It is proven in the originating paper of Reiter and Rubin that in the Crowds network, the initiator of a message enjoys a single path anonymity. Following Shmatikov, we would like to investigate what is the level of anonymity provided by the Crowds network if multiple paths are built to communicate with the same server. Furthermore, we extend the same approach to analyze Adithia, Onion Routing, and Tarzan, in order to compare the anonymity properties provided by the networks if the composition of the number of the honest and corrupt members is the same for all of them. To do so, some desired anonymity properties are computed by PRISM and after that the results of the computation are discussed.

Based on the results, the anonymity provided by Crowds, Adithia, Onion Routing, and Tarzan is compared. Before they are compared, we first translate the problem into a square which is built based on two categories that determine the significant differences between the networks, namely the encryption method which is employed and the choice of forwarder selection.
# CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>v</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. Network anonymity</td>
<td>5</td>
</tr>
<tr>
<td>3. Learning PRISM</td>
<td>7</td>
</tr>
<tr>
<td>4. The Dining Cryptographers and Three Door Problems</td>
<td>9</td>
</tr>
<tr>
<td>4.1 The Dining Cryptographers problem</td>
<td>9</td>
</tr>
<tr>
<td>4.1.1 Overview of the problem</td>
<td>9</td>
</tr>
<tr>
<td>4.1.2 The model</td>
<td>10</td>
</tr>
<tr>
<td>4.1.3 Results and analysis</td>
<td>11</td>
</tr>
<tr>
<td>4.2 The Three Door problem</td>
<td>12</td>
</tr>
<tr>
<td>4.2.1 Overview of the problem</td>
<td>12</td>
</tr>
<tr>
<td>4.2.2 The model</td>
<td>12</td>
</tr>
<tr>
<td>4.2.3 Results and analysis</td>
<td>13</td>
</tr>
<tr>
<td>5. General picture of the model</td>
<td>15</td>
</tr>
<tr>
<td>5.1 Overview of the networks</td>
<td>15</td>
</tr>
<tr>
<td>5.2 Adversary</td>
<td>16</td>
</tr>
<tr>
<td>5.3 The model</td>
<td>16</td>
</tr>
<tr>
<td>5.3.1 Adversary model</td>
<td>16</td>
</tr>
<tr>
<td>5.3.2 Path reformulation</td>
<td>17</td>
</tr>
<tr>
<td>5.3.3 Mimics construction</td>
<td>18</td>
</tr>
<tr>
<td>5.4 Formalization of the anonymity properties</td>
<td>18</td>
</tr>
<tr>
<td>5.5 Computation method</td>
<td>19</td>
</tr>
</tbody>
</table>
## Contents

6. The model

6.1 Crowds

6.1.1 Overview of the network

6.1.2 The model

6.1.3 Results and analysis

6.2 Adithia

6.2.1 Overview of the network

6.2.2 The model

6.2.3 Results and analysis

6.3 Onion Routing

6.3.1 Overview of the network

6.3.2 The model

6.3.3 Results and analysis

6.4 Tarzan

6.4.1 Overview of the network

6.4.2 The model

6.4.3 Results and analysis

6.5 Conclusions of the results

7. Discussion

7.1 The Anonymity Square problem

7.2 Comparison between Crowds, Adithia, Onion Routing, and Tarzan networks

7.2.1 Crowds → Onion Routing → Tarzan

7.2.2 Crowds → Adithia → Tarzan

7.3 Conclusions of the discussion

8. Concluding remarks

Bibliography

Appendix

A. The Dining Cryptographers Problem’s code

B. The Three Door Problem’s code
C. Crowds’ code ................................................................. 61
D. Adithia’s code ............................................................... 63
E. Onion Routing’s code ....................................................... 67
F. Tarzan’s code ............................................................... 71
1. INTRODUCTION

People require privacy when doing transaction via Internet connection. For example, some people do not want other people to know what web page they request. Anonymous cash is not anonymous anymore if the channel used for the connection identifies the identity of the participating entities. Furthermore, email users sometimes want to hide their email addresses. These days, the Internet does not really care about privacy. Encryption is provided to protect privacy, but it only protects the content of a transaction. Thus an eavesdropper can still learn the IP addresses of the internet users to infer their identities.

Based on the motivation mentioned above, people create special networks to protect privacy, especially the identities of the entities participating in a communication via Internet connections. In other words, the main goal of the networks is to provide anonymity for their users. Each network employs some specific methods to reach the goal.

An example of the special networks is the Crowds network proposed by Reiter and Rubin [18] which has the main idea “blending into a crowd”, i.e. hiding one’s actions within the actions of many others. In general, a crowd is a collection of users. If a user wants to send a request via this network, the request is sent to several other users before it is sent to the end server.

Chaum proposed the Mix-net system [3] that employs layered encryption to protect privacy. Some practical implementations of a Mix-net system are available such as the Reliable [10] and Mixmaster [4] systems. As an improvement of the Crowds network, the Onion Routing network [17, 15] was built by Syverson c.s. which not only forwards a user’s request to some other users first before it is sent to the server, but applies layered encryption to the request as well. After that, Syverson created Tor [6] which improves the key generation employed in the Onion Routing network. The last one is Tarzan [7] which applies multi-hop routing as in Crowds, Onion Routing, and Tor networks, and also layered encryption to a request. In addition, it introduces the concept of a traffic mimic.

Shmatikov has analyzed the anonymity properties provided by Crowds [19] by using a model-checker tool called PRISM [12] developed by Kwiatkowska et al. In his model, Shmatikov assumes that there are some corrupt members in the network who try to infer the identity of the true sender of a request. The detailed explanation about this model is given in Chapter 6.

In this project, we investigate the applicability of the approach used by Shmatikov to other anonymous networks, namely Onion Routing and Tarzan. Furthermore, we compare the anonymity provided by the three networks. To do so, we translate our problem into a square such that the three corners of the square represent the three networks. The last corner represents the so-called Adithia network, which is a network created by us to complete our analysis and to investigate the effects of the concept of mimics to the anonymity provided by a network. The edges of the square represent the correlation between each corner.

This report is organized as follows: Chapter 2 gives some background in network anonymity. In Chapter 3, an overview of the model-checker tool PRISM is given. After that, Chapter 4 provides the description of the models of the Dining Cryptographers and the Three Door problems. We create these models as preliminary steps to understand how PRISM works. Next, in Chapter
5, we describe the models of Crowds, Adithia, Onion Routing, and Tarzan networks in general. The main approach and the analysis method are also given. Chapter 6 gives our description of the Crowds network’s model. The analysis result of this model is also given. Furthermore, based on the Crowds model, we model Adithia, Onion Routing, and Tarzan networks and analyze them. Chapter 7 describes the Anonymity Square problem more precisely. Furthermore, we continue by comparing the anonymity provided by the networks discussed before and investigate the correlation between them. Lastly, Chapter 8 gives the conclusions of the report, future work, and closing remarks.
2. NETWORK ANONYMITY

As mentioned before in this report, people require anonymity when doing communications via Internet connections. The definition of anonymity according to Pfitzmann and Kohntopp \[13\] is as follows:

**Definition 2.1.** The state of being not identifiable within a set of subjects, the anonymity set.

Based on this definition, a subject does communications anonymously if he cannot be distinguished by the adversary from other subjects. Here, we have that an anonymous network is a network that provides anonymity to its users when doing communications via Internet connections.

Furthermore, there are three types of anonymity:

- Sender anonymity: the identity of the sender of a message is hidden;
- Receiver anonymity: the identity of the receiver of a message is hidden;
- Unlinkability \[13\] of two or more items of interests, e.g. subjects, messages, events, actions, etc. This means that although the sender and receiver of a message can be identified as participating in a communication, they cannot be identified as communicating with each other.

If a system is able to provide anonymity to its sender, the next interesting aspect is how to quantify the anonymity provided by the system. More precisely, how to determine the *degree of anonymity* provided by a system. Reiter and Rubin \[18\] define a degree of anonymity as the probability \(1 - p\), where \(p\) is the probability assigned by an attacker to potential senders. More precisely, \[18\] defines the degree of anonymity which is intended for the sender anonymity. However, it can also be extended for receiver anonymity and unlinkability.

- Beyond suspicion which means a sender is beyond suspicion if the attacker can see evidence of a sent message but the sender appears no more likely to be the originator of the message than any other potential senders in the system.
- Probable innocence which means a sender is probably innocent if from the attacker’s point of view, the sender appears no more likely to be the originator than to not be the originator.
- Possible innocence which means a sender is possible innocent if from the attacker’s point of view, there is a nontrivial probability that the real sender is someone else.

Here, the probable innocence property is used to support the deniability of an initiator for sending message. If a system provides a probable innocence property, it does not mean that it hides the sender’s identity from the adversary. Instead, it only says that the probability that an adversary observes the real sender of a message is less than \(\frac{1}{2}\).
Furthermore, there are other definitions of the degree of anonymity. For example, Berthold et al. [1] have their own definition of the degree of anonymity which depends on the number of users in a system. However, in this project, we use the degree anonymity as defined by Reiter and Rubin. We distinguish the properties of an attacker [5] as given below:

- **passive-active**: Passive attackers are able to listen to the communication and/or read internal information of entities participating in the protocols. They typically perform traffic analysis of the communication. While active attackers can add, delay, alter or remove messages and modify internal information of participating entities.

- **internal-external**: An internal attacker controls one or several entities that are part of the system (e.g., the attacker controls some communication nodes). External attackers only control communication links.

- **local-global**: Global attackers have access to the entire communication system (e.g., all communication links), while local attackers only have to some part of the resources.

- **static-adaptive**: Static attackers control a predefined set of resources and are unable to alter their behavior once a transaction is in progress. Adaptive attackers gain control on new resources or modify their behavior, depending on intermediate results of the attack.

Attacks on an anonymity system can be distinguished into two types, namely attacks on anonymity and attacks on the availability of the anonymity service (also called denial of service attacks). Denial of service attacks can be deployed by active attackers. Furthermore, the goal of this attack is to reduce the availability of the system, and moreover to reduce the anonymity provided by them. In this project, we focus on attacking the anonymity.
3. LEARNING PRISM

The model and analysis described in this report are performed by using PRISM. PRISM is a probabilistic model-checker developed by Parker, Norman, and Kwiatkowska at the University of Birmingham. By using this tool, systems that exhibit a probabilistic behavior can be modeled and analyzed. A complete description of PRISM can be found in [12].

The tool mainly requires two inputs, namely the description of the system to be analyzed and a set of properties to be checked related to the system. As for the description, a system can be modeled either as a discrete-time Markov chain (DTMC) which allows probabilistic behaviors, a Markov decision process (MDP) which extends the DTMC by allowing nondeterministic behaviors, or a continuous-time Markov chain (CTMC) which is also an extension of the DTMC with the modeling of continuous time. Note that for this project, we only model systems as a DTMC or an MDP.

The fundamental components of PRISM language are modules and variables. A system can consist of several modules that can interact each other. Each module contains a number of variables which have values that determine the state of the module at any given time. Furthermore, a module can be specified as follows:

\[
\text{module <name> ... endmodule}
\]

In PRISM, there are two types of variables, namely boolean variables and integer variables. The following is an example how to declare a boolean variable,

\[
\text{start : bool init false;}
\]

It declares a boolean variable start, initialised to the value of false. While the following declaration,

\[
\text{length : [0..4] init 0;}
\]

declares an integer variable length to integers ranged from 0 to 4 and has initial value 0.

Furthermore, PRISM language also allows to declare a constant of types integer, double, or boolean. The example of the declaration is

\[
\text{const int Crowdsize = 5;}
\]

which declares a constant Crowdsize equal to an integer 4.

State transition rules in PRISM are specified using guarded commands which have form

\[
[] <guard> \rightarrow <command> ;
\]
where \texttt{guard} is a predicate over system variables and \texttt{command} is a transition executed by the system when \texttt{guard} evaluates to the value of \texttt{true}. The transitions done by the \texttt{command} is an update of the values of variables inside it. If the transition must be chosen probabilistically, the discrete probability distribution is specified as

\[
[] \texttt{guard} \rightarrow \texttt{prob0} : \texttt{command0} +
\quad \ldots +
\quad \texttt{probN} : \texttt{commandN};
\]

It means that \texttt{commandi} is executed with probability \texttt{prob_i}, where \(\sum_{i=1}^{N} \texttt{prob}_i = 1\).

The properties to be checked are stated in \textit{probabilistic computation tree logic} (PCTL) for DTMC and MDP models and \textit{continuous stochastic logic} (CSL) for CTMC models. In PCTL, one may write

\[
P = \? \left[ \texttt{true U deliver \& x=3} \right]
\]
\[
P_{\max} = \? \left[ \texttt{true U next} \right]
\]

The first one is an example of a DTMC property. The expression allows PRISM to compute the probability that eventually \texttt{deliver} is true and \(x = 3\) in the model. Furthermore, the second example is a property for an MDP and it allows PRISM to compute the maximum probability that eventually \texttt{next} is true in the model. Note that for an MDP model, we can only compute maximum or minimum probabilities since the probabilities can only be computed once the nondeterministic choices have been resolved.
4. THE DINING CRYPTOGRAPHERS AND THREE DOOR PROBLEMS

Before we model and analyze some anonymous networks, we model two well-known problems, namely the Dining Cryptographers and the Three door problems as means of introducing a protocol modeling using PRISM. The descriptions of the models and the analyses of those models are given in the next sections.

4.1 The Dining Cryptographers problem

This section is organized as follows: First we give an overview of the problem. Next, we continue by providing a description of the model. Lastly, the analysis to check the functional correctness of the protocol is given.

4.1.1 Overview of the problem

The Dining Cryptographers problem is introduced by Chaum in 1985 [2] and the main idea of this problem is to build an anonymous communication among several parties. Furthermore, the problem can be described by the following situation.

Three cryptographers who work for the National Security Agency (NSA) go to a restaurant to have dinner together. At the restaurant, they take a circle table for three people. The dinner is paid by either the NSA or one of the cryptographers. At the end of the dinner, the NSA informs each of the cryptographers in secret whether or not he is paying.

The cryptographers would like to know whether it is one of them who is paying or whether the NSA is paying. But, if one of them is paying, the identity of him should be remain anonymous. To find out who is paying, each of the cryptographers throws a coin which is made visible to themself and the right-hand neighbor. Next, each cryptographer reads two coins which are visible to him, i.e. on his right-hand and left-hand side. Afterwards, each of the cryptographer should make an announcement whether the coins they read are the same or not. If a cryptographer is not paying then he should say that the coins agree if the results on the coins are the same, and say disagree if the results differ. On the other hand, a paying cryptographer should say the opposite.

If the number of ‘disagree’ announcements is even, then the NSA is paying. Whereas if the number of ‘disagree’ announcements is odd then one of the cryptographers is paying. In the second case, the other two cryptographers who are not paying cannot identify the payer from the information available.
4. The Dining Cryptographers and Three Door Problems

4.1.2 The model

We model the Dining Cryptographers problem above with PRISM. Since the protocol allows nondeterministic choices, we model it as an MDP model.

First we set the identities of the NSA and all cryptographers as constant integers. More precisely, we set constants $\text{Crypt}_i = i$, with $i \in \{0, 1, 2\}$ and $\text{NSA} = 3$ to represent the identity of the first, the second, and the third cryptographer, and the NSA respectively.

As the first step in the protocol, the master chooses either the NSA or one of the cryptographers to pay the dinner. This step is described as nondeterministic choices as follows:

- [] choose -> (choosensa’ = true) & (choose’ = false);
- [] choose -> (choosencrypt’ = true) & (choose’ = false);

Where if the NSA or the $i$th cryptographer is chosen to pay then either one of the following steps is taken. Furthermore, as described before, whoever is chosen to pay, each of the cryptographer throws a coin. This next step is represented by a state variable $\text{throw}_\text{coins}$.

- [] choosensa -> (payer’ = NSA) & (throw_coins’ = true) & (choosensa’ = false);
- [] choosecrypt -> (payer’ = Crypt$_i$) & (throw_coins’ = true) & (choosecrypt’ =false);

For the throwing coins step, we introduce a state variable $\text{tc}_i$ representing the $i$th cryptographer throws a coin. This state variable has integer value ranged from 0 to 2. It is set to 0 in the initial state, and the value is changed to 1 to enable the throwing coins step of each cryptographer. Later on, we change the value to 2 to unable the throwing coins step.

- [] throw_coins -> ( tc0’ = 1 ) & ( tc1’ = 1 ) & ( tc2’ = 1 ) & ( cont_throw’ = true ) & ( throw_coins’ = false );

Note that for each coin$i$ which is thrown by the $i$th cryptographer, there are two possibilities with equal probability, namely HEAD or TAIL appears.

- [] cont_throw & $\text{tc}_i = 1$ -> 0.5: ( coin’$_i$ = HEAD ) & ( $\text{tc}_i’ = 2$ ) & ( done_throw’ = true ) +
  0.5: ( coin’$_i$ = TAIL ) & ( $\text{tc}_i’ = 2$ ) & ( done_throw’ = true );

After the throwing coin step is finished, each each cryptographer checks his coin. Here we introduce a state variable $\text{cc}_i$ which has an initial value equal to 0. We change the value to 1 to enable the $i$th cryptographer to check his coin and later on we change its value to 2 to unable the checking coins step.

- [] done_throw & (tc0 = 2) & (tc1 = 2) & (tc2 = 2) -> (cont_throw’ = false) & (done_throw’ = false) & (check_coins’ = true);
- [] check_coins -> ( cont_check’ = true ) & ( cc0’ = 1 ) & ( cc1’ = 1 ) & ( cc2’ = 1 ) & ( check_coins’ = false );

After the $i$th cryptographer checks two coins which are visible to him. There are four possibilities namely, he is the payer and the coins he checks are the same, he is the payer and the coin he checks are not the same, he is not the payer and the coins he checks are the same and he is not the payer and the coins he checks are not the same. Furthermore, we count the number of disagree announcements with a state variable $\text{countdisagree}$. The following is a part of the checking done by the 0th cryptographer.

- [] cont_check & (cc0 = 1) & (payer = Crypt0) & (coin0 = coin2) ->
  ( countdisagree’ = countdisagree + 1 ) & ( cc0’ = 2 ) & ( done’ = true );
4.1. The Dining Cryptographers problem

\[
\begin{align*}
\text{[] } \text{cont}\_\text{check} \& (cc0 = 1) \& (\text{payer} = \text{Crypt0}) \& (\text{coin0} \neq \text{coin2}) \rightarrow \\& (\text{countdisagree}' = \text{countdisagree}) \& (cc0' = 2) \& (\text{done}' = \text{true});
\end{align*}
\]

Lastly, we finish the protocol as shown below.

\[
\begin{align*}
\text{[] } \text{done} \& (cc0 = 2) \& (cc1 = 2) \& (cc2 = 2) \rightarrow (\text{cont}\_\text{check}' = \text{false}) \& (\text{done}' = \text{false}) \& (\text{finish}' = \text{true});
\end{align*}
\]

The state-transition diagram of the model is shown in Figure 4.1(a).

![State-transition diagram](image)

(a) The Dining Cryptographers  
(b) The Three Door

Fig. 4.1: The state-transition diagram of the Dining Cryptographers and Three Door models

4.1.3 Results and analysis

In this section, we check the functional correctness of the protocol by computing its probabilistic properties. More precisely, we check whether it is true that when the number of disagree
announcements is even then the NSA is the payer, otherwise one of the cryptographers is the payer.

Since the model is an MDP model, we can only compute the maximum or minimum probabilities of the properties we would like to check. We check four properties here: The first two properties are the actual requirements that should be provided by the protocol, namely if the number of the disagree announcements is even then the payer is the NSA, otherwise the payer is one of the cryptographers. We also check the opposite, namely the number of the disagree announcements is even but the payer is one of the cryptographers, otherwise the payer is the NSA. The PRISM code below is an example of the property checked as mentioned before.

\[ \text{Pmax=? } [\text{true } U (\text{countdisagree } = 0 \mid \text{countdisagree } = 2) \& \text{finish } \& (\text{payer }= \text{NSA})] \]

With this code, we compute the maximum probability that eventually the number of disagree announcements is even and the payer is the NSA in the model.

From PRISM computation, we obtain that the first two properties have probabilities 1 and the last two properties have probabilities 0. This proves that the protocol functions correctly.

4.2 The Three Door problem

This section is organized as follows: First we give an overview of the problem. Next, we continue by providing a description of the model. Lastly, the analysis to find out the best strategy to win the prize is given.

4.2.1 Overview of the problem

The Three Door problem \[11, 8\] can be explained as follows. In a quiz-show, there are three closed doors. Behind one of these three doors, there is a prize hidden to win, whereas the other two hide nothing or only a fake prize, referred to as a goat. Next, the quiz master asks the contestant to choose one of the doors. After that, the quiz master opens one of the door which is not chosen to show that the prize is not there. Now, the quiz master asks the contestant whether he wants to stick with his choice or switch to the other door.

4.2.2 The model

With PRISM, we model this problem to determine the best strategy to win the prize, i.e. whether to stick to the choice or switch to a new choice is the best strategy to win. Furthermore, since the protocol has nondeterministic choices, we make the model as an MDP model.

As the first part in the quiz, the quiz master asks the contestant to choose one out of three doors. Thus each door has equal probability to be chosen namely \( \frac{1}{3} \). Furthermore, we introduce an integer variable \text{choosedoor} \ which has value from 0 to 3, representing the door chosen by the contestant. The PRISM code for this part is given below.

\[
[] \text{choose -> 1/3:(choosedoor' = 1) } \& \text{ (choose' = false) } \& \text{ (open' = true) } + \\
1/3:(\text{choosedoor' = 2}) \& \text{ (choose' = false) } \& \text{ (open' = true) } + \\
1/3:(\text{choosedoor' = 3}) \& \text{ (choose' = false) } \& \text{ (open' = true)};
\]
As the next step, the quiz master opens a door which is not chosen and hides a goat. To enable this step, we change the value of boolean variable `open` to true as given above. The PRISM code below describes the step if door 1 is chosen. In addition, we introduce an integer variable `digoat` representing door $i$ hides a goat. This variable has a value of either 0 or 1 and we set its value to 1 to enable the transition.

\[
\begin{align*}
\& \ (\text{choosedoor} = 1) \land \text{open} \rightarrow (\text{opendoor2}' = \text{true}) \land (\text{open}' = \text{false}); \\
\& \ (\text{choosedoor} = 1) \land \text{open} \rightarrow (\text{opendoor3}' = \text{true}) \land (\text{open}' = \text{false});
\end{align*}
\]

\[
\begin{align*}
\& \ \text{id}' = i \land (\text{digoat'} = 1) \land (\text{sos}' = \text{true});
\end{align*}
\]

As mentioned before, the quiz master continues the quiz by asking whether the contestant wants to switch his choice to other door which has not been opened, or stick to his choice. This step is described as follows:

\[
\begin{align*}
\& \ (\text{choosedoor} = 1) \land (\text{id} = 2) \land (\text{sos}) \rightarrow (\text{switchtod3}' = \text{true}) \land (\text{d2ob}' = 1) \land (\text{check}' = \text{true}) \land (\text{sos}' = \text{false}); \\
\& \ (\text{choosedoor} = 1) \land (\text{id} = 2) \land (\text{sos}) \rightarrow (\text{stickd1}' = \text{true}) \land (\text{d2ob}' = 1) \land (\text{check}' = \text{true}) \land (\text{sos}' = \text{false});
\end{align*}
\]

The boolean variable `sos` represents the choice given by the quiz master, namely to switch or stick with the original choice. Whereas the integer variables `switchtod` and `stickd` represent whether the contestant wants to switch or stick with the door $i$ respectively. These variables have values either 0 or 1 and we set their values to 1 to enable the transition.

After the contestant decides his final choice, the quiz master check what is behind the door of the final choice. The following describes the transition if the contestant decides to switch to door 3 and therefore the quiz master checks what is behind door 3.

\[
\begin{align*}
\& \ (\text{switchtod3} = 1) \land (\text{d2ob} = 1) \land (\text{check}) \rightarrow (\text{checkdoor3}' = \text{true}) \land (\text{sw3}' = 1) \land (\text{check}' = \text{false});
\end{align*}
\]

As the result of the checking, there are two possibilities with equal probability, namely the contestant wins the prize hence the door of his original choice hides a goat, or the contestant loses the game which means that his final choice door hides a goat.

\[
\begin{align*}
\& \ \text{checkdoor3} \land (\text{d2ob} = 1) \land (\text{sw3} = 1) \\
\& \ \rightarrow 0.5: (\text{digoat}' = 1) \land (\text{win}' = \text{true}) \land (\text{checkdoor3}' = \text{false}) + \\
\& \ \ 0.5: (\text{d3goat}' = 1) \land (\text{lose}' = \text{true}) \land (\text{checkdoor3}' = \text{false});
\end{align*}
\]

The state-transition diagram of this model is shown in Figure 4.1(b).

### 4.2.3 Results and analysis

In this section, we try to find the best strategy to win the prize by means of probabilistic analysis. More precisely, we compute the probability to win the prize if the contestant decides to switch his choice to other door and if the contestant decides to stick with the door of his original choice, and furthermore compare the results of the computation. The following PRISM code are examples of the properties we want to check.

\[
\begin{align*}
\& \ \text{Pmax=? \ [ \ true \ U \ switchtod1=1 \ & \ \text{win} \ ]} \\
\& \ \text{Pmax=? \ [ \ true \ U \ stickd1=1 \ & \ \text{win} \ ]}
\end{align*}
\]
The first expression allows PRISM to compute the maximum probability that eventually the contestant chooses to switch to door 1 and win the prize. The second one computes the maximum probability that eventually the contestant chooses to stick with his original choice, for example door 1, and win the prize.

For the first property, we obtain that the probability to switch to other door and win the prize is \( \frac{2}{3} \), whereas the probability to stick with the original choice and win the prize is \( \frac{1}{3} \). Thus, the contestant has a greater probability to win the prize if he switches his choice to other door. Therefore we can conclude that the best strategy to win the prize is to switch to other door as also concluded in [8].
5. GENERAL PICTURE OF THE MODEL

This chapter describes the models of the Crowds, Adithia, Onion Routing, and Tarzan networks in general. The main approach and the analysis method are also given.

5.1 Overview of the networks

In general, the Crowds, Adithia, Onion Routing, and Tarzan networks are collections of users or members. When a user wants to execute a web transaction using the system, he should first join a “crowd” of other users. Furthermore in this report, we use the term crowd to represent a collection of users of a network.

The transaction is initiated by a request of a member to a web server. This request is first passed to a random member of the crowd. With this method, when the web server accepts the request, it cannot identify the true initiator of the request since the request is submitted by a random member. Furthermore, the goal of this method is to complicate a traffic analysis by adversaries.

More precisely, when a member in the networks wants to execute a transaction with a web server, he should first setup a path of members. Once the path is setup, any communication from the member to the web server is routed through this path. When the crowd membership changes, the existing paths should be canceled and a new setup protocol should be executed. Note that anonymity guarantees provided by Crowds, Adithia, Onion Routing, and Tarzan are based on the path setup protocol.

The communication between any two of the member in Crowds and Adithia networks is link encrypted using a pairwise symmetric key, thus a communication appears as a plain text to users joined the path. Link encryption is a process of encrypting information at the data link level as it is transmitted between two points within a network. While in Onion Routing and Tarzan networks, the communication is layered encrypted to increase the anonymity provided for their users. With layered encryption, a data is repeatedly encrypted such that after the encryption the data can be considered has layers. As the layered encrypted data moves along the path, each member who receives the data removes one layer of encryption and after that forward the data to the next member on the path. Hence, the members belonging to a path only receive an encrypted communication from which they cannot gain any other information. Moreover, because of the layered encryption, the communication also looks different to each member along the path.

In addition to the path setup protocol and layered encryption, the Tarzan network introduces the concept of a traffic mimic (we apply this concept to Adithia network too). More precisely, a communication in a Tarzan network can only be routed between mimics, or pairs of members assigned by the system in a secure and universally-verifiable manner. Thus, unlike in Crowds and Onion Routing networks, the members to be selected in Tarzan networks are limited. Furthermore, each user can exchange cover traffic with its mimics to increase the anonymity provided by the system.
5.2 Adversary

There are three types of attackers that should be considered in Crowds, Adithia, Onion Routing, and Tarzan, namely:

- A local eavesdropper, an attacker who can observe all and only communication from and to the user’s computer.
- A coalition of corrupt members, other members that can pool their information together and even deviate from the prescribed protocol.
- The end server, the Web server to which the communication is directed.

In this project, we model the adversary as a coalition of corrupt members which is able to observe a communication within a path. Note that the observation can only be done provided the adversary is selected to join the path.

5.3 The model

We model Crowds, Adithia, Onion Routing and Tarzan members’ behavior only on the path setup protocol, ignoring all subsequent communication conducted along an established static path. Furthermore, we assume that once the path is established, when a member receives a message, he cannot gain any additional information about the originator of the message. Since the model does not contain any nondeterministic choices, it is built as a DTMC.

The members in the model are represented by member = i where i ∈ {1, 2, ..., n}, with n is the number of members. Furthermore, we assume that the initiator of the path is member=0 and therefore member=0 always be an honest member.

5.3.1 Adversary model

In the Crowds and Onion Routing models, we divide the members into two sets, i.e. honest or good members and corrupt or bad members sets. A member can be an honest member with probability gmember and can be a corrupt member with probability bmember. Note that bmember = 1 - gmember.

On the other hand, in the Adithia and Tarzan models, the corrupt members have identities which are chosen among all the members with equal probability. This method is employed since the identities of the corrupt members are used in the mimics checking. Furthermore, even the corrupt members have identities in the models, their individual observations are combined into one single observation by the adversary. More precisely, we do the following to choose the corrupt member for crowd size 6:

\[
\begin{align*}
\[] \text{bad_id} = 6 &\to 1/5: (\text{bad_guy} = 1) \& (\text{new} = \text{true}) \& (\text{bad_id} = 0) + \\
&1/5: (\text{bad_guy} = 2) \& (\text{new} = \text{true}) \& (\text{bad_id} = 0) + \\
&1/5: (\text{bad_guy} = 3) \& (\text{new} = \text{true}) \& (\text{bad_id} = 0) + \\
&1/5: (\text{bad_guy} = 4) \& (\text{new} = \text{true}) \& (\text{bad_id} = 0) + \\
&1/5: (\text{bad_guy} = 5) \& (\text{new} = \text{true}) \& (\text{bad_id} = 0);
\end{align*}
\]

The expression bad_guy = i represents member i is chosen as a corrupt member.
5.3. The model

However, due to the limitation of the tool, the same method cannot be applied for models which have more than one corrupt members. Instead, the identities of corrupt members should be assigned manually. For example, if there are two corrupt members in the model, the following is done:

\[
\text{bad_id} = 6 \rightarrow (\text{bad_guy1}= i) \land (\text{bad_guy2}= j) \land (\text{new}= \text{true}) \land (\text{bad_id}= 5);
\]

With the code above, we set the first corrupt member \text{bad_guy1} has identity \( i \) and the second one \text{bad_guy2} has identity \( j \), with \( i, j \in \{1, \ldots, n\}, i \neq j \).

By using this method, each model is divided into \( \binom{n}{c} \) models, with \( c \) is the number of corrupt members. The properties of the overall model can be computed as explained in section 5.5.

5.3.2 Path reformulation

In Crowds and Adithia networks, when a member wants to send a message, he then picks one member from the crowd to forward the message to it. The selected member who receives a message has to choose, namely forward the message to another member of the crowd or deliver the message directly to the end server. Therefore, we introduce parameters \( \text{PF} \) and \( \text{notPF} \) satisfying \( \text{PF} = 1 - \text{notPF} \) which are the probabilities that a member forward the message and the probabilities that a member deliver the message respectively.

In Onion Routing and Tarzan networks, when a member wants to send a message, he builds the path himself before the message is sent. Here we use the assumption in the analysis of Tarzan [7] that the initiator flips a biased coin to decide whether he wants to extend the path or terminate it. The probability of the initiator to extend the path is \( \text{PE} \) and the probability of the initiator to terminate the path is \( \text{notPE} \) where \( \text{notPE} = 1 - \text{PE} \).

The path construction is done on a regular basis. Therefore, we set a parameter \( \text{TotalAns} = k \), with \( k = 3, 4, 5, 6 \), which is the number of times the path construction protocol is executed. When the path construction protocol is executed then the path is reformulated. Thus the number of times the path construction protocol is executed also means the number of path reformulations in the model. With this parameter given, we would like to find out the level of observation that can be done by the corrupt members, i.e. whether they can identify the path initiator. This assumes that the corrupt members are able to link paths, i.e. the corrupt members are able to identify whether the paths are initiated by the same initiator or not.

The assumption can be adapted from the protocol used by Crowds and Adithia networks since in these networks a message is only link encrypted. However, this is not the case for Onion Routing and Tarzan networks since layered encryption is employed. But we still use the assumption for Onion Routing and Tarzan networks to investigate the possibility to use the same approach as in Crowds and Adithia networks, to analyze them.

Based on our assumptions, the corrupt member knows that he joins the path with the same initiator as the previous path. Thus he can accumulate the number of times he observes a particular member chosen before him on the path. Therefore, we introduce a state \( \text{observei} \) representing the number of times a corrupt member observe an honest member \( i \). This number increases by one every time a corrupt member observes the honest one.

In the models, we introduce a state variable \( \text{countrun} \) set to the value of \( \text{TotalAns} \) in the initial state, to count the number of path reformulation protocol that has been done.
5.3.3 Mimics construction

As mentioned before, Tarzan network introduce the concept of a traffic mimic. Upon joining the network, a member \( a \) asks \( k \) other members to exchange cover traffic with it. Similarly, \( k \) members ask member \( a \) to be their mimic. Thus, each member in the network has the same number of mimics, namely \( E(\kappa) = 2k \), to which they can route their messages.

We can translate the network into a graph which each of its vertex has the same degree, i.e. each vertex has the same number of neighbors. This kind of graph is called \( d \)-regular graph with \( d \) is the degree of each vertex. Furthermore, in this graph the vertices represent the Adithia and Tarzan’s members and the neighbors represent the mimics.

Next, consider the following theorem from [9].

**Theorem 5.3.1.** Let \( v_1, v_2, ..., v_p \) be the vertices of a graph \( G \), and let \( d_1, d_2, ..., d_p \) be the degrees of the vertices respectively. Let \( q \) be the number of edges of \( G \). Then \( \sum_{i=1}^{p} d_i = 2q \).

The theorem above determines the existence of a graph with a particular number of vertices and with given degrees for each vertex. Based on it, we can decide the number of mimics which is possible to model is 4 with any given number of members.

Furthermore, in PRISM code, the following method is used to set the network i.e. assigning which member belong to whom:

```
formula nMimics = member = 1,3,4,5,0;
```

which means that member 0’s mimics are members 1, 3, 4, 5, and itself. Here we assume that a member is the mimic of itself since in these networks, a member is allowed to pick itself to join a path.

5.4 Formalization of the anonymity properties

In this section, we formalize the anonymity properties of the Crowds, Adithia, Onion Routing, and Tarzan networks and use PRISM to analyze them as in [19]. More precisely, suppose that the coalition of corrupt members are able to link paths that originate from the same initiator, we try to answer questions, such as what is the likelihood that the corrupt members observe the initiator with higher probability than the other members and how confident the corrupt members of their observations.

The main idea of the Crowds network analysis is as follows. As proved in [18], in Crowds networks, the initiator enjoys probable innocence if the number of corrupt members \( c \), the number of members \( n \) and the probability to forward \( p_f \) satisfy [Equation 6.1]. But as described before, the probable innocence properly supports the deniability of the initiator for sending message if only one path is established. Hence if the initiator constructs multiple paths to communicate with the same server, when he is detected over multiple sessions, he cannot deny anymore that he is the real sender.

For Adithia, Onion Routing, and Tarzan networks, it has not been reported yet that the initiator enjoys probable innocence if [Equation 6.1] is satisfied. However, we extend the same approach to analyze them in order to compare the anonymity properties provided by the networks if the composition of the number of the honest and corrupt members is the same for all of them.

Furthermore, let \( K_i \) be the number of times the adversary observes a crowd member \( i \) which means that there are \( K_i \) paths in which member \( i \) selected a corrupt member as the next member.
on the path, thus enable the corrupt member to observe $i$’s identity.

Next, we consider a notion of what it means for a crowd member to be detected such that a
crowd member is detected if it is observed at least twice. Thus, we can define the following events
\[ E_{\text{det}} = K_0 > 1, \forall j \neq 0 \]  
( initiator observed twice or more)
\[ E_{\text{fpos}} = K_j > 1, \text{ for some } j \neq 0 \]  
( false positive: non-initiator observed twice or more)
\[ E_{\text{nofpos}} = K_j \leq 1, \forall j \neq 0 \]  
( complement of false positive)

To do the analyses, we compute the following:

\[ P_{\text{pos}} = P(E_{\text{det}}) \]  
( detection of the true path initiator)
\[ P_{\text{fpos}} = P(E_{\text{fpos}}) \]  
( detection of the non-initiator)
\[ P_{\text{conf}} = P(E_{\text{nofpos}} | E_{\text{det}}) \]  
( detection of only the true initiator)

In the computation, we use $p_f = p_c = 0.8$, $n = 6, 11, 12, 18, 22, \text{ and } 24$ with proportions of
corrupt members are 0.167 and 0.091. Note that the values of $p_f$, $n$, and $c$ that are used in the
computation, all satisfy Equation 6.1.

\[ 5.5 \text{ Computation method} \]

Due to the limitations of the tool, the models of Adithia and Tarzan networks should be divided
into cases to make the tool able to do the computation. More precisely, the models are separated
into files based on the properties of the corrupt members with respect to the initiator of a
message. For example, for each model of Adithia and Tarzan networks, the possible properties
of the corrupt members are:

1. all corrupt members are the initiator’s mimics,
2. some of the corrupt members are the initiator mimics, and
3. none of the corrupt members is the initiator’s mimics.

Based on the properties above, all possible combinations of corrupt members can be found from
the networks. However, due to the time limitation, we do not take all of possible combinations
of corrupt members for the computation. Instead, we take one combination of corrupt members
for each property and later on the results can be generalized for all combinations with the same
property. The overall probability results can be obtained with the following formula [11]:

\[ P \left( \bigcup_{i=1}^{n} A_i \right) = \sum_{i=1}^{n} P(A_i) - \sum_{i=1, j=1}^{n} P(A_i \cap A_j) + \sum_{i=1, j=1, k=1}^{n} P(A_i \cap A_j) - \ldots + (-1)^{n-1} P(\cap_{i=1}^{n} A_i). \]
Since in the model \( \sum_{i=1,j=1}^{n} P(A_i \cap A_j) + \sum_{i=1,j=1}^{n} P(A_i \cap A_j) - \ldots + (-1)^{n-1} P(\bigcap_{i=1}^{n} A_i) = 0\),
the formula can be simplified to:

\[
P(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} P(A_i).
\]

Here \( A_i \) is one of the events given below:

- detection of the true path initiator if property 1, 2 or 3 holds,
- detection of the non-initiator if property 1, 2 or 3 holds,
- detection of only the true initiator if property 1, 2 or 3 holds.

This computation method does not give accurate results but only gives estimated results which are enough for the analyses.
6. THE MODEL

In this chapter, we give an overview of Crowds, Adithia, Onion Routing, and Tarzan networks. Furthermore, we build models of the networks using PRISM. Based on these models, we do probabilistic analyses of the models, especially the anonymity properties provided by them.

6.1 Crowds

In this section, we discuss the Crowds network. A model of the network is built using PRISM. Furthermore, based on this model, an analysis of the model, especially the anonymity properties provided by it, is given.

6.1.1 Overview of the network

Crowds [18] is a system that provides the privacy of web transactions, especially users anonymity, which was proposed in 1998 by Reiter from Bell Laboratories, Lucent Technologies and Rubin from AT&T Labs.

The main idea of Crowds’ approach is “blending into a crowd”, which means that the system hides a user’s action among many actions of other users. With this method, any communication of a user is routed randomly through a crowd of similar users. Thus, even if an eavesdropper observes a message being sent by a particular user, he cannot be sure whether the sender is the actual user or simply forwards another user’s message.

Path setup protocol

A user in Crowds is represented by a process on his computer called jondo (from “John Doe”, the representation of someone without an identity). When a user wants to start a web transaction, the jondo is started by contacting a server called blender to request admittance to the crowd. If the jondo is admitted, the blender reports this jondo the current membership of the crowd and other information that enables the jondo to join the crowd. Therefore, every member in the system knows the identities of all other members. Furthermore, the user selects the jondo as his web proxy by specifying its host name and port number in his web browser as the proxy for all services. Thus any request coming from the browser is sent directly to the jondo.

As part of the procedure, the members establish pairwise encryption keys which are used to encrypt pairwise communication. Thus, the communication between any two jondos is link encrypted. Therefore, the contents of the communication are not secret from jondos participated in the communication but are secret from an external eavesdropper. From this fact also, we can see that the main goal of this system is to provide sender anonymity for its users rather than confidentiality.
When a user wants to initiate a transaction, his jondo starts to establish a random path of jondos that carries the request to and from the intended web servers. Each request contains the information of the intended destination. The next process is given below:

- The initiator’s jondo picks up a random jondo from the crowd, possibly itself, to forward the request to it, encrypted by the corresponding pairwise key.

- The selected jondo then flips a biased coin to decide whether it forward the request to another random jondo or submit it to the web servers directly. The biased coin has probability to forward $p_f$. So, each request travels from the initiator to a number of jondos and finally to the web server. The next jondo selected repeats this step.

![Fig. 6.1: Path setup protocol of the Crowds network. An example of a path from the initiator 0 to a web server: 0-5-5-3-C.](image)

See Figure 6.1 for an example of the path setup protocol of the Crowds network. In the figure, jondo 0 wants to initiate a transaction with web server C. First, it should build a path of jondos to carry the request. The blue dotted arrows represent the possible path it can build. For example, it picks jondo 5 to join the path. Next, jondo 5 may forward the request to other jondos, possibly itself, with probability $p_f$ (the purple dotted arrows represent other possible choices that can be taken by jondo 5), or deliver the request directly to web server C with probability $notPF$ (represented by the red dotted arrow). An example of the path built is represented by the black arrows which is through jondos 0-5-5-3 and after that jondo 3 deliver the request to web server C.

Once the path is generated, subsequent requests initiated by the same jondo to a particular web server are routed through the same path and the server may reply through the same path as the requests, only in reverse.

Each jondo who joins the path maintains an identifier for the created path. If a particular jondo appears twice on the path, the identifier should be different. This is necessary since otherwise the jondo will behave identically every time it receives a request and this results infinite loops. Each subsequent request from the initiator to the destination is routed along this path, which means the path is static, i.e. they are not modified often. This is useful to avoid corrupt crowd members from linking multiple paths and then use this information to conclude the actual initiator’s identity.
6.1. Crowds

Security analysis

Based on the degree of anonymity and the attackers’ capabilities, the Crowds networks provides a guarantee for sender anonymity to be probable innocence against the collaborating crowds members. More precisely, [18] has proved that in a crowd of size \( n \), the path initiator enjoys probable innocence against \( c \) collaborating members, if the following inequality holds:

\[
\frac{p_f}{p_f - \frac{1}{2}} (c + 1) \geq n
\]  \hspace{1cm} (6.1)

More formally, let \( H_{i+} \) be the event that at least one of the corrupt members is selected to join the path and \( I \) be the event that the initiator is chosen immediately before the corrupt member. By putting \( P(I|H_{i+}) \leq \frac{1}{2} \) as required by the probable innocence property described previously, the inequality above can be obtained.

Furthermore, the probability of beyond suspicion for the receiver anonymity against local eavesdropper increases as the number of crowd’s members increases to infinity. This happens since \( P(H_{i+}) \rightarrow 0 \) as \( n \rightarrow \infty \). Similarly, the probability of absolute privacy for the receiver anonymity against the collaborating members increases as the number of crowd’s members increases to infinity.

Since the Crowds membership changes overtime, a new path should be built periodically. But if the initiator uses multiple paths to communicate with the same server, then the anonymity provided by the network might degrades as conjectured in [18]. Later on in the discussion of the Crowds model, our automated analysis shows that this conjecture is correct.

On the other hand, the Crowds network does not provide anonymity against a global eavesdropper, who can observe all communication from and to all jondos in the crowd, since the communication is only link encrypted. Furthermore, Crowds also does not have method against a denial-service-attack by unreliable members, which happens if members do not want to pass a request accepted from other members. Moreover, an active attack by a member, such as substituting a request with a wrong information, also cannot be detected and avoided.

6.1.2 The model

In this section, we describe the probabilistic model of the Crowds network. A model using PRISM had been made by Shmatikov [19] to formalize and analyze the anonymity properties of the network. The model we build here is based on the model made by Shmatikov. In our model, some improvements are made in the organization of the model to make it clearer and easier to understand.

As explained in the previous subsection, the first part of the path construction is an initiator wanting to send a message, thus a new path through the crowd should be built. In this model, member \( i \) in the crowd who is chosen as the next member on the path are represented by a state variable \( \text{member} = i \). Furthermore, to initiate a new path construction, we introduce a state variable \( \text{new} \) which is set to the value of true in the initial state. We also introduce a state variable \( \text{start} \) which represents the process to construct the path is started. Note that when a new path is constructed, it means that a new protocol is run. The value of \( \text{countrun} \) reduce by one every time a new path is constructed. This process can be described in PRISM code as follows:

\[
[] \text{new} \land (\text{countrun} \geq 1) \rightarrow (\text{start}' = \text{true}) \land (\text{member}' = 0) \land (\text{countrun}' = \text{countrun} - 1) \land (\text{new}' = \text{false});
\]
When the path construction process is started, the member who is holding the message chooses the next member that will be on the path. Therefore, we introduce a state variable `choose` which is set to the value of false in the initial state. The next member chosen can be an honest member or a corrupt member. Furthermore, we introduce state variables `good` and `bad` that both set to the value of false in the initial state. These state variables represent the next step if an honest or a corrupt member is chosen respectively. The process is described below:

- \[ \text{start} \rightarrow (\text{choose}' = \text{true}) \land (\text{start}' = \text{false}); \]
- \[ \text{choose} \rightarrow \text{gmember: (good}' = \text{true}) \land (\text{choose}' = \text{false}) + \]
  \[ \text{bmember: (bad}' = \text{true}) \land (\text{choose}' = \text{false}); \]

If an honest member is chosen, then it means one out of the `Crowdsize` honest member is chosen with equal probability, namely \( \frac{1}{\text{Crowdsize}} \). Note that it is possible that the member sends a message to himself. After that, we also introduce a state variable `next` to represent the next step taken by the member chosen, whether to forward the message or deliver it to the end server. If the member decides to forward the message, then the state goes to the state `start` to repeat the choosing next member on the path process. Otherwise, the path is terminated and a new path can be constructed again if \( \text{countrun} \geq 1 \). The PRISM code below describes the transitions model for `Crowdsize = n`.

- \[ \text{good} \rightarrow \sum_{i=0}^{n-1} \frac{1}{n} : (\text{member}' = i) \land (\text{next}' = \text{true}) \land (\text{good}' = \text{false}); \]
- \[ \text{next} \rightarrow \text{PF: (forward}' = \text{true}) \land (\text{next}' = \text{false}) + \text{notPF: (deliver}' = \text{true}) \land (\text{next}' = \text{false}); \]
- \[ \text{forward} \rightarrow (\text{start}' = \text{true}) \land (\text{forward}' = \text{false}); \]

If a corrupt member is chosen, then he observes the member who sends the message to him, represented by the state variable `observe`. Furthermore, after receiving a message, the corrupt member terminates the path by delivering the message to the end server since he cannot obtain any additional information for his observations even if he continues building the path.

- \[ \text{bad} \rightarrow (\text{observe}' = \text{true}) \land (\text{bad}' = \text{false}); \]
- \[ \text{observe} \land (\text{member} = i) \land (\text{observe} < \text{TotalAns}) \rightarrow (\text{observe}' = \text{observe} + 1) \land (\text{deliver}' = \text{true}) \land (\text{observe}' = \text{false}); \]
- \[ \text{deliver} \rightarrow (\text{done}' = \text{true}) \land (\text{deliver}' = \text{false}); \]
- \[ \text{done} \rightarrow (\text{new}' = \text{true}) \land (\text{done}' = \text{false}); \]

The state-transition diagram of the model is given in Figure 6.2.

### 6.1.3 Results and analysis

The result obtained from our model confirms the result in [19]. The complete result is given in Table 6.1.

As we can see in Table 6.1, \( P_{pos} \) increases as the number of path reformulations grows even if not all path involving the corrupt members. This happens since the possibility of the corrupt members to be chosen increases if more paths are constructed. As a result, the possibility of the initiator is detected also increases. This means that even with only 3-6 path reformulations, the anonymity provided by the system degrades with the increase of the number of path reformulations.
Furthermore, also from the table we see that $P_{\text{conf}}$ decreases as the number of path reformulations grows. The reason is, as more paths are constructed, the chances of honest members other than the initiator appear before the corrupt member increases. This is shown by the increase of $P_{\text{fpos}}$.

Also from the table, $P_{\text{pos}}$ decreases as the size of the crowd grows as long as the proportion of corrupt members remains constant. This happens since the chances of the corrupt members to be chosen immediately after the initiator decrease as the crowd grows.

$P_{\text{conf}}$ increases as the size of the crowd grows with the constant proportion of corrupt members for any given number of path reformulations. The intuitive explanation of this result is that if the crowd size is really large, the probability that a random honest members is chosen in more than one path is negligible, while the initiator appears in every path; it at least appears once it initiates the path. This is shown by the decrease of $P_{\text{fpos}}$.

Since $P_{\text{pos}}$ decreases only slightly, assuming that $E_{\text{pos}}$ happens, the increase of $P_{\text{conf}}$ can be interpreted as a degradation of the anonymity provided by the system.

### 6.2 Adithia

In this section, we give an overview of the Adithia network. Furthermore, we build a model of the network using PRISM. Based on this model, we provide an analysis of the model, especially the anonymity properties provided.
6. The model

<table>
<thead>
<tr>
<th>Crowds size</th>
<th>Path reformulations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>15 honest, 3 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 4 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
</tbody>
</table>

Tab. 6.1: Probabilities of adversary’s observations of the Crowds network

6.2.1 Overview of the network

The Adithia network is an improvement of the Crowds network. We made this network to complete our analysis of network anonymity and to investigate the effects of the concept of mimics, as in the Tarzan network, on the anonymity provided by a system. More precisely, we adapt the concept of mimics to improve the Crowds network and we would like to see the anonymity provided by the new network after the improvement.

An Adithia network uses the same path setup protocol as the Crowds network. The improvement is in this network, each jondo $i$ has the same number of mimics as adapted from Tarzan network.

The path setup protocol of Adithia network is given as follows:

- The initiator’s jondo picks up a random jondo from its mimics, possibly itself, to forward the request to it, encrypted by the corresponding pairwise key.
- The selected jondo then flips a biased coin to decide whether it forwards the request to another random jondo from its mimics or submit it to the web servers directly. The biased coin has probability to forward $p_f$. So, each request travels from the initiator to a number of jondos and finally to the web server. The next jondo selected repeats this step.

6.2.2 The model

The model of Adithia networks is based on the Crowd network’s model with the concept of a mimics as an improvement.

Because the concept of mimics, we first number each jondo from 0 to $n$, with $n$ is the number of jondos in the crowd. Furthermore, we set the mimics in the network, as shown in subsection 5.3.3.
Instead of using probability of a member to be a corrupt member like in the Crowds model, the corrupt member is chosen randomly from jondo \( \{1, \ldots, n\} \) before the path construction is started. It means that each jondo, except the initiator, has the same probability to be a corrupt member.

After that, the path construction can be started as in the Crowds model. For checking mimics purpose, we introduce a state variable \( \text{path} \) which has an integer value, to record the previous member on the path. Furthermore, we put the initiator as the first member on the path.

\[
\text{new} \&\ (\text{countrun} \geq 1) \rightarrow (\text{start}' = \text{true}) \& (\text{member}' = 0) \& (\text{path}' = 0) \&
\text{countrun}' = \text{countrun} - 1) \& (\text{new}' = \text{false});
\]

After jondo \( i \) selects one jondo as a forwarder, we check whether the jondo is one of jondo \( i \)'s mimics. If it is not, then jondo \( i \) does the selection again.

\[
\text{check} \& (\text{path} = i) \& \neg \text{mimics} \rightarrow (\text{check}_i' = \text{true}) \& (\text{check}' = \text{false});
\]

\[
\text{check} \& (\text{path} = i) \& !\text{mimics} \rightarrow (\text{choose}' = \text{true}) \& (\text{check}' = \text{false});
\]

If the jondo selected is one of jondo \( i \)'s mimics, then we check whether the selected jondo is honest or corrupt. If it is an honest member, then we go to the next state, to enable the jondo to decide whether it wants to forward or deliver the message. If the jondo decides to forward the message then we put it on the path, i.e. update the value of state variable \( \text{path} \).

\[
\text{next} \rightarrow \text{PF} : (\text{forward}' = \text{true}) \& (\text{next}' = \text{false}) +
\text{notPF} : (\text{deliver}' = \text{true}) \& (\text{next}' = \text{false});
\]

Otherwise, the jondo does an observation of the previous jondo selected and deliver the message directly to the end server.

\[
\text{check}_i' \& (\text{member} \neq \text{bad_guy}) \rightarrow (\text{next}' = \text{true}) \& (\text{check}_i' = \text{false});
\]

\[
\text{check}_i' \& (\text{member} = \text{bad_guy}) \rightarrow (\text{observe}' = \text{true}) \& (\text{check}_i' = \text{false});
\]

Furthermore, after receiving a message, the corrupt member terminates the path by delivering the message to the end server since he cannot obtain any additional information for his observations even if he continues building the path.

\[
\text{observe} \& (\text{path} = i) \& (\text{observe}0 < \text{TotalAns}) \rightarrow
(\text{observe}' = \text{observe}0 + 1) \& (\text{deliver}' = \text{true}) \& (\text{observe}' = \text{false});
\]

The state-transition diagram of the model is given in Figure 6.3.

### 6.2.3 Results and analysis

The complete computation results for \( P_{pos} \), \( P_{conf} \) and \( P_{fpos} \) are given in Table 6.2. The table shows that \( P_{pos} \) increases as the number of path reformulations grows even the corrupt members do not join all paths. This happens since if more paths are constructed then the chances of the corrupt members to be chosen increase thus enable the corrupt members to detect the initiator.

Another result that can be seen in the table is that \( P_{conf} \) decreases as the number of path reformulations grows. The reason is, as more paths are constructed, the chances of honest
members other than the initiator appear before the corrupt member increase. This is shown by the increase of false detection $P_{fpos}$ by the corrupt members.

With respect to the number of path reformulations, it can be concluded that even with the mimics, the anonymity provided by the system degrades with the increase of the number of path reformulations.

Also from the table, $P_{pos}$ decreases as the size of the crowd grows as long as the proportion of corrupt members remains constant. This happens since the chances of the corrupt members to be chosen as mimics of the initiator decrease as the crowd grows. For example, the probability of at least one of the corrupt members is chosen as the initiator’s mimics for crowd size six with one corrupt member is 0.8 whereas for crowd size 24 with four corrupt members is about 0.5. As a result, the chances to be chosen immediately after the initiator on the path also decrease.

Furthermore, as can be seen in the table, there is no exact pattern of the values of $P_{conf}$ as the size of the crowd grows with any given proportion of corrupt members. But if the result for size 18 with three corrupt members is excluded then it can be seen that $P_{conf}$ increases with the increase of the size of the crowd. The explanation is, in our model each corrupt member has four mimics. Thus the chance of each mimic to be chosen before the corrupt member is $\frac{1}{4}$. After three path reformulations, it is most likely that each of the mimic is chosen only once. But if at least one corrupt member is chosen as one of the initiator’s mimics then the possibility of the initiator appears before the corrupt member is high since it always appear at the first location.
6.3 Onion Routing

This section discusses the Onion Routing network. Moreover, we build a model of the network using PRISM. An analysis of the model is given, in particular the anonymity properties provided.

### 6.3.1 Overview of the network

Onion Routing [17][15] was proposed by Syverson, Goldschlag, and Reed from Naval Research Laboratory. It purposes are to complicate traffic analysis, separate identification from routing, and supports many different applications.

Onion Routing provides anonymous connections that are resistant to eavesdropping and traffic analysis. When an application wants to make a connection to a machine, instead of making a socket connection directly to the destination, it makes a connection to an onion routing proxy on some remote machine. That onion routing proxy then establishes an anonymous connection through several other onion routers to the destination.

Data passes along the onion routers appear different at and to each onion router since layered

---

<table>
<thead>
<tr>
<th>Adithia size</th>
<th>Path reformulations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>15 honest, 3 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 4 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
</tbody>
</table>

Tab. 6.2: Probabilities of adversary’s observations of the Adithia network

on the path. Moreover, as the crowd size grows, the number of corrupt members also grows. For example, for crowd size 24 with four corrupt members, it is possible that all the initiator’s mimics are corrupt members. Hence the initiator certainly appears before the corrupt members whereas other honest members have no chances to be selected before the corrupt members.

Even $P_{pos}$ decreases, but the observations still happen with stronger confidence of the corrupt members as the crowd size grows. Thus, this can be interpreted as a degradation of the anonymity provided by the system.
encryption is employed by the system. Thus, data cannot be tracked en route and compromised onion routers cannot cooperate.

**Path setup protocol**

An application in an Onion Routing network is represented by an onion router. Public network contains a set of onion routers. Each onion router has a single socket connection to each of a small set of neighboring onion routers. Furthermore, an anonymous connection is routed through a sequence of neighboring onion routers. In addition, onion routers only talk to their neighbors.

In the Onion Routing network, a proxy is used to establish an anonymous connection. This proxy is called onion routing proxy. Onion routing proxy has two functions, namely link the initiator of a connection to the anonymous connection and link the anonymous connection to the responder of a connection. Furthermore, an onion routing proxy must also function as an intermediate onion router in other anonymous connections since otherwise observers can trace back the message sent from the onion routing proxy. Moreover, if the onion routing proxy is also busy as an intermediate onion router, the observers cannot distinguish whether the onion routing proxy belongs to an initiator, responder, or is only a forwarder of a message.

When an initiator wants to execute a transaction with a machine, he should first contact his onion routing proxy to build an anonymous connection and after the anonymous connection is ready, it can be used to transmit data. The detailed steps are as follows:

- The initiator’s proxy chooses several others onion routers through which it wants to make an anonymous connection.
- The initiator’s proxy constructs a layered data structure called onion. Each layer of the onion contains the identity of the next onion router in the connection to which the onion should be sent afterwards, and a key to be used when communicating with the previous onion router in the connection. More detail explanation about the onion can be found in [16]. See [Figure 6.4](#) for an example of an onion.
- The initiator’s proxy sends the onion to the first onion router in the connection. The onion router peels of the outermost layer to get the key and sends the onion to the next onion router in the connection. Each of the rest onion routers in the connection repeats this step until lastly the responder’s proxy obtains its key and after that the anonymous connection is ready to use for transmitting data.
- Data is possible to be sent forward or backward. To send data forward (to the responder), the initiator first sends the data as a plain text to its proxy.
- The initiator’s proxy repeatedly encrypts the data using the keys in the onion starting from the innermost key and after that sends the encrypted data through the anonymous connection.
- Each onion router removes one layer of encryption and sends the data to the next onion router until the data arrives at the responder’s proxy. The responder’s proxy removes the last layer of encryption and obtains a plain text. Next, it forwards the plain text to the responder.

To send data backward, the same method as sending data forward is used, only the direction and the encryption are the reverse of sending data forward.
Fig. 6.4: An onion, with X is the first onion router which appears immediately after the initiator’s onion routing proxy in the connection, and Z is the responder’s onion routing proxy.

It is possible that the initiator expects a reply from the responder. One way to receive the reply is letting the connection open. But there is another way namely a reply onion \[17\] that can be constructed by the initiator’s proxy to define a route back to it.

Each layer of the onion is intended for a particular onion router and encrypted by using the public key of the intended onion router thus only the intended recipient can peel off the outermost layer. The public key is generated by each onion router once they join the network. The innermost layer is intended for the last onion router in the connection, which also functions as the responder’s onion routing proxy, thus it does not contain the identity of the next onion router in the connection. Furthermore, onion routers obtain no other information from an onion except the identity of whom they receive the onion from and to whom they have to send the embedded onion.

The key at each onion router can be used for forward and backward communication. The second one is useful when the responder wants to send a reply data to the initiator. Each layer of the onion also contains an expiration time. With this expiration time, an onion router may ignore expired and replayed onions.

Another feature of the Onion Routing network is a destroy message. If the connection is broken then anonymous connections should be destroyed. An onion router that decides to destroy the connection then sends a destroy message forward and backward. Upon receiving the destroy message, onion routers should clean its own table which contains identifiers of incoming and outgoing connections, and forward the destroy message in the same direction.

Security analysis

The Onion Routing network provides sender and receiver anonymity against both corrupt onion routers and global eavesdroppers not only because of the path construction protocol but also because the data passes through the anonymous connection are layered encrypted.

However, both of the initiator and the responder identities are known by the initiator’s proxy thus if it is compromised then all information is revealed. But in general, it is enough to have a single uncompromised onion router to complicate a traffic analysis.

Additionally, Onion Routing networks also provides an unlinkability of both sender and receiver, and the data sent. The first one is because of the path construction protocol and the latter is because the data passes through the connection looks different at and to each onion router.

Onion Routing uses the expiration times contained in onions to prevent replay attacks. But if
the clock is not well synchronized then the expiration times may cause denial of service.

However, Onion Routing is still vulnerable to traffic analysis attacks [17]. When the attackers have enough data, it is possible for them to analyze the usage patterns and make a deduction about the routing message. Dummy traffic can be added to reduce this kind of attack but it causes cost and network efficiency problems later on.

Moreover, Onion Routing also does not have a method against a denial-service-attack caused by compromised onion routers. The compromised onion routers can send destroy message to tear down the connection. Another way is it can simply refuse to send any data pass through it.

6.3.2 The model

When an initiator wants to send data via an Onion Routing network, an anonymous connection should be built. In the first step, the initiator chooses several other onion routers to join the connection. Here we assume that once the initiator finishes choosing onion routers to be in the connection, it also finishes sending the onion so that the connection is ready to use.

The new connection construction is represented by the command given below. In the Onion Routing network, an onion router is chosen and put in the connection afterwards. Therefore we introduce a state variable or_path=i to represent that member i is already in the connection. Since member=0 is the initiator then it takes the first position in the connection. Thus every time a new connection is constructed, we put or_path=0.

\[
\text{new} \land (\text{countrun} \geq 1) \rightarrow (\text{start'} = \text{true}) \land (\text{member'} = 0) \land (\text{or_path'} = 0) \land (\text{countrun'} = \text{countrun} - 1) \land (\text{new'} = \text{false});
\]

Once the initiator starts establishing a connection, an onion router is chosen to be the next hop in the connection. Different with the Crowds model, in Onion Routing, the initiator only appears in the first location in the connection. As an additional assumption, we assume that onion routers other than the initiator may appear more than once in the connection. The following PRISM code describes the selection of an onion router among honest members.

\[
\text{good} \rightarrow \frac{1}{4} : (\text{member'} = 1) \land (\text{put'} = \text{true}) \land (\text{good'} = \text{false}) + \frac{1}{4} : (\text{member'} = 2) \land (\text{put'} = \text{true}) \land (\text{good'} = \text{false}) + \frac{1}{4} : (\text{member'} = 3) \land (\text{put'} = \text{true}) \land (\text{good'} = \text{false}) + \frac{1}{4} : (\text{member'} = 4) \land (\text{put'} = \text{true}) \land (\text{good'} = \text{false});
\]

After an honest member is chosen, it is put in the connection by updating the value of or_path as follows:

\[
\text{put} \rightarrow (\text{or_path'} = \text{member}) \land (\text{next'} = \text{true}) \land (\text{put'} = \text{false});
\]

As the next step, the initiator may extend the path of connection or terminate it. If the initiator decides to terminate the path of connection then it can start sending data through it.

\[
\text{next} \rightarrow \text{PE} : (\text{extend'} = \text{true}) \land (\text{next'} = \text{false}) + \text{notPE} : (\text{send'} = \text{true}) \land (\text{send_id'} = 1) \land (\text{next'} = \text{false});
\]

\[
\text{extend} \rightarrow (\text{start'} = \text{true}) \land (\text{extend'} = \text{false});
\]
If a corrupt member is chosen, then it is also put on the path represented by a state variable put\_bad. Afterwards, we introduce a state variable continue. With this variable, we assume that the initiator continues building the connection and after the connection is ready, the initiator can start sending data through it.

\[
\text{put\_bad} \rightarrow (\text{continue}'=\text{true}) \& (\text{put\_bad}'=\text{false});
\]
\[
\text{continue} \rightarrow (\text{send}'=\text{true}) \& (\text{send\_id}'=2) \& (\text{continue}'=\text{false});
\]

If all onion routers chosen are honest then the protocol is finished after the data sending process is completed. If at least one of the onion routers chosen is corrupt then it does an observation to the onion router appears before it in the connection and after that finishes the process.

\[
\text{send} \& (\text{send\_id} = 1) \rightarrow (\text{finish}'=\text{true}) \& (\text{send}'=\text{false});
\]
\[
\text{send} \& (\text{send\_id} = 2) \rightarrow (\text{observe}'=\text{true}) \& (\text{send}'=\text{false});
\]
\[
\text{observe} \& (\text{or\_path} = i) \& (\text{observe0} < \text{TotalAns}) \rightarrow
\]
\[
(\text{observe}' = \text{observe} + 1) \& (\text{finish}' = \text{true}) \& (\text{observe}' = \text{false});
\]

The state-transition diagram of the model is given in Figure 6.5.

### 6.3.3 Results and analysis

The complete computation results for $P_{pos}$, $P_{conf}$ and $P_{fpos}$ are given in Table 6.3.

As we can see in the table, $P_{pos}$ increases with the increase of the number of path reformulations without assuming that the corrupt members are selected to join the path. The explanation of this result is that if more paths are constructed then the chances of the corrupt members to be chosen increase thus enable the corrupt members to detect the initiator. This means that the anonymity provided by the system degrades with the increase of the number of path reformulations.

Another result that can be seen in the table is that $P_{conf}$ decreases as the number of path reformulations grows. The reason is, as more paths are constructed, the chances of honest members other than the initiator appear before the corrupt member increase. This is shown by the increase of false detection $P_{fpos}$ by the corrupt members.

A nice result of $P_{pos}$ with respect to the size of the crowd is obtained from PRISM’s automated analysis. The table shows that as the size of the crowd grows with the same proportion of corrupt members, $P_{pos}$ remains the same. This happens because in the Onion Routing network, the corrupt members can detect the initiator only if they are chosen to be at the second location on the path (remember that in this network, the initiator only appears at the first location on the path). Since the proportion of the corrupt members is all the same for all sizes of the crowd thus the probability of the corrupt members to be at the second location on the path is all the same.

The table also shows that for any given number of path reformulations, $P_{conf}$ increases as the size of the crowd grows with the same proportion of corrupt members. The reason is in a large crowd, the probability that a random honest members is chosen in more than one path is negligible, while the initiator appears in every path; it at least appears once it initiates the path. This is shown by the decrease of $P_{fpos}$.

From the result, even $P_{pos}$ remains the same as the size of the crowd grows with the same proportion of corrupt members, the detection always happen with a high confidence of the corrupt members. Thus, this can be interpreted as a degradation of the anonymity provided by the system.
6.4 Tarzan

This section gives an overview of the Tarzan network. Furthermore, we build a model of the network using PRISM. Based on this model, we do some analysis of the model, especially the anonymity properties provided.

6.4.1 Overview of the network

Tarzan [7] is a peer-to-peer anonymous network developed by Freedman and Morris in 2002. The anonymity provided by Tarzan is achieved from layered encryption that is employed and multi-hop routing, i.e. an initiator of a message builds a path of several other user through a message is routed. As for the encryption, Tarzan uses layered encryption similar to Chaumian mix [3]. Furthermore, the Tarzan network introduces the concept of a traffic mimic that complicates a traffic analysis since each user can exchange cover traffic with its mimics. In addition, as mentioned before in section 5.1 a communication in a Tarzan network can only be routed between
6.4. Tarzan

Tab. 6.3: Probabilities of adversary’s observations of Onion Routing network

<table>
<thead>
<tr>
<th>Onion Routing size</th>
<th>Path reformulations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>15 honest, 3 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 4 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
</tbody>
</table>

Tarzan is designed to be application independent, i.e. transparent to existing application and allow users to interact with existing services. To achieve this, Tarzan provides the abstraction of an IP tunnel. Furthermore, Tarzan provides sender and recipient anonymity against colluding users and also global eavesdroppers. The message sent through this network cannot be linked to any sender or recipient. As part of the system, Tarzan has a method to resist adversaries’ attempt to block the system.

In this network, the following terminologies are used: a node is an Internet host’s virtual identity in the system. The virtual identity can be created by running an instantiation of the Tarzan software on a single IP address. A tunnel is a virtual circuit of a sequence of nodes used to send packets. We use a term path in Adithia and Crowds networks and anonymous connection in the Onion Routing network which all represent the same meaning. A relay is a node acting as a packet forwarder as part of a path.

All nodes in Tarzan run software that, (1) discovers all other participating nodes thus Tarzan does not have a centralized registration such as in Crowds network, (2) intercepts a packet generated by local applications that should be anonymized, (3) manages tunnels to anonymize packets, (4) forward packets, (5) operates PNAT (pseudonymous network address translator) to forward a packet onto the ordinary Internet.

Upon joining the network, each node asks several other nodes to exchange mimic traffic with it. These nodes become mimics to each other, thus each node has the same number of mimics. The mimics are assigned verifiably at random from the set of nodes in the network. Furthermore, a packet only can be routed between mimics. See Figure 6.6 for an example of a Tarzan network. To complicate traffic analyses, a node establishes a cover traffic with its mimic, where a real packet can be inserted but indistinguishable from the cover traffic. In Tarzan, cover traffic is possible to be added without causing cost efficiency problems since the number of users exchanging the mimics, thus restricting the choices of users to join a path.
cover traffic is limited. The mimic relationship should be verifiable to stop an adversary from selecting more mimics that other nodes do.

Tarzan uses a simple gossip-based protocol for peer discovery, i.e. method for each node to learn about all other nodes (see [7] for detail explanation of Tarzan’s peer discovery). A method to select nodes from all possible nodes is also introduced. Tarzan does not choose nodes simply randomly among all of nodes based on their IP addresses, but chooses among distinct IP prefixes to decrease the possibility of malicious nodes to be chosen [7].

Fig. 6.6: An example of the Tarzan network where each node has four mimics. The nodes in yellow circle are node 0’s mimics. An example of Tarzan’s tunnel is shown by the red arrows.

Path setup protocol

When an initiator wants to send a message using Tarzan network, the initiator should build a tunnel first. The steps of tunnel setup protocol are given below.

- The initiator’s node running an application to select a set of nodes to form the tunnel. More precisely, first, the initiator chooses one of its mimics as its first tunnel relay. After that, the selected mimic send a list of its own mimics to the initiator. The initiator knows most of the mimics from gossiping and can verify them. Then the initiator chooses one node from the list to be its second relay. This process is repeated until the initiator decides to stop. Next, the initiator chooses a random node to run the PNAT. Figure 6.6 gives an example of a tunnel with node 0 as the initiator.

- The initiator generates the symmetric keys that are used for the layered encryption. The keys then are distributed to the relays encrypted by their corresponding public keys. Note that the public keys are generated by each node the first time they join the network and the initiator can find the public keys in the peer discovery process.

Tarzan has two types of message: data packets to be relayed through the tunnel, and control packet that contains commands and responses that establish and maintain the tunnel. Both packets are encapsulated inside UDP. To relay a packet, the initiator perform a nested encoding for each tunnel relay and encapsulates the result in a UDP packet. For each packet, symmetric keys are used to hide the data and a MAC is used to protect the integrity of the data. Furthermore, separate keys are used in each direction of each relay. Detail explanation of the nested encoding can be found in [7].
Upon receiving the packet, in the forward direction, the first relay in the tunnel performs an integrity check, strips one layer of encryption and sends the packet to the next relay. If the packet fails the integrity check then it is dropped by the relay. This process continues until the packet reaches the last relay. This relay reveals the initiator’s packet and send it to the node running the PNAT.

In the backward direction, each successive relay encrypts a packet using its appropriate key for the reverse direction, re-tags and forward the packet in the reverse direction.

It is possible that a tunnel fails sending packets. This happens because one of its relays stop forwarding packets. To detect the failure, the initiator regularly sends ping messages to the PNAT through the tunnel. If the node that run the PNAT does not response the ping message then a new node to run the PNAT is chosen by the initiator. If there is a node between the initiator and the PNAT does not response the ping message then a new tunnel is rebuilt starting from the immediate node after the broken node.

Security analysis

Considering static adversaries, Tarzan provides sender anonymity against both colluding relays and global eavesdroppers. Because of the multi-hop routing alone, compromised relays cannot be sure whether its predecessor relay in the tunnel is the initiator or simply a forwarder of a packet. In addition to the multi-hop routing, Tarzan uses cover traffic that more complicates the analysis of the compromised relays since all nodes both originate and forward traffic. Furthermore, Tarzan also provides recipient anonymity and unlinkability of packets pass through each relay because of the layered encryption.

Other possible attacks to the Tarzan network may be done by time-bounded adaptive adversaries [7]. These adversaries can pick and choose which machine to compromise after it joins the system. To protect against this kind of adversaries, the period to compromise all tunnel relays must be longer than the tunnel’s duration and tunnels should not be built repeatedly through the same small size of largely-compromised relays. The design of Tarzan makes sure that the period to compromise is non-trivial. Furthermore, in the tunnel construction, Tarzan provides a large choice to be chosen. In addition, Tarzan’s mimics are reassigned daily thus the initiator’s possible relays changes daily also.

To protect against fake entries, Tarzan distinguishes between validated and unvalidated addresses in the peer discovery and selection process. Thus, honest nodes only broadcast the validated addresses and only select their mimics from the set of validated addresses.

A similar quantitative analysis as the analysis of the Crowds network [15] is also given in [7]. The significant difference between both analysis is the Tarzan’s restricting relay choice to source-routing mimics. The analysis can briefly described as follows. Let \( AS_i \) be the set of possible initiators of a packet and let \( E(|AS_i|) \) be the size of the set. Furthermore, let \( h_{i+} \) be a corrupt relay which appears at or after position \( i \). The confidence of \( h_{i+} \) that a specific node from the set of initiators is the initiator is formulated below.

\[
C_i = \frac{Pr(I_i|H_{i+})}{E(|AS_i|)}
\]  

where \( H_{i+} \) is the event that the first corrupt relay \( h_{i+} \) appears at or after position \( i \) and \( Pr(I_i|H_{i+}) \) is the adversary’s confidence that some nodes preceding it by \( i \) hops is the tunnel initiator.

The plot of \( C_i \) with \( i = 1, ..., 5 \) and the number of each node’s mimics is 6, is given in the appendix part of [7]. The plot shown that if \( \frac{M}{N} \to 1 \) then \( C_1 \to 1 \), with \( M \) is the number of corrupt relays
and $N$ is the number of nodes in the network. Whereas when $i = 5$ then $C_5$ is less then 10% even $\frac{M}{N} \rightarrow 1$.

### 6.4.2 The model

The first part of the Tarzan network’s model is setting up the mimics in the network which can be done by first numbering the nodes from 0 to $n$, with $n$ is the number of nodes, and then set up the mimics in the network as explained in subsection 5.3.3.

Furthermore, before the path construction is started, as a preliminary step, the corrupt members are chosen randomly from nodes $\{1, ..., n\}$. Note that node 0 is always be an honest node.

When an initiator wants to send data via a Tarzan network, a tunnel should be built. The first step is the initiator chooses several other nodes to join the connection. Here we assume that once the initiator finish choosing nodes to be in the tunnel, it has chosen the node to run the PNAT and also finishes sending the symmetric keys so that the tunnel is ready to use.

The new tunnel construction is represented by the command given below. In Tarzan networks, a node is chosen and put in the tunnel afterwards. Therefore we introduce a state variable $\text{path} = i$ to represent that member $i$ is already in the tunnel. This state variable is also used to check whether the next node chosen is the previous node’s mimic. Since $\text{member} = 0$ is the initiator then it takes the first position in the tunnel. Thus every time a new tunnel is constructed, we put $\text{path} = 0$.

$\square \text{new } \& (\text{countrun} \geq 1) \rightarrow (\text{start'} = \text{true}) \& (\text{member'} = 0) \& (\text{path'} = 0) \& (\text{countrun'} = \text{countrun} - 1) \& (\text{new'} = \text{false})$;

Once the initiator starts establishing a tunnel, a node is chosen to be the next hop in the tunnel. Similar with the Onion Routing network, in Tarzan networks, the initiator only appears in the first location in the tunnel. As an additional assumption, we assume that nodes other than the initiator may appear more than once in the tunnel. The following PRISM code describes the selection of a node among all nodes.

$\square \text{choose } \rightarrow 1/5 : (\text{member'} = 1) \& (\text{check'} = \text{true}) \& (\text{choose'} = \text{false}) +
\quad 1/5 : (\text{member'} = 2) \& (\text{check'} = \text{true}) \& (\text{choose'} = \text{false}) +
\quad 1/5 : (\text{member'} = 3) \& (\text{check'} = \text{true}) \& (\text{choose'} = \text{false}) +
\quad 1/5 : (\text{member'} = 4) \& (\text{check'} = \text{true}) \& (\text{choose'} = \text{false}) +
\quad 1/5 : (\text{member'} = 5) \& (\text{check'} = \text{true}) \& (\text{choose'} = \text{false})$;

After node $i$ selects one other node as the next hop in the tunnel, we check whether the node is one of node $i$’s mimic. If it is not, then node $i$ does the selection again.

$\square \text{check } \& (\text{path} = i) \& \neg \text{mimics } \rightarrow (\text{check_again'} = \text{true}) \& (\text{check'} = \text{false})$;
$\square \text{check } \& (\text{path} = i) \& \neg \text{!mimics } \rightarrow (\text{choose'} = \text{true}) \& (\text{check'} = \text{false})$;

Otherwise, whether the selected node is honest or corrupt is checked. If the node chosen is honest, then it is put in the tunnel represented by a state variable $\text{put}$. If it is corrupt, then it is also put in the tunnel. But instead of using the state variable $\text{put}$ to represent this condition, a new state variable $\text{put\_bad}$ is introduced to make the model more simple.

$\square \text{check\_again } \& (\text{member} \neq \text{bad\_guy}) \rightarrow (\text{put'} = \text{true}) \& (\text{check\_again'} = \text{false})$;
$\square \text{check\_again } \& (\text{member} = \text{bad\_guy}) \rightarrow (\text{put\_bad'} = \text{true}) \& (\text{check\_again'} = \text{false})$;
When an honest node $i$ is put in the tunnel, the value of the state variable $\text{path}$ is updated into equal to $i$. After that, the initiator may decide the next step, i.e. whether extending the tunnel or finishing building the tunnel thus it means the tunnel is ready to use for sending data.

\[
\begin{align*}
\text{next} & \rightarrow \text{PE} : (\text{extend}' = \text{true}) & (\text{next}' = \text{false}) + \\
& \quad \text{notPE} : (\text{send}' = \text{true}) & (\text{send_id}' = 1) & (\text{next}' = \text{false})
\end{align*}
\]

\[
\begin{align*}
\text{extend} & \rightarrow (\text{start}' = \text{true}) & (\text{extend}' = \text{false})
\end{align*}
\]

While when a corrupt node is put in the tunnel, then a new state variable $\text{continue}$ is introduced. With this variable, we assume that the initiator continue building the tunnel and after the tunnel is ready then the initiator can start sending data through it.

\[
\begin{align*}
\text{put_bad} & \rightarrow (\text{continue}' = \text{true}) & (\text{put_bad}' = \text{false})
\end{align*}
\]

\[
\begin{align*}
\text{continue} & \rightarrow (\text{send}' = \text{true}) & (\text{send_id}' = 2) & (\text{continue}' = \text{false})
\end{align*}
\]

If all nodes chosen are honest, then the protocol is finished after the data sending process is completed. If at least one of the nodes chosen is corrupt, then it does an observation to the node appears before it in the tunnel and after that finishes the process.

\[
\begin{align*}
\text{send} & & (\text{send_id} = 1) \rightarrow (\text{finish}' = \text{true}) & (\text{send}' = \text{false})
\end{align*}
\]

\[
\begin{align*}
\text{send} & & (\text{send_id} = 2) \rightarrow (\text{observe}' = \text{true}) & (\text{send}' = \text{false})
\end{align*}
\]

\[
\begin{align*}
\text{observe} & & (\text{path} = i) & (\text{observe0 < TotalAns}) \rightarrow \\
& & (\text{observe}' = \text{observe} + 1) & (\text{finish}' = \text{true}) & (\text{observe}' = \text{false})
\end{align*}
\]

The state-transition diagram of the model is given in Figure 6.7.

### 6.4.3 Results and analysis

The complete computation results for $P_{\text{pos}}$, $P_{\text{conf}}$ and $P_{\text{fpos}}$ are given in Table 6.4. Similar results as another network with mimics, the Adithia network, are obtained from the Tarzan model. As can be seen in Table 6.4, $P_{\text{pos}}$ increases as the number of path reformulations grows even the corrupt members do not join all paths. This happens since if more paths are constructed then the chances of the corrupt members to be chosen increase thus enable the corrupt members to detect the initiator.

The table also shows that $P_{\text{conf}}$ decreases as the number of path reformulations grows. The reason is as more paths are constructed, the chances of honest members other than the initiator appear before the corrupt member increase. This is shown by the increase $P_{\text{fpos}}$, which means that the corrupt members observe honest members other than the initiator more often with the increase of the number of path reformulations.

We can conclude that even with the mimics, the anonymity provided by the system degrades with the increase of the number of path reformulations.

Also from the table, $P_{\text{pos}}$ decreases with the increase of the size of the crowd as long as the proportion of corrupt members remains constant. This is happened since the chances of the corrupt members to be chosen as the mimics of the initiator decrease as the the crowd grows. See also the explanation in . However, $P_{\text{pos}}$ decreases really slightly; the values are almost the same. The reason is, since in Tarzan, the initiator always at the first location on the path, then the corrupt members can observe the initiator only if they are chosen to be at the second location on
Fig. 6.7: The state-transition diagram of the Tarzan model

the path. Once the corrupt members be part of the initiator’s mimics, their chances to be chosen to be at the second location are all the same. Then now, the decrease of $P_{\text{pos}}$ is only depend on the number of corrupt members in each given crowd with respect to the number of mimics owned by the initiator.

Furthermore, the table shows that there is no exact pattern of the values of $P_{\text{conf}}$ as the size of the crowd grows with any given proportion of corrupt members. But if the result for size 12 with two corrupt members is excluded then the pattern is clear, namely $P_{\text{conf}}$ increases with the increase of the size of the crowd. The explanation is in our model, each corrupt member has four mimics. Thus the chance of each mimic to be chosen before the corrupt member is $\frac{1}{4}$. After three path reformulations, it is most likely that each of the mimic is chosen only once. But if at least one corrupt member is chosen as one of the initiator’s mimics then the possibility of the initiator appears before the corrupt member is high since it always appear at the first location on the path. Furthermore, as the crowd size grows, the number of corrupt members also grows. For example, for crowd size 24 with four corrupt members, it is possible that all the initiator’s mimics are corrupt members. Thus the initiator certainly appears before the corrupt members
6.5 Conclusions of the results

From the result in general, $P_{pos}$ decreases with the increase of the number of path reformulations and the crowd’s size as long as the proportion of corrupt members remains constant. However, a nice result is obtained from Onion Routing’s analysis namely $P_{pos}$ remains constant as the crowd size grows as long as the proportion of corrupt members remain constant. Almost similar result is also obtained from the analysis of the Tarzan network. Here, $P_{pos}$ is almost constant as the crowd size grows as long as the proportion of corrupt members remain constant. These mean, in Onion Routing and Tarzan networks, the crowd’s size does not affect the detection of the corrupt members.

Furthermore, as well as $P_{pos}$ in general, $P_{conf}$ decreases with the increase of the number of path reformulations. On the other hand, $P_{conf}$ increases as the crowd size grows with the same proportion of corrupt members.

The overall conclusions that can be taken based on the results are as follows: the anonymity provided by the networks degrades as the crowd size grows as long as the proportion of corrupt members remains constant, and with the increase of the number of path reformulations.

Similarly, $P_{fpos}$ increases as the crowd size grows with the same proportion of corrupt members.

<table>
<thead>
<tr>
<th>Tarzan size</th>
<th>Path reformulations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>15 honest, 3 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 4 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>10 honest, 1 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
<tr>
<td>20 honest, 2 corrupt</td>
<td>$P_{pos}$</td>
</tr>
<tr>
<td></td>
<td>$P_{conf}$</td>
</tr>
<tr>
<td></td>
<td>$P_{fpos}$</td>
</tr>
</tbody>
</table>

Tab. 6.4: Probabilities of adversary’s observations of the Tarzan network

whereas other honest members have no chances to be selected before the corrupt members.

Even $P_{pos}$ decreases, the observations still happen with stronger confidence of the corrupt members as the crowd size grows. Thus this can be interpreted as a degradation of the anonymity provided by the system.

### 6.5 Conclusions of the results

From the result in general, $P_{pos}$ decreases with the increase of the number of path reformulations and the crowd’s size as long as the proportion of corrupt members remains constant.

However, a nice result is obtained from Onion Routing’s analysis namely $P_{pos}$ remains constant as the crowd size grows as long as the proportion of corrupt members remain constant. Almost similar result is also obtained from the analysis of the Tarzan network. Here, $P_{pos}$ is almost constant as the crowd size grows as long as the proportion of corrupt members remain constant. These mean, in Onion Routing and Tarzan networks, the crowd’s size does not affect the detection of the corrupt members.

Furthermore, as well as $P_{pos}$ in general, $P_{conf}$ decreases with the increase of the number of path reformulations. On the other hand, $P_{conf}$ increases as the crowd size grows with the same proportion of corrupt members.

The overall conclusions that can be taken based on the results are as follows: the anonymity provided by the networks degrades as the crowd size grows as long as the proportion of corrupt members remains constant, and with the increase of the number of path reformulations.
7. DISCUSSION

In this chapter, the description of the square problem is given. Based on the square representation, Crowds, Adithia, Onion Routing, and Tarzan networks are compared based on the PRISM-supported analysis results given in previous chapter.

7.1 The Anonymity Square problem

![Anonymity Square diagram]

Fig. 7.1: The representation of the problem as an Anonymity Square problem

To be able to compare the anonymity provided by the networks discussed, we first translate our problem into a square. See [Figure 7.1]. The square is built based on the specifications of the networks. We have two categories that determine the significant differences between the networks, namely the encryption method which is employed and the choice of forwarder selection. For the first category, the encryption method is divided into two methods: link and layered encryption. Note that as part of the procedure to employ layered encryption, the path is all decided by the initiator. Thus the initiator only appears at the first location on the path. While for the second category, there are two types of choice of forwarder selection namely global forwarder which means that the forwarder can be chosen among all of the members, and local forwarder which means that the forwarder only can be chosen among a small group of members, i.e. the mimics.

Based on the categories, now we can put each of the network as a corner of the square, such that Crowds as the upper left corner, Onion Routing as the upper right corner, Adithia as the bottom left corner, and Tarzan as the bottom right one.

Furthermore, the edges of the square represent one step improvement of one network to become another. Regarding the encryption method, Onion Routing and Tarzan improve Crowds and Adithia networks, by changing the link encryption employed into layered encryption. Similarly, based on the forwarder selection, Adithia and Tarzan improved upon Crowds and Onion Routing networks by adding the concept of mimics.
7.2 Comparison between Crowds, Adithia, Onion Routing, and Tarzan networks

In this section, the anonymity provided by Crowds, Adithia, Onion Routing, and Tarzan networks are compared based on the square, following the direction of the arrow on each edge starting from Crowds.

7.2.1 Crowds → Onion Routing → Tarzan

The first comparison is following this path: Crowds → Onion Routing → Tarzan. More precisely, this comparison is divided into two parts. The first part is the comparison between Crowds and Onion Routing and the second one is the comparison between Onion Routing and Tarzan.

As can be seen in Table 6.1 and Table 6.3, the results of $P_{pos}$ of Onion Routing’s analysis are much lower than of Crowds’. The reason is in Onion Routing, the initiator only appears at the first location on the path. Thus the corrupt members only can detect the initiator if they are chosen at the second position. Comparing with in Crowds, where the detection can be done at any location on the path as long as the corrupt members are chosen immediately after the initiator, the corrupt members in Onion Routing have really limited chances.

On the other hand, the results of $P_{conf}$ of Onion Routing’s analysis are higher in general. However, the difference is only slightly. Hence, since in Onion Routing networks the possibility of the corrupt members to detect the initiator does not depend on the crowd size (as long as the proportion of corrupt members remains constant), it can be concluded that Onion Routing networks are better than Crowds.

The fact that the crowd size in Onion Routing networks does not affect the observations of the corrupt members gives an additional proof that Onion Routing networks are better than Crowds networks.

Remember that in the model of Onion Routing networks, we still use the assumption that the corrupt members can link paths, which is not the case for Onion Routing since layered encryption is employed. But even with this assumption, Onion Routing is still a lot better than Crowds.

Next, we continue by comparing Onion Routing and Tarzan networks. Table 6.3 and Table 6.4 show that the results of $P_{pos}$ of Tarzan’s analysis are much greater than of Onion Routing’s. The explanation is that in Tarzan networks, if at least one of the corrupt members is the initiator’s mimic then the chance for him to detect the initiator increases. Even worse, it is possible that all corrupt members be the initiator’s mimics.

In contrast, the results of $P_{conf}$ of Tarzan’s analysis are lower in general. This is because in Tarzan, if not all of the corrupt members are the initiator’s mimics then the possibilities of a random honest member appears before the corrupt members increases. Since the difference between the results of $P_{conf}$ of Tarzan’s analysis and of Onion Routing’s is only slightly, it can be concluded that Onion Routing is more secure than Tarzan based on the results of $P_{pos}$.

From the second part of the comparison, we can conclude that with the concept of mimics alone, a network can still be vulnerable. Hence, layered encryption and cover traffic method should be employed as additional means to overcome the vulnerabilities.
7.2.2 Crowds $\rightarrow$ Adithia $\rightarrow$ Tarzan

The second comparison is following this path: Crowds $\rightarrow$ Adithia $\rightarrow$ Tarzan. More precisely, this comparison is divided into two parts. The first part is the comparison between Crowds and Adithia and the second one is the comparison between Adithia and Tarzan.

We first begin with the comparison between Crowds and Adithia networks. It can be seen in Table 6.1 and Table 6.2 that the results of $P_{\text{pos}}$ of Adithia’s analysis are higher than of Crowds’. As expected, the results of $P_{\text{conf}}$ of Adithia’s analysis are lower in general, although only slightly. To explain this, similar reasoning can be taken as in the comparison between Onion Routing and Tarzan since Adithia and Tarzan have the same way of improvement from Crowds and Onion Routing networks respectively, namely the addition of the concept of mimics.

Since the difference between the results of $P_{\text{conf}}$ of Tarzan’s analysis and of Crowds’ is only slightly, it can be concluded that Crowds is more secure than Tarzan based on the results of $P_{\text{pos}}$.

In this case, note that Adithia network employs link encryption like Crowds. Thus, the assumption that the corrupt members can link paths still can be used. Based on the first part of our comparison, we can conclude that Crowds is better than Adithia network. Furthermore, from this comparison, we can clearly see that the concept of mimics makes a network more vulnerable if it is applied alone.

As the next step, we continue by comparing Adithia and Tarzan networks. See Table 6.2 and Table 6.4. From the table, we can see that the results of $P_{\text{pos}}$ of Tarzan’s analysis are lower than of Adithia’s. The explanation of this is similar with the explanation for the comparison of Crowds and Onion Routing in the previous subsections since Onion Routing and Tarzan have the same way of improvement from Crowds and Adithia networks respectively, namely the used of layered encryption which also means that the initiator only appears at the first location on the path. As well as $P_{\text{pos}}$, the results of $P_{\text{conf}}$ of Tarzan’s analysis are slightly lower than of Adithia’s.

In the model of Tarzan networks, we still use the assumption that the corrupt members can link paths, which is not the case for Tarzan network since layered encryption is employed. But even with this assumption, the results of both $P_{\text{pos}}$ and $P_{\text{conf}}$ of Tarzan’s analysis are less than of Adithia’s then we can conclude that Tarzan networks are more secure than Adithia networks. The fact that the crowd size in Tarzan networks does not affect the observations of the corrupt members gives an additional proof that Tarzan networks are better than Adithia networks.

7.3 Conclusions of the discussion

In the previous sections, the comparison of anonymity provided by Crowds, Adithia, Onion Routing, and Tarzan networks is given. From the discussion of the comparison, there are two important points that should be remarked.

The first one is related with the concept of mimics. From the discussion, it can be concluded that the addition of the concept of mimics alone can make networks more vulnerable. This can be seen in the comparison between Onion Routing and Tarzan, and Crowds and Adithia. The reason is in networks with mimics, once the corrupt members be the initiator’s mimics, their chances to detect the initiator become really high. However, if all of the initiator’s mimics are honest than the networks are completely secure from observations.

Based on the explanation above, it is important to employ the mimics’ concept not only to limit
the choice of forwarder selections but also to exchange cover traffic between minics. Furthermore, it is necessary to apply layered encryption to make networks more secure.

For the second remark, it is important that the initiator only appears at the first location on the path since this way, the chances of corrupt members to detect him greatly decrease. This is shown by the results of $P_{pos}$ of Onion Routing’s and Tarzan’s analysis, which are less than the results of $P_{pos}$ of Crowds's and Adithia's analysis respectively.
8. CONCLUDING REMARKS

This report discussed probabilistic analysis of the anonymity provided by anonymity networks, namely Crowds, Adithia, Onion Routing, and Tarzan, by using a model-checker tool called PRISM. Previously, Shmatikov had analyzed the anonymity properties provided by the Crowds network using PRISM. In this project, we investigated the applicability of the approach used by Shmatikov to other anonymous networks, in particular Onion Routing and Tarzan. An additional network, so-called Adithia, was made to complete the analysis.

As preliminary steps, we first made models of the Dining Cryptographers and the Three Door problems. For the first model, we checked the functional correctness of the protocol by computing some probabilistic properties of it. PRISM automated analysis gave a proof that the protocol functions correctly. As for the second model, we tried to find the best strategy to win the prize by means of probabilistic analysis. From the analysis, it can be concluded that the best strategy to win the prize is to switch the choice to the other door.

Next, we remodeled the Crowds network based on Shmatikov’s model. Furthermore, we extended the approach to model Adithia, Onion Routing, and Tarzan networks. From these models, some anonymity properties were computed by PRISM and after that the results of the computations were discussed.

The overall conclusions that can be taken based on the results are as follows: the anonymity provided by the networks degrades as the crowd size grows as long as the proportion of corrupt members remains constant, and with the increase of the number of path reformulations.

Based on the results, the anonymity provided by Crowds, Adithia, Onion Routing, and Tarzan were compared. The first remark is that the addition of the concept of mimics alone can make networks more vulnerable. Hence, it is important to employ the mimics’ concept not only to limit the choice of forwarder selections but also to exchange cover traffic between mimics. Furthermore, it is necessary to apply layered encryption to make networks more secure. As the second remark, it is important that the initiator only appears at the first location in a path.

The models were made based on one important assumptions, namely the corrupt members can link paths. This assumption is not suitable for Onion Routing and Tarzan networks since both networks employ layered encryption. Furthermore, the cover traffic that is used in Tarzan network is also not modeled. However, even the models do not really represent the actual networks, it can be seen from the results that the approach still can be used to analyze them. More precisely, from the results, the behavior of the networks can be investigated, the vulnerabilities of the networks can be found out, and the comparison between networks can be done.

PRISM as the model-checker tool is really easy to learn. However, it provides limited features. Furthermore, it is also not powerful to make large size models and compute their probabilistic properties. Due to these limitations, the models made do not represent the actual networks precisely. For future work, it is important to use more powerful tools to model and analyze the anonymity networks. Thus more detail behavior of the networks can be included in the models.
8. Concluding remarks
BIBLIOGRAPHY


APPENDIX
A. THE DINING CRYPTOGRAPHERS PROBLEM'S CODE

//The dining cryptographers problem

nondeterministic

const int HEAD = 1;
const int TAIL = 2;
const int NSA = 3;
const int Crypt0 = 0;
const int Crypt1 = 1;
const int Crypt2 = 2;

module cryptographers

choose : bool init true; // The master chooses the payer
choosensa : bool init false; // The master chooses NSA to pay
choosecrypt : bool init false; // The master chooses one of the cryptographers to pay
payer : [0..4] init 4; // The payer of the dinner
throw_coins : bool init false; // State for throwing coins
cont_throw : bool init false; // State for throwing coins
done_throw : bool init false; // State for throwing coins
tc0 : [0..2] init 0; // State for throwing coin 0
tc1 : [0..2] init 0; // State for throwing coin 1
tc2 : [0..2] init 0; // State for throwing coin 2
coin0 : [0..2] init 0; // Coin 0 is head or tail
coin1 : [0..2] init 0; // Coin 1 is head or tail
coin2 : [0..2] init 0; // Coin 2 is head or tail
check_coins : bool init false; // State for cryptographers checking coins
cont_check : bool init false; // State for cryptographers checking coins
cc0 : [0..2] init 0; // State for checking two coins by cryptographer 0
cc1 : [0..2] init 0; // State for checking two coins by cryptographer 1
cc2 : [0..2] init 0; // State for checking two coins by cryptographer 2
countdisagree : [0..3] init 0; // The number of disagree announcements
done : bool init false; // Finish paying
finish : bool init false; // Finish the process

// The master chooses NSA or one of the cryptographers to pay
[] choose -> (choosensa' = true) & (choose' = false);
[] choose -> (choosecrypt' = true) & (choose' = false);

// NSA is paying
[] choosensa -> (payer' = NSA) & (choosensa' = false) & (throw_coins' = true);
A. The Dining Cryptographers Problem’s code

// One of the cryptographers is paying
[] choosecrypt -> (payer=' Crypt0) & (choosecrypt'=false) & (throw_coins'=true);
[] choosecrypt -> (payer=' Crypt1) & (choosecrypt'=false) & (throw_coins'=true);
[] choosecrypt -> (payer=' Crypt2) & (choosecrypt'=false) & (throw_coins'=true);

// Each cryptographer throws his coin
[] throw_coins ->
  ( tc0' = 1 ) & ( tc1' = 1 ) & ( tc2' = 1 ) &
  ( cont_throw'= true ) & ( throw_coins' = false );

[] cont_throw & tc0 = 1 -> 0.5: ( coin0' = HEAD ) & ( tc0' = 2 ) & ( done_throw'= true ) +
  0.5: ( coin0' = TAIL ) & ( tc0' = 2 ) & ( done_throw'= true );
[] cont_throw & tc1 = 1 -> 0.5: ( coin1' = HEAD ) & ( tc1' = 2 ) & ( done_throw'= true ) +
  0.5: ( coin1' = TAIL ) & ( tc1' = 2 ) & ( done_throw'= true );
[] cont_throw & tc2 = 1 -> 0.5: ( coin2' = HEAD ) & ( tc2' = 2 ) & ( done_throw'= true ) +
  0.5: ( coin2' = TAIL ) & ( tc2' = 2 ) & ( done_throw'= true );

[] done_throw & ( tc0 = 2 ) & ( tc1 = 2 ) & ( tc2 = 2 )
  -> ( cont_throw'= false ) & ( done_throw'= false ) & ( check_coins' = true );

// Each cryptographer checks the coins
[] check_coins -> ( cont_check' = true ) & ( cc0' = 1 ) & ( cc1' = 1 ) &
  & ( cc2' = 1 ) & ( check_coins' = false );

[] cont_check & ( cc0 = 1 ) & ( payer = Crypt0 ) & ( coin0 = coin2 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc0' = 2 ) & ( done'= true );
[] cont_check & ( cc0 = 1 ) & ( payer = Crypt0 ) & ( coin0 != coin2 ) ->
  ( countdisagree' = countdisagree ) & ( cc0' = 2 ) & ( done'= true );

[] cont_check & ( cc0 = 1 ) & ( payer != Crypt0 ) & ( coin0 != coin2 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc0' = 2 ) & ( done'= true );
[] cont_check & ( cc0 = 1 ) & ( payer != Crypt0 ) & ( coin0 = coin2 ) ->
  ( countdisagree' = countdisagree ) & ( cc0' = 2 ) & ( done'= true );

[] cont_check & ( cc1 = 1 ) & ( payer = Crypt1 ) & ( coin1 = coin0 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc1' = 2 ) & ( done'= true );
[] cont_check & ( cc1 = 1 ) & ( payer = Crypt1 ) & ( coin1 != coin0 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc1' = 2 ) & ( done'= true );
[] cont_check & ( cc1 = 1 ) & ( payer != Crypt1 ) & ( coin1 != coin0 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc1' = 2 ) & ( done'= true );
[] cont_check & ( cc1 = 1 ) & ( payer != Crypt1 ) & ( coin1 = coin0 ) ->
  ( countdisagree' = countdisagree ) & ( cc1' = 2 ) & ( done'= true );

[] cont_check & ( cc2 = 1 ) & ( payer = Crypt2 ) & ( coin2 = coin1 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc2' = 2 ) & ( done'= true );
[] cont_check & ( cc2 = 1 ) & ( payer = Crypt2 ) & ( coin2 != coin1 ) ->
  ( countdisagree' = countdisagree ) & ( cc2' = 2 ) & ( done'= true );
[] cont_check & ( cc2 = 1 ) & ( payer != Crypt2 ) & ( coin2 = coin1 ) ->
  ( countdisagree' = countdisagree + 1 ) & ( cc2' = 2 ) & ( done'= true );
[] cont_check & ( cc2 = 1 ) & ( payer != Crypt2 ) & ( coin2 != coin1 ) ->
  ( countdisagree' = countdisagree ) & ( cc2' = 2 ) & ( done'= true );
( countdisagree' = countdisagree ) & ( cc2' = 2 ) & (done'= true);

//Finish the process
[] done & (cc0 = 2) & (cc1 = 2) & (cc2 = 2) -> (cont_check' = false) & (done'= false) & (finish' = true);

endmodule

// Properties list

// Pmax=? [true U (countdisagree = 0 | countdisagree = 2) & finish & (payer= NSA)]
// Pmax=? [true U (countdisagree = 1 | countdisagree = 3) & finish & (payer= Crypt0 | payer = Crypt1 | payer = Crypt2)]
// Pmax=? [true U (countdisagree = 0 | countdisagree = 2) & finish & (payer= Crypt0 | payer = Crypt1 | payer = Crypt2)]
// Pmax=? [true U (countdisagree = 1 | countdisagree = 3) & finish & (payer= NSA)]
A. The Dining Cryptographers Problem’s code
B. THE THREE DOOR PROBLEM’S CODE

// Three door problem
nondeterministic
module door

choose : bool init true; // Choose which door
open : bool init false; // The quiz master opens one of remaining doors
choosedoor : [0..3] init 0; // Choosing door i
opendoor1 : bool init false; // The quiz master is opening door 1
opendoor2 : bool init false; // The quiz master is opening door 2
opendoor3 : bool init false; // The quiz master is opening door 3
id : [0..3] init 0;
next : bool init false; // Next step
d1goat : [0..1] init 0; // Only goat is behind door 1
d2goat : [0..1] init 0; // Only goat is behind door 2
d3goat : [0..1] init 0; // Only goat is behind door 3
sos : bool init false; // Switch or stick?
switchtod1 : [0..1] init 0; // Do you want to switch to door1?
switchtod2 : [0..1] init 0; // Do you want to switch to door2?
switchtod3 : [0..1] init 0; // Do you want to switch to door3?
stickd1 : [0..1] init 0; // Do you want to stick with door 1?
stickd2 : [0..1] init 0; // Do you want to stick with door 2?
stickd3 : [0..1] init 0; // Do you want to stick with door 3?
check : bool init false; // Checking what’s behind the door
checkdoor1 : bool init false; // Check what is hidden behind door 1
checkdoor2 : bool init false; // Check what is hidden behind door 2
checkdoor3 : bool init false; // Check what is hidden behind door 3
d1ob : [0..1] init 0; // The last door opened
d2ob : [0..1] init 0; // The last door opened
d3ob : [0..1] init 0; // The last door opened
sw1 : [0..1] init 0; // Continue switching to door 1
sw2 : [0..1] init 0; // Continue switching to door 2
sw3 : [0..1] init 0; // Continue switching to door 3
st1 : [0..1] init 0; // Continue stick with door 1
st2 : [0..1] init 0; // Continue stick with door 2
st3 : [0..1] init 0; // Continue stick with door 3
win : bool init false; // The player is win
lose : bool init false; // The player is lose

// Choose one of the door
[] choose -> 1/3:(choosedoor’ = 1) & (choose’ = false) & (open’ = true) +
    1/3:(choosedoor’ = 2) & (choose’ = false) & (open’ = true) +
    1/3:(choosedoor’ = 3) & (choose’ = false) & (open’ = true);
// The quiz master opens one of the remaining doors
[] (choosedoor = 1) & open -> (opendoor2' = true) & (open' = false);
[] (choosedoor = 1) & open -> (opendoor3' = true) & (open' = false);
[] (choosedoor = 2) & open -> (opendoor1' = true) & (open' = false);
[] (choosedoor = 2) & open -> (opendoor3' = true) & (open' = false);
[] (choosedoor = 3) & open -> (opendoor1' = true) & (open' = false);
[] (choosedoor = 3) & open -> (opendoor2' = true) & (open' = false);

[] opendoor1 -> (id' = 1) & (d1goat' = 1) & (sos' = true);
[] opendoor2 -> (id' = 2) & (d2goat' = 1) & (sos' = true);
[] opendoor3 -> (id' = 3) & (d3goat' = 1) & (sos' = true);

// Switch or stick?
[] (choosedoor = 1) & (id = 2) & sos -> (switchtod3' = 1) & (d2ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 1) & (id = 2) & sos -> (switchtod2' = 1) & (d3ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 1) & (id = 3) & sos -> (switchtod2' = 1) & (d3ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 1) & (id = 3) & sos -> (stickd1' = 1) & (d3ob' = 1) & (check' = true)
  & (sos' = false);

[] (choosedoor = 2) & (id = 1) & sos -> (switchtod3' = 1) & (d1ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 2) & (id = 1) & sos -> (stickd2' = 1) & (d1ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 2) & (id = 3) & sos -> (switchtod1' = 1) & (d3ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 2) & (id = 3) & sos -> (stickd2' = 1) & (d3ob' = 1) & (check' = true)
  & (sos' = false);

[] (choosedoor = 3) & (id = 1) & sos -> (switchtod2' = 1) & (d2ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 3) & (id = 1) & sos -> (switchtod3' = 1) & (d1ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 3) & (id = 2) & sos -> (switchtod1' = 1) & (d2ob' = 1) & (check' = true)
  & (sos' = false);
[] (choosedoor = 3) & (id = 2) & sos -> (stickd3' = 1) & (d2ob' = 1) & (check' = true)
  & (sos' = false);

// The quiz master opens the door of final choice
[] (switchtod3 = 1) & (d2ob = 1) & check -> (checkdoor3' = true) & (sw3' = 1)
  & (check' = false);
[] (stickd1 = 1) & (d2ob = 1) & check -> (checkdoor1' = true) & (st1' = 1)
  & (check' = false);
[] (switchtod2 = 1) & (d3ob = 1) & check -> (checkdoor2' = true) & (sw2' = 1)
  & (check' = false);
[] (stickd1 = 1) & (d3ob = 1) & check -> (checkdoor1' = true) & (st1' = 1)
  & (check' = false);
[] (switchtod3 = 1) & (d1ob = 1) & check -> (checkdoor3' = true) & (sw3' = 1)
  & (check' = false);
[] (stickd2 = 1) & (d1ob = 1) & check -> (checkdoor2' = true) & (st2' = 1)
& (check' = false);
[] (switchtod1 = 1) & (d3ob = 1) & check -> (checkdoor1' = true) & (sw1' = 1) & (check' = false);
[] (st1d2 = 1) & (d3ob = 1) & check -> (checkdoor2' = true) & (st2' = 1) & (check' = false);

[] (switchtod2 = 1) & (d1ob = 1) & check -> (checkdoor2' = true) & (sw2' = 1) & (check' = false);
[] (stickd3 = 1) & (d1ob = 1) & check -> (checkdoor3' = true) & (st3' = 1) & (check' = false);

// Look what's inside the door
[] checkdoor3 & (d2ob = 1) & (sw3 = 1) -> 0.5:(d1goat' = 1) & (win' = true) & (checkdoor3' = false) + 0.5:(d3goat' = 1) & (lose' = true) & (checkdoor3' = false);
[] checkdoor1 & (d2ob = 1) & (st1 = 1) -> 0.5:(d3goat' = 1) & (win' = true) & (checkdoor1' = false) + 0.5:(d1goat' = 1) & (lose' = true) & (checkdoor1' = false);
[] checkdoor2 & (d3ob = 1) & (sw2 = 1) -> 0.5:(d1goat' = 1) & (win' = true) & (checkdoor2' = false) + 0.5:(d2goat' = 1) & (lose' = true) & (checkdoor2' = false);
[] checkdoor1 & (d3ob = 1) & (st1 = 1) -> 0.5:(d2goat' = 1) & (win' = true) & (checkdoor1' = false) + 0.5:(d3goat' = 1) & (lose' = true) & (checkdoor1' = false);
[] checkdoor3 & (d1ob = 1) & (sw3 = 1) -> 0.5:(d2goat' = 1) & (win' = true) & (checkdoor3' = false) + 0.5:(d3goat' = 1) & (lose' = true) & (checkdoor3' = false);
[] checkdoor2 & (d1ob = 1) & (st2 = 1) -> 0.5:(d3goat' = 1) & (win' = true) & (checkdoor2' = false) + 0.5:(d2goat' = 1) & (lose' = true) & (checkdoor2' = false);
[] checkdoor1 & (d3ob = 1) & (sw1 = 1) -> 0.5:(d2goat' = 1) & (win' = true) & (checkdoor1' = false) + 0.5:(d3goat' = 1) & (lose' = true) & (checkdoor1' = false);
[] checkdoor2 & (d3ob = 1) & (st2 = 1) -> 0.5:(d1goat' = 1) & (win' = true) & (checkdoor2' = false) + 0.5:(d2goat' = 1) & (lose' = true) & (checkdoor2' = false);
[] checkdoor2 & (d1ob = 1) & (sw2 = 1) -> 0.5:(d3goat' = 1) & (win' = true) & (checkdoor2' = false) + 0.5:(d2goat' = 1) & (lose' = true) & (checkdoor2' = false);
[] checkdoor3 & (d1ob = 1) & (st3 = 1) -> 0.5:(d2goat' = 1) & (win' = true) & (checkdoor3' = false) + 0.5:(d3goat' = 1) & (lose' = true) & (checkdoor3' = false);
B. The Three Door Problem’s code

```plaintext
(checkdoor3' = false)
+ 0.5:(d3goat' = 1) & (lose' = true) &
  (checkdoor3' = false);

[] checkdoor1 & (d2ob = 1) & (sw1 = 1) -> 0.5:(d3goat' = 1) & (win' = true) &
  (checkdoor1' = false)
+ 0.5:(d1goat' = 1) & (lose' = true) &
  (checkdoor1' = false);

[] checkdoor3 & (d2ob = 1) & (st3 = 1) -> 0.5:(dgoat' = 1) & (win' = true) &
  (checkdoor3' = false)
+ 0.5:(d3goat' = 1) & (lose' = true) &
  (checkdoor3' = false);
```

endmodule

// Properties list

// Pmax=? [ true U switchtod1=1 & win ]
// Pmax=? [ true U switchtod2=1 & win ]
// Pmax=? [ true U switchtod3=1 & win ]
// Pmax=? [ true U stickd1=1 & win ]
// Pmax=? [ true U stickd2=1 & win ]
// Pmax=? [ true U stickd3=1 & win ]
C. CROWDS' CODE

// Crowds network

probabilistic

// Probability of forwarding
const double PF = 0.8;
const double notPF = 0.2;

// Probability that a crowd member is honest or corrupt
const double bmember = 0.167; // or 0.091
const double gmember = 0.833; // or 0.909

const int TotalAns = 3; // The number of path reformulations
const int Crowdsize = 5; // The number of honest members

module crowds

    new : bool init true; // New path construction protocol
    start : bool init false; // The protocol is started to run
    deliver : bool init false; // The request is sent to the end server
    forward : bool init false; // The request is forwarded to other member
    choose : bool init false; // Choose at random the next member on the path
    countrun : [0..TotalAns] init TotalAns; // Number of protocol has been run
    good : bool init false; // Possible honest member chosen
    bad : bool init false; // A corrupt member is chosen
    next : bool init false; // Decide the next step, forward or deliver
    record : [0..4] init 0; // Record member i in the path
    observe : bool init false; // Observe the previous member in the path
    member : [0..4] init 0; // Identity of each member
    observe0 : [0..TotalAns+1] init 0; // # observations done by the attackers to member0
    observe1 : [0..TotalAns+1] init 0; // # observations done by the attackers to member1
    observe2 : [0..TotalAns+1] init 0; // # observations done by the attackers to member2
    observe3 : [0..TotalAns+1] init 0; // # observations done by the attackers to member3
    observe4 : [0..TotalAns+1] init 0; // # observations done by the attackers to member4
    done : bool init false; // Finish one protocol

// A new path is built, it is assumed that member 0 is the initiator
[] new & (countrun >= 1) ->
    (start' = true) & (member' = 0) & (countrun' = countrun - 1) & (new' = false);

// The protocol is started by choosing the next member in the path
[] start -> (choose' = true) & (start' = false);
// There are two possibilities, i.e. the next member is honest or corrupt
[72] choose -> gmember: (good' = true) & (choose' = false) + bmember: (bad' = true) & (choose' = false);

// There are 5 possibilities of the honest members chosen
[72] good -> 1/5 : (member' = 0) & (record' = 0) & (next' = true) & (good' = false) + 1/5 : (member' = 1) & (record' = 1) & (next' = true) & (good' = false) + 1/5 : (member' = 2) & (record' = 2) & (next' = true) & (good' = false) + 1/5 : (member' = 3) & (record' = 3) & (next' = true) & (good' = false) + 1/5 : (member' = 4) & (record' = 4) & (next' = true) & (good' = false);

// The honest member chosen decides whether forward the request or deliver it
[72] next -> PF : (forward' = true) & (next' = false) + notPF : (deliver' = true) & (next' = false);
[72] forward -> (start' = true) & (forward' = false);

// If a corrupt member is chosen, then he observes the previous member in the path
[72] bad -> (observe' = true) & (bad' = false);
[72] observe & (member = 0) & (observe0 < TotalAns) -> (observe0' = observe0 + 1) & (deliver' = true) & (observe' = false);
[72] observe & (member = 1) & (observe1 < TotalAns) -> (observe1' = observe1 + 1) & (deliver' = true) & (observe' = false);
[72] observe & (member = 2) & (observe2 < TotalAns) -> (observe2' = observe2 + 1) & (deliver' = true) & (observe' = false);
[72] observe & (member = 3) & (observe3 < TotalAns) -> (observe3' = observe3 + 1) & (deliver' = true) & (observe' = false);
[72] observe & (member = 4) & (observe4 < TotalAns) -> (observe4' = observe4 + 1) & (deliver' = true) & (observe' = false);

// Finish the protocol
[72] deliver -> (done' = true) & (deliver' = false);
[72] done -> (new' = true) & (done' = false);

endmodule

// Properties list

// P=? [ true U new & countrun=0 & observe0>1 ]
// P=? [ true U new & countrun=0 & observe0<1 & (observe1>1 | observe2>1 | observe3>1 | observe4>1) ]
// P=? [ true U (new & countrun=0 & observe1<1 & observe2<1 & observe3<1 & observe4<1 & observe0<1) ]
D. ADITHIA’S CODE

// Adithia network
// 1 corrupt member, 4 mimics

probabilistic

// Probability of forwarding
const double PF = 0.8;
const double notPF = 0.2;

const int TotalAns = 3; // The number of path reformulations
const int AdithiaSize = 6; // The number of members

formula n0mimics = member = 1,3,4,5,0;
formula n1mimics = member = 0,5,3,2,1;
formula n2mimics = member = 1,5,4,3,2;
formula n3mimics = member = 1,2,0,4,3;
formula n4mimics = member = 0,5,2,3,4;
formula n5mimics = member = 2,0,1,4,5;

module adithia

bad_id : [0..6] init 6; // Dummy variable
bad_guy : [0..AdithiaSize-1] init 0; // Identity of the corrupt member
new : bool init false; // New path construction protocol
countrun : [0..TotalAns] init TotalAns; // Number of protocol has been run
start : bool init false; // The protocol is started to run
choose : bool init false; // Choose at random the next member in the path
member : [0..AdithiaSize-1] init 0; // Identity of each member
check : bool init false; // Mimics checking
path : [0..AdithiaSize-1] init 0; // Member already in the path
check_again : bool init false; // Check whether the member chosen is a corrupt member
next : bool init false; // Decide the next step, forward or deliver
deliver : bool init false; // The request is sent to the end server
forward : bool init false; // The request is forwarded to other member
observe : bool init false; // Observe the previous sender
observe0 : [0..TotalAns+1] init 0; // # observations done by the attackers to member0
observe1 : [0..TotalAns+1] init 0; // # observations done by the attackers to member1
observe2 : [0..TotalAns+1] init 0; // # observations done by the attackers to member2
observe3 : [0..TotalAns+1] init 0; // # observations done by the attackers to member3
observe4 : [0..TotalAns+1] init 0; // # observations done by the attackers to member4
observe5 : [0..TotalAns+1] init 0; // # observations done by the attackers to member5
done : bool init false; // Finish one protocol
D. Adithia’s code

// Preliminary step: assign a corrupt member
[] bad_id = 6 -> 1/5: (bad_guy' = 1) & (new' = true) & (bad_id' = 0) +
1/5: (bad_guy' = 2) & (new' = true) & (bad_id' = 0) +
1/5: (bad_guy' = 3) & (new' = true) & (bad_id' = 0) +
1/5: (bad_guy' = 4) & (new' = true) & (bad_id' = 0) +
1/5: (bad_guy' = 5) & (new' = true) & (bad_id' = 0);

// A new path is built, it is assumed that member 0 is the initiator
[] new & (countrun >= 1) ->
(start' = true) & (member' = 0) & (path' = 0) & (countrun' = countrun - 1) &
(new' = false);

// The protocol is started by choosing the next member in the path
[] start -> (choose' = true) & (start' = false);

// There are 6 possibilities for a member to be chosen
[] choose -> 1/6 : (member' = 0) & (check' = true) & (choose' = false) +
1/6 : (member' = 1) & (check' = true) & (choose' = false) +
1/6 : (member' = 2) & (check' = true) & (choose' = false) +
1/6 : (member' = 3) & (check' = true) & (choose' = false) +
1/6 : (member' = 4) & (check' = true) & (choose' = false) +
1/6 : (member' = 5) & (check' = true) & (choose' = false);

// Checking whether the member chosen is the previous member’s mimic
[] check & (path = 0) & n0mimics -> (check_again' = true) & (check' = false);
[] check & (path = 0) & !n0mimics -> (choose' = true) & (check' = false);

[] check & (path = 1) & n1mimics -> (check_again' = true) & (check' = false);
[] check & (path = 1) & !n1mimics -> (choose' = true) & (check' = false);

[] check & (path = 2) & n2mimics -> (check_again' = true) & (check' = false);
[] check & (path = 2) & !n2mimics -> (choose' = true) & (check' = false);

[] check & (path = 3) & n3mimics -> (check_again' = true) & (check' = false);
[] check & (path = 3) & !n3mimics -> (choose' = true) & (check' = false);

[] check & (path = 4) & n4mimics -> (check_again' = true) & (check' = false);
[] check & (path = 4) & !n4mimics -> (choose' = true) & (check' = false);

[] check & (path = 5) & n5mimics -> (check_again' = true) & (check' = false);
[] check & (path = 5) & !n5mimics -> (choose' = true) & (check' = false);

// Check whether the member chosen is honest or corrupt
[] check_again & (member != bad_guy) -> (next' = true) & (check_again' = false);
[] check_again & (member = bad_guy) -> (observe' = true) & (check_again' = false);

// The honest member chosen decides whether to forward the request or deliver it
[] next -> PF : (forward' = true) & (next' = false) +
notPF : (deliver' = true) & (next' = false);
[] forward -> (start' = true) & (path' = member) & (forward' = false);
// Observations by the corrupt member
[] observe & (path = 0) & (observe0 < TotalAns) ->
  (observe0' = observe0 + 1) & (deliver' = true) & (observe' = false);
[] observe & (path = 1) & (observe1 < TotalAns) ->
  (observe1' = observe1 + 1) & (deliver' = true) & (observe' = false);
[] observe & (path = 2) & (observe2 < TotalAns) ->
  (observe2' = observe2 + 1) & (deliver' = true) & (observe' = false);
[] observe & (path = 3) & (observe3 < TotalAns) ->
  (observe3' = observe3 + 1) & (deliver' = true) & (observe' = false);
[] observe & (path = 4) & (observe4 < TotalAns) ->
  (observe4' = observe4 + 1) & (deliver' = true) & (observe' = false);
[] observe & (path = 5) & (observe5 < TotalAns) ->
  (observe5' = observe5 + 1) & (deliver' = true) & (observe' = false);

// Finish the protocol
[] deliver -> (done' = true) & (deliver' = false);
[] done -> (new' = true) & (done' = false);

e ndmodule

// Properties list

// P=? [ true U new & countrun=0 & observe0>1 ]
// P=? [ true U new & countrun=0 & observe0<=1 &
//   (observe1>1 | observe2>1 | observe3>1 | observe4>1 | observe5>1) ]
// P=? [ true U (new & countrun=0 & observe1<=1 &
//   observe2<=1 & observe3<=1 & observe4<=1 & observe5<=1 & observe0>1) ]
D. Adithia’s code
E. ONION ROUTING’S CODE

// Onion Routing network
probabilistic

// Probability of extending the path
const double PE = 0.8;
const double notPE = 0.2;

// Probability that a crowd member is bad
const double bmember = 0.167; // or 0.091
const double gmember = 0.833; // or 0.909

const int TotalAns = 6; // The number path reformulations
const int ORsize = 5; // The number of honest members

module onion

  new : bool init true; // New path construction protocol
  start : bool init false; // The protocol is started to run
  send : bool init false; // Start sending the message through the path
  extend : bool init false; // The request is forwarded to other member
  choose : bool init false; // Choose at random the next member in the path
  countrun : [0..TotalAns] init TotalAns; // Number of protocol has been run
  good : bool init false; // Possible honest member chosen
  bad : bool init false; // A corrupt member is chosen
  put : bool init false; // Put the member in the path
  or_path : [0..ORsize-1] init 0; // Put onion router i in the path
  put_bad : bool init false; // Put the corrupt member in the path
  next : bool init false; // Decide the next step, extend or terminate the path
  send_id : [0..2] init 0;
  continue : bool init false; // Continue building the path
  observe : bool init false; // Observe the previous sender
  finish : bool init false; // Finish sending the message
  member : [0..ORsize-1] init 0; // Identity of each member
  observe0 : [0..TotalAns+1] init 0; // # observations done by the attackers to member0
  observe1 : [0..TotalAns+1] init 0; // # observations done by the attackers to member1
  observe2 : [0..TotalAns+1] init 0; // # observations done by the attackers to member2
  observe3 : [0..TotalAns+1] init 0; // # observations done by the attackers to member3
  observe4 : [0..TotalAns+1] init 0; // # observations done by the attackers to member4
  done : bool init false; // Finish one protocol

// A new path is built, it is assumed that member 0 is the initiator
[] new & (countrun >= 1) ->
(start' = true) & (member' = 0) & (or_path'= 0) & (countrun' = countrun - 1) & (new' = false);

// The protocol is started by choosing the next member in the path
[] start -> (choose' = true) & (start' = false);

// There are two possibilities, i.e. the next member is honest or corrupt
[] choose -> gmember: (good' = true) & (choose' = false) +
          bmember: (bad' = true) & (choose' = false);

// There 4 possibilities of the honest members chosen
[] good -> 1/4 : (member' = 1) & (put' = true) & (good' = false) +
        1/4 : (member' = 2) & (put' = true) & (good' = false) +
        1/4 : (member' = 3) & (put' = true) & (good' = false) +
        1/4 : (member' = 4) & (put' = true) & (good' = false);

// Put the honest member chosen in the path
[] put -> (or_path'= member) & (next'= true) & (put'= false);

// The initiator decides whether to extend or terminate the path
[] next -> PE : (extend' = true) & (next' = false) +
       notPE : (send' = true) & (send_id'= 1) & (next' = false);
[] extend -> (start' = true) & (extend' = false);

// If a corrupt member is chosen, then the initiator put the corrupt member in the path
[] bad -> (put_bad' = true) & (bad' = false);

// It is assumed that the path construction is continued
[] put_bad -> (continue'= true) & (put_bad'= false);
[] continue -> (send'= true) & (send_id'= 2) & (continue'= false);

// The message is sent through the path
[] send & (send_id = 1) -> (finish'= true) & (send'= false);
[] send & (send_id = 2) -> (observe'= true) & (send'= false);

// The corrupt member observe the member appears before him
[] observe & (or_path = 0) & (observe0 < TotalAns) ->
  (observe0' = observe0 + 1) & (finish' = true) & (observe' = false);
[] observe & (or_path = 1) & (observe1 < TotalAns) ->
  (observe1' = observe1 + 1) & (finish' = true) & (observe' = false);
[] observe & (or_path = 2) & (observe2 < TotalAns) ->
  (observe2' = observe2 + 1) & (finish' = true) & (observe' = false);
[] observe & (or_path = 3) & (observe3 < TotalAns) ->
  (observe3' = observe3 + 1) & (finish' = true) & (observe' = false);
[] observe & (or_path = 4) & (observe4 < TotalAns) ->
  (observe4' = observe4 + 1) & (finish' = true) & (observe' = false);

// Finish the protocol
[] finish -> (done' = true) & (finish' = false);
[] done -> (new' = true) & (send_id'= 0) & (done' = false);
endmodule
// Properties list

// P=? [ true U new & countrun=0 & observe0>1 ]
// P=? [ true U new & countrun=0 & observe0<=1 &
// (observe1>1 | observe2>1 | observe3>1 | observe4>1) ]
// P=? [ true U (new & countrun=0 & observe1<=1 &
// observe2<=1 & observe3<=1 & observe4<=1 & observe0>1) ]
F. TARZAN’S CODE

// Tarzan network
// 1 corrupt member, 4 mimics

probabilistic

// Probability of extending the path
const double PE = 0.8;
const double notPE = 0.2;

const int TotalAns = 3; // The number of path reformulations
const int Tarzan_size = 6; // The number of members

formula n0mimics = member = 1,3,5,0;
formula n1mimics = member = 0,5,3,2,1;
formula n2mimics = member = 1,5,4,3,2;
formula n3mimics = member = 1,2,0,4,3;
formula n4mimics = member = 0,5,2,3,4;
formula n5mimics = member = 2,0,1,4,5;

module adithia

bad_id : [0..6] init 6; // Dummy variable
bad_guy : [0..Tarzan_size-1] init 0; // Identity of the corrupt member
new : bool init false; // New path construction protocol
countrun : [0..TotalAns] init TotalAns; // Number of protocol has been run
start : bool init false; // The protocol is started to run
choose : bool init false; // Choose at random the next member in the path
member : [0..Tarzan_size-1] init 0; // Identity of each member
check : bool init false; // Mimics checking
path : [0..Tarzan_size-1] init 0; // Member already in the path
check_again : bool init false; // Checking whether the member chosen is the corrupt member
put : bool init false; // Put the honest member in the path
put_bad : bool init false; // Put the corrupt member in the path
next : bool init false; // Decide the next step, forward or deliver
send_id : [0..2] init 0; // Sending identity
continue : bool init false; // Continue building the path
send : bool init false; // The request is sent through the path
extend : bool init false; // The path is extended
observe : bool init false; // Observe the previous sender
observe0 : [0..TotalAns+1] init 0; // # observations done by the attackers to member0
observe1 : [0..TotalAns+1] init 0; // # observations done by the attackers to member1
observe2 : [0..TotalAns+1] init 0; // # observations done by the attackers to member2
observe3 : [0..TotalAns+1] init 0; // # observations done by the attackers to member3
observe4 : [0..TotalAns+1] init 0; // # observations done by the attackers to member4
observe5 : [0..TotalAns+1] init 0; // # observations done by the attackers to member5
finish : bool init false; // Finish sending the message
done : bool init false; // Finish one protocol

// Preliminary step: assign a corrupt member
[] bad_id = 6 -> 1/5: (bad_guy'= 1) & (new'= true) & (bad_id'= 0) +
1/5: (bad_guy'= 2) & (new'= true) & (bad_id'= 0) +
1/5: (bad_guy'= 3) & (new'= true) & (bad_id'= 0) +
1/5: (bad_guy'= 4) & (new'= true) & (bad_id'= 0) +
1/5: (bad_guy'= 5) & (new'= true) & (bad_id'= 0);

// A new path is built, it is assumed that member 0 is the initiator
[] new & (countrun >= 1) ->
(start' = true) & (member' = 0) & (path' = 0) & (countrun' = countrun - 1) &
(new' = false);

// The protocol is started by choosing the next member in the path
[] start -> (choose' = true) & (start' = false);

// There 6 possibilities for a member to be chosen
[] choose -> 1/5 : (member' = 1) & (check' = true) & (choose' = false) +
1/5 : (member' = 2) & (check' = true) & (choose' = false) +
1/5 : (member' = 3) & (check' = true) & (choose' = false) +
1/5 : (member' = 4) & (check' = true) & (choose' = false) +
1/5 : (member' = 5) & (check' = true) & (choose' = false);

// Check whether the member chosen is the previous member’s mimic
[] check & (path = 0) & n0mimics -> (check_again' = true) & (check' = false);  
[] check & (path = 0) & !n0mimics -> (choose' = true) & (check' = false);

[] check & (path = 1) & n1mimics -> (check_again' = true) & (check' = false);  
[] check & (path = 1) & !n1mimics -> (choose' = true) & (check' = false);

[] check & (path = 2) & n2mimics -> (check_again' = true) & (check' = false);  
[] check & (path = 2) & !n2mimics -> (choose' = true) & (check' = false);

[] check & (path = 3) & n3mimics -> (check_again' = true) & (check' = false);  
[] check & (path = 3) & !n3mimics -> (choose' = true) & (check' = false);

[] check & (path = 4) & n4mimics -> (check_again' = true) & (check' = false);  
[] check & (path = 4) & !n4mimics -> (choose' = true) & (check' = false);

[] check & (path = 5) & n5mimics -> (check_again' = true) & (check' = false);  
[] check & (path = 5) & !n5mimics -> (choose' = true) & (check' = false);

// Check whether the member chosen is honest or corrupt
[] check_again & (member != bad_guy) -> (put' = true) & (check_again'= false);  
[] check_again & (member = bad_guy) -> (put_bad' = true) & (check_again'= false);

// Put the honest member chosen in the path
[] put -> (path'= member) & (next'= true) & (put'= false);

// The initiator decides whether to extend or terminate the path
[] next -> PE : (extend' = true) & (next' = false) +
notPE : (send' = true) & (send_id'= 1) & (next' = false);

// If a corrupt member is chosen, it is assumed that the path construction is continued
put_bad -> (continue'= true) & (put_bad'= false);
continue -> (send'= true) & (send_id'= 2) & (continue'= false);

// The message is sent through the path
send & (send_id = 1) -> (finish'= true) & (send'= false);
send & (send_id = 2) -> (observe'= true) & (send'= false);

// Observations by the corrupt member
observe & (path = 0) & (observe0 < TotalAns) ->
(observe0' = observe0 + 1) & (finish' = true) & (observe' = false);
observe & (path = 1) & (observe1 < TotalAns) ->
(observe1' = observe1 + 1) & (finish' = true) & (observe' = false);
observe & (path = 2) & (observe2 < TotalAns) ->
(observe2' = observe2 + 1) & (finish' = true) & (observe' = false);
observe & (path = 3) & (observe3 < TotalAns) ->
(observe3' = observe3 + 1) & (finish' = true) & (observe' = false);
observe & (path = 4) & (observe4 < TotalAns) ->
(observe4' = observe4 + 1) & (finish' = true) & (observe' = false);
observe & (path = 5) & (observe5 < TotalAns) ->
(observe5' = observe5 + 1) & (finish' = true) & (observe' = false);

// Finish the protocol
finish -> (done' = true) & (finish' = false);
done -> (new' = true) & (send_id'= 0) & (done' = false);

endmodule

// Properties list

// P=? [ true U new & countrun=0 & observe0>1 ]
// P=? [ true U new & countrun=0 & observe0<=1 &
  (observe1>1 | observe2>1 | observe3>1 | observe4>1 | observe5>1) ]
// P=? [ true U (new & countrun=0 & observe1<=1 &
  observe2<=1 & observe3<=1 & observe4<=1 & observe5<=1 & observe0>1) ]