P. Van Den Hurk

THE COST OF COMPARING SECRETS WITHOUT LEAKING INFORMATION

MASTER THESIS

P.vandenHurk@student.ru.nl

Supervisor:	Dr. P. van Rossum
	Dr. W. Teepe
Co-referent:	Dr. J. H. Hoepman
Thesis number:	618

P. Van Den Hurk

THE COST OF COMPARING SECRETS WITHOUT LEAKING INFORMATION

MASTER THESIS

Nijmegen,
November 23, 2009
Institute for Computing and Information Science (ISIS)
Radboud University Nijmegen
Toernooiveld 1
6525 ED Nijmegen

Author:	P. Van Den Hurk
E-mail address:	P.vandenHurk@student.ru.nl
Course:	Computer Science
Variant:	Security
Supervisor:	Dr. P. van Rossum
	Dr. W. Teepe
Co-referent:	Dr. J. H. Hoepman
Thesis number:	618

ABSTRACT

Suppose two parties, Alice and Bob, each hold a set of private values. Private set matching privately computes the intersection of these two sets without leaking any other information (about the set elements that do not belong to the intersection). This thesis studies the communication complexity of a certain private set matching algorithm [Tee06] , which was designed to be highly efficient.

We prove mathematically, that the expected communication complexity of the algorithm is at most 3 bits per set element. This implies that for small random subsets of a large domain, the communication complexity is on average a constant number of bits per element.

During our research we also discovered that this protocol does not provide all the required security properties for private set matching. In the malicious attacker model [DY83], the algorithm assumes the sets have high min-entropy, which implies that every element in the set has a low chance of occurring in a set. Would this not be the case, the adversary is able to construct a set with a the values that occur more often in sets. Furthermore the fairness principle, which states that both parties learn the intersection of their sets is not guaranteed. In the case of the honest-but-curious attacker model however, the protocol does not suffer from these flaws.

ACKNOWLEDGEMENTS

I would like to thank Wouter Teepe who was willing to supervise me. When I was interested in another topic he let me try to find anything within that topic to base my research on. Unfortunately that didn't work out, after which he suggested this topic and provided good guidance. Of course, I would like to thank Peter van Rossum as well, who was willing to study up on the subject when Wouter was no longer able to supervise me. His help with the math and writing was of great use. I would also like to thank Emile Strijbos, who has taken the time to read through the thesis several times, eradicating mistakes and inconsistencies where he could find them.

For everybody I didn't mention explicitly, let me silently thank family, fellow students, teachers and friends, who have made this education possible, enjoyable and enriching.

CONTENTS

1 INTRODUCTION

Suppose two parties, Alice and Bob, each hold a set of private values. Private set matching privately computes the intersection of these two sets without leaking any other information (about the set elements that do not belong to the intersection). Private set matching can be used to compare databases for instance. For security purposes, these protocols are build in such a way that the communication doesn't require a trusted third party, ttp, or any other participant for that matter. This ensures that the participants themselves have total control over what information is shared, which is generally limited to references to information instead of the information itself. The information being compared, is considered to be a number of discrete sets of pieces of information, i.e. each set only contains unique elements. Private set matching is discussed in more details in chapter 2.

In the dissertation by Wouter Teepe [Tee06], such a private set matching protocol is described which provides the participants of the protocol the actual intersection without leaking the information itself. It is important to note here that all participants will learn the same information, which is the intersection in private set matching.

Simulations have shown that only three bits per piece of information are communicated in the best case scenario. Unfortunately, these simulations are very costly to perform. In their configuration there was:

- A maximum of 1000 pieces of information per party,
- a two party setup [Tee08; p. 163-165]
- and the intersection was empty.

Since these simulations only indicate a low communication complexity of 3 bits per element, this thesis will present a proof for the low communication complexity in chapter 5.

During the writing of this thesis, we noticed that the algorithm doesn't completely protect the data which is outside the intersection. Therefore a brief security analysis is applied to the algorithm as well. The analysis will assume a two party setup in both an honest-but-curious (HBC) and malicious attacker model (Dolev-Yao) [DY83]. The analysis will show why the algorithm is secure in an honest-but-curious model and why it isn't in the malicious model. The security analysis is incorporated in chapters 2, 3 and 4.

2 PRIVATE SET MATCHING

Private set matching has already been explained as the operation of determining the intersection of two datasets without conveying the actual information. This specific form of operation is viewed from three perspectives: A functional, security and complexity perspective. This chapter introduces these perspectives and they will be considered in more detail in later chapters.

2.1 FUNCTIONAL MODELS

2.1.1 IDEAL WORLD MODEL

The principle of private set matching is easy to understand with an example. Assume we have a certain number of people who want to surprise each other on movie night. They each have a list of movies they own themselves but they want to bring movies which at least one other person doesn't have.
In order to do this, they each send their respective list to somebody trustworthy and otherwise uninvolved. This trusted person then compares the information and sends the list of the movies they all have in common back to each of them.

In this example, none of the friends learn about the movies they don't all share and everybody learns the intersection and the intersection alone.

2.1.2 REAL WORLD MODEL

In private set matching algorithms, trust is generally not placed in a third party since it is one other party which has access to the information being compared. These algorithms will only communicate information possession proofs which are compared in a manner that all and only the participating parties will learn the same result, which is the intersection derived from references. It is also required that the original information cannot be reconstructed from the proof alone and that the proof cannot be constructed without the original information.

The ideal world model can be considered an ideal model for private set matching. The distinction between the ideal and real model is made since in the malicious attacker model, not every malicious attack is considered to be the responsibility of the algorithm. If an attack using the algorithm (the real model) can be executed on the ideal model as well, it is a problem of the model and not the algorithm. Moreover, one could say that the responsibility of the ideal world model's trusted third party now lies with the real world model and its participants.

2.2 SECURITY MODEL

Earlier, the attacker models [DY83] were mentioned which can be applied to this example as well.

Honest-but-curious attacker: The honest-but-curious party does not deviate from the protocol rules. In our case this means his input is based on what he knows and not what messages he can construct. The HBC party does try to infer more information from the information communicated than intended.

Malicious insider: The malicious insider is able to construct any message and will try to abuse a protocol in order to learn more than he is allowed. He is also capable of being a HBC attacker.

Malicious outsider: The malicious outsider is not a participant of the protocol. The malicious is not able to construct messages that are encrypted and thus must rely on attacks like disrupting, replaying, relaying and delaying messages.

We assume the communication between the parties to be secured in a way that only the participating parties can decrypt the communicated information. This means that there are no malicious outsiders.

2.2.1 SECURITY CONTEXT

In this paper we assume that the protocol participants are either good, HBC or malicious. There is always one good participant. Since an intersection is interesting for any attacked the security context assumes the possibility of one.

The participants their sets have a common large domain Ω such that any set $S \subset \Omega$. If the set S has a low entropy, i.e. the range of possible or highly likely elements is small, then the security context differs much from a high entropy situation:

Low entropy: There is a high probability for certain elements to be part of a set S.

Some elements are more likely to be part of any participants set than others. Knowledge of this likelihood can be obtained from either the domain's context, like knowing how names are formed, or knowing for example which people are likely clients. The ideal world model accepts input which is deemed a well formed proof of possession, the real world model should therefore make sure that it is possible to make a secure proof for elements with a high likelihood. If anybody is able to construct either the elements or their respective proofs used in the algorithm, e.g. the contents of the domain are known, not even the ideal model can protect the elements from leaking out.

High entropy: There is a low probability for certain elements to be part of set S.

In this situation, it is assumed that no elements have a heightened likelihood of occurring in any one set S. If the size of the domain Ω has the same strength of a descent hash, the domain itself may be known for it is unfeasible to construct references for every element.

It was stated that it should be impossible to construct the references or proofs over information without owning that information. It is possible to reconstruct names from hashes, while hashing the name along with the date of birth makes it a lot harder to reconstruct information from the hash. Enforcing this input requirement ensures that the information in a low entropy is still safe.

2.2.2 SECURITY PROPERTIES

Confidentiality:

Certain information is only accessible to the person authorized to view it:

- <u>The amount of secrets shared</u>
- <u>The secrets shared</u>

The amount of shared secrets is not protected by the considered algorithm and it doesn't pretend to protect this kind of information.

Fairness principle:

No protocol participant can learn information whilst preventing any other from learning the same.

Privacy:

For any information $x \in S$, only the result of a proof construction function $f(x)$ is communicated for which it is unfeasible to construct the information x from $f(x)$ alone.

2.3 COMMUNICATION COMPLEXITY

First and foremost, the security is the most important aspect of the algorithm that needs to be properly implemented. After security the complexity becomes the most important thing to consider. There are two kinds of complexity:

- Computational complexity
- Communication complexity

The computational complexity is not considered. Not because it is not important, but simply because this thesis focuses on the communication complexity. The communication complexity is usually calculated for the average case scenario, which is sometimes hard to calculate.

In the case of private set matching there are some major implementation independent influences:

- The number of information references/proofs inputted in the algorithm
- The ratio of the independent sets sizes
- The number of secrets shared
- The number of participants

What kind of scenario is used, can only be explained when the inner working of the protocol are explained, A couple of important variables are set as follows:

- The intersection is empty (best case scenario);
- There is a two participant setup;
- The sets of the two participants are of equal size (worst case scenario)

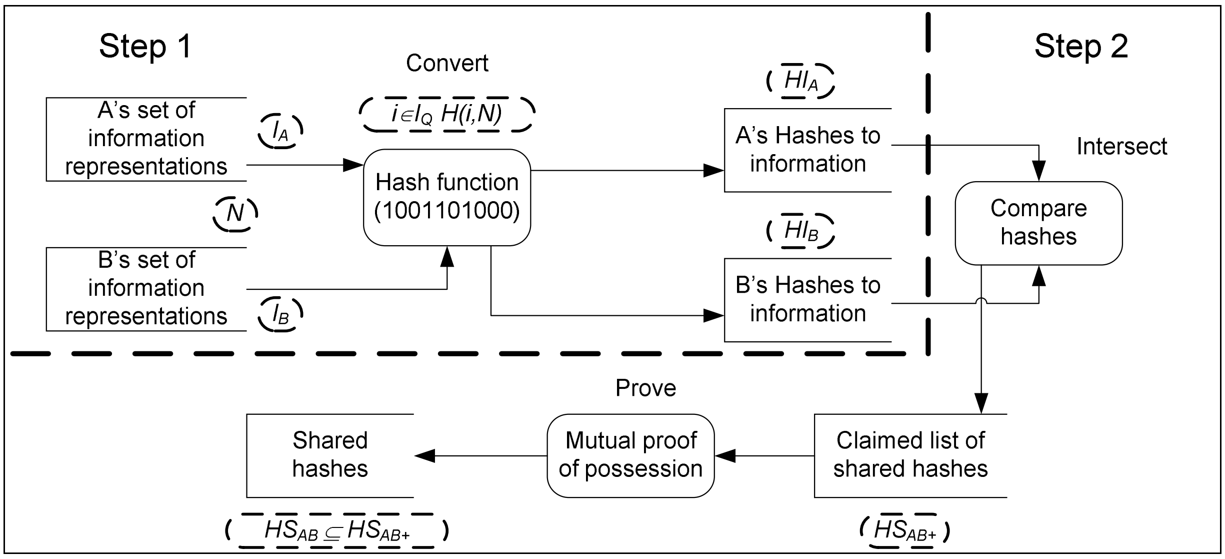
What will become clear in the next chapter, is that the empty intersection has a very big influence on the communication complexity.

3 INTERNAL MODEL OF THE ALGORITHM

The private set algorithm to be considered will henceforward be referred to as T-2 and this chapter will explain how it works. It will start off with step 1, describing the preparation each participant has to perform. The chapter then continues with step 2, touching the subject of communication, whilst leaving the details to be explained in the chapter 4. Finally some general security remarks are made about the information input.

3.1 GENERAL

The general idea of the T-2 is simple and schematically displayed in Figure 1. The algorithm starts in step 1, where it converts the discrete sets into hashes. In step 2, the algorithm then communicates part of this hashed information to mutually determine the intersection. A proving protocol is then used to prove mutual possession of the information instead of possession of the references. The two steps are explained in Figure 1 below and serves as a reference for the variable notation used in the rest of this thesis.



Step 1
Convert
Step 2
A's set of information representations
I_A
$i \in I_Q\ H(i,N)$
HI_A
A's Hashes to information
Intersect
Hash function (1001101000)
N
Compare hashes
HI_B
B's set of information representations
B's Hashes to information
I_B
Prove
Shared hashes
Mutual proof of possession
Claimed list of shared hashes
$HS_{AB} \subseteq HS_{AB+}$
HS_{AB+}

Figure 1 *- The simple representation of T-2 augmented by a division of general steps. The steps indicate a shift from internal computation to computation by communication. The open rectangles represent data, the rounded boxes are operations and the dashed rounded boxes contain variable expressions for the data.*

3.2 STEP 1 - CONVERTING THE INFORMATION INTO HASHES

This offline individual step simply prepares information for the communication in step 2, by securing and hashing everything.

1. **Generate uniform set of information references**
   Each party will have their respective set of information. The sets of information they are comparing will have certain features in common in order to compare them. In the case of names, one could decide that the full first name and last name without middle names is sufficient to compare names. When comparing files for example, the file-hash is sufficient.

   Result:

   I_Q = Set of references to information
   A,B = An identity of a participating party
   Q = Placeholder for any identity, such as A or B

2. **Convert - Generate hashes based upon reference**
   As soon as every party has made their respective list of information references, each party generates actual hashes for each reference. In order to make sure the hashes are only usable for the participants of a single instance of the protocol, the references are cryptographically hashed along with some shared secret number, a nonce *N*. The nonce is assumed to be secure (random for this execution and not feasibly guessable) and only known by the participating parties. The parties are responsible for generating that nonce. By hashing the information reference along with only the nonce, only a party member in a particular algorithm execution can link the hash to the original information.

   Result:

   $HI_Q = \{ \mathrm{H}(i,N) \mid i \in I_Q \}$
   h_1 = A hash which is an element of some HI_Q
   N = Secret Nonce, shared between A and B

3.3 STEP 2 - THE COMMUNICATION ASPECT OF THE PROTOCOL

This step will generate the list of shared information hashes of which mutual possession is proven; i.e. HS_{AB}. The communication cost of the algorithm is based solely on this step.

3. **Intersect - Compare the hashes by using a communication protocol**
   This sub protocol produces the acclaimed intersection of information references, HS_{AB+}. Acclaimed, since some information references might seem like shared information, where they are random hashes which are not related to any information. The plus sign indicates these possible extra references. The details of the intersection sub protocol will be covered in section 4.2.

   Result:

   $HS_{AB+} \subseteq HI_A \cup HI_B$

4. **Prove - Per hash, simultaneously prove possession of related information**
   Using the individually calculated HS_{AB+}, the possession of each piece of information will be proven by each party. The result is a list of references of which all parties have the corresponding information and the mutual knowledge of this fact. The details of the proving sub protocol will be covered in section 4.2 as well.

   Result:

   $HS_{AB} = HI_A \cap HI_B \subseteq HS_{AB+}$

3.4 SECURITY ASPECTS

For both the offline preparation and the actual private set matching using protocols, there are some things to be said about the security.

The set elements are converted into hashes. Because of the nonce *N,* these hashes are only valid for the duration of the execution of the algorithm. How the nonce *N* is generated or exchanged exactly is not within the scope of this thesis, but it is very unlikely that two executions should ever end up with the same nonce and no participant should be feasibly able to influence the protocol into using a previously used nonce.

The hashing function $HI_Q=\mathrm{H}(I_Q,N)$ should have the following properties.

- **Preimage resistance**
  It should be hard to find an I_Q for a given HI_Q and N where $HI_Q=\mathrm{H}(I_Q, N)$
- **Second preimage resistance**
  It is hard to find an I given I_Q and $\mathrm{H}(I, N)=\mathrm{hash}(I_Q, N)$.

Collision resistance is not of importance here since the algorithm does not require participants to sign their data, which is the main function for collision resistance. Finding an I_{Q2} for a given $H(I_Q, N)$ such that $H(I_Q, N) = H(I_{Q2}, N)$ would not render any knowledge about I_Q.

Using this information, the reference generation can be considered a random oracle model and prevents anybody from inferring information from the hash alone.
For the offline part, we already discussed entropy requirements of input information which therefore falls outside of this scope. Whether a malicious party constructs his set of information is also not relevant.

The security of step 2, the protocols themselves, will be discussed in the next chapter.

3.5 CONCLUSION

It is important to understand what the purpose of this algorithms usage is. T-2 can be used to determine which information is shared between parties without giving out any other information and without using any third party. The chapter introduced a set of variables which will be used later on in the document. The distinction between HS_{AB} and HS_{AB+} is not clear yet for instance, why a reference found in the intersection protocol may not be a correct reference can only be explained by explaining the protocol.

Some security has been discussed, but as of yet only the information input has any influence on the protocol.

4 COMMUNICATING THE INTERSECTION

The offline step constructed the hashes from the participants sets. These hashes are going to be used by the protocols to infer the intersection. This step requires the hashes to be put into the binary tree data structure, which will be explained.

The end goal being the communication complexity, this chapter provides a couple of lemma's based upon the specific encoding of the two protocols. But before one can understand the encoding and it's lemma's, the functionality of both will be covered first.

Since this chapter explains the inner workings of the protocols, we are now able to say something about the security from a white box perspective and whether the protocols are able to remain secure according to our requirements.

4.1 THE BINARY TREE DATASTRUCTURE

The hashes, each consisting of a series of bits, are fitted into a data structure called a binary tree of which the intersection sub protocol makes use. For the reader who already understands what a binary tree is, please skip to the paragraph 4.1.2 where the effects of the binary tree on the communication complexity is discussed.

The binary tree will discussed using graph theory. A tree is a special case of a graph, which can be seen in Figure 2.1. A graph consists of nodes (vertices) and lines (edges). Vertices are often depicted as circles with or without a value. Edges are depicted by lines, which are either a simple line or an arrow; an arrow would indicate direction. Edges are used to indicate some sort of relation between two vertices, in the case of a tree it simply is a connection. Let's say a tree $\boldsymbol{T}$ consists of a set of Vertices $\boldsymbol{V}$ and a set of Edges $\boldsymbol{E}$. We denote the tree as $T_X = (V,E)$, where X represents the identity of the tree.

The vertices V are all connected to each other by edges, either directly or by means of a path. A path consists of vertices which are all connected by $e=v$-1 lines. Apart from the first and the last vertex, each vertex has two connecting edges. An example of a path is indicated by the light gray vertices in Figure 2.1.

A tree is free of cycles. A cycle being a path where the first and last node of a path are connected by a single edge, as indicated by the thicker lines in Figure 2.1. In a tree, there is no $V_S \subseteq V$ *and* $E_s \subseteq E$ such that (V_S,E_S) forms a cycle. For every $v \in V$ there is at least one corresponding $e \in E$ edge and every $e \in E$ links exactly two corresponding $v_1,v_2 \in V$ vertices, where $v_1 \neq v_2$.

Any vertex can be called the root of the tree, but only one is chosen. Every vertex with only one connected edge and is not the root, is called a leaf. In the case of a binary tree, every vertex has at most two outgoing edges. Figure 2.2 and Figure 2.3 are different representations of the same binary tree.

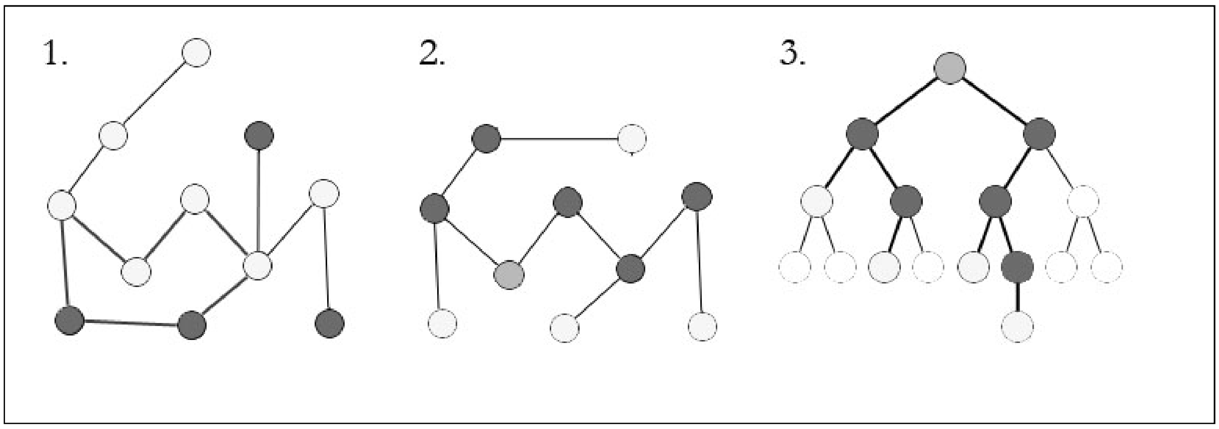


Figure 2 *- Different graphs. 1) The light gray vertices form a path and the thick edges form a cycle. 2) An actual binary tree with light gray vertices as leaves. A random node has been chosen as root and has been colored medium gray. 3) A more common representation of a tree for the tree in 2 with empty white vertices to indicate how a full four level binary tree would look like.*

4.1.1 BINARY TREE REPRESENTATION OF HASHES

In order to quickly compare two sets of hashes, the binary tree graph is used in [Tee06] to represent an individual set of hashes. A hash can be represented by nothing more than a string of bits, in other words, a binary string; e.g. "11001011100010011111". Each bit in this binary string, can be considered a subsequent choice between the values '0' and '1' in the binary tree as is made visible in Figure 3 below.

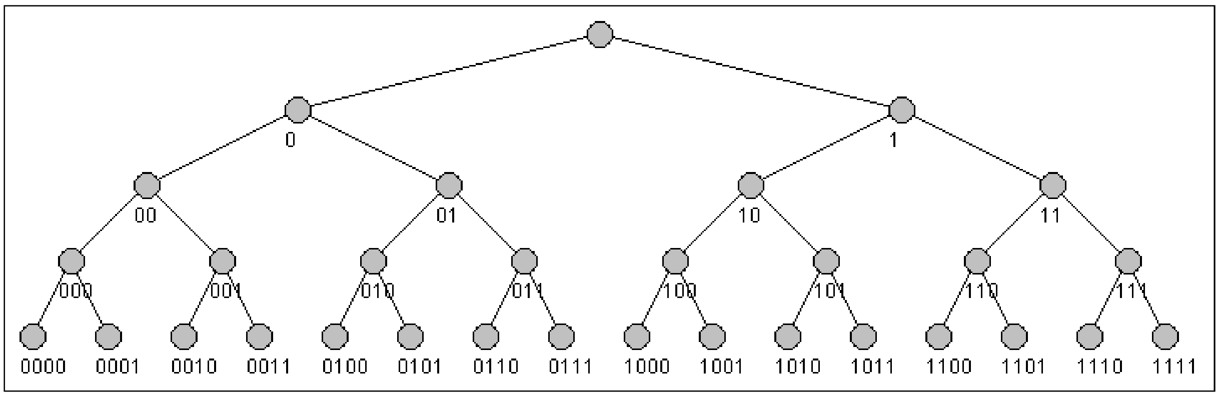


Figure 3 *- A five level deep binary tree representing each binary number from 0 to 15.*

4.1.2 COMPARING HASHES USING THE TREE

If the amount of hashes inserted into the tree is big enough, e.g. 100, not only is it highly likely that every node in level five in Figure 3 is in use, but it's also as good as certain that multiple hashes will start off with the same first four bits. The percentage of nodes used on a tree level, disregarding whether or not there is a parent, will be called the **tree level coverage**. The hashes which use a certain node are called the node's **related hashes**.

If two of these trees would be compared, they would actually be comparing the prefixes to a greater number of hashes. If such a node in one tree happens to have a large amount of related hashes and in the other tree it does not, then there is no point in comparing any of the child nodes for that node. In that situation, all the related hashes in the first tree would become irrelevant for further comparing. This makes comparing very cheap when the binary tree is used.

All parties, in our case (A)lice and (B)ob, have generated their respective personal trees T_A and T_B in the first step. Which they will use as input for the first sub protocol, the intersection protocol.

4.2 DETERMINING INTERSECTION

This paragraph's goal is to describe the workings of the two sub protocols, give a security analysis and prepare some lemma's for determining the communication complexity.

4.2.1 NOTATION INTRODUCTION

Some new notations are introduced in Table 1 below:

NOTATION	***MEANING***
$\|X\|$	The number of elements in X. If X represents a string, each character is considered an element. In all other cases, X is simply a collection of elements.
ε, h, p and c	Binary strings consisting of zero or more concatenated bits. ε A representation for the empty binary string. h A complete hash which is written like h_1 or h_2. The hash h_1 is a reference to information both parties understand and h_{2Q} is a reference which can only be reconstructed by one party using the original information, the nonce N and a challenge c from the other party. This serves as proof of possession for one of the parties. p A binary string which is written like p_1 or p_2 which respectively is the prefix of h_1 and h_2. c A challenge, which both parties use to construct h_2, but is constructed by one of the parties.

Table 1 *– Some definitions of statements used throughout the thesis.*

4.2.2 PROTOCOL DESCRIPTIONS

The first protocol simply compares references by first comparing all the first bits of every reference in S_1 to the first bits of every reference in S_2. This step is repeated for each bit index in $[1..|h_1|]$. This is done by viewing and treating each set of references as a binary hash tree. By comparing the identical tree nodes of the different sets we can determine if one of the two nodes has no related hashes by which rule no subsequent node (or prefix) should be considered in either tree. When the algorithm reaches the last index, the remaining nodes indicate the intersection.

The second protocol assumes that by a small margin of error some hashes may falsely coincide. This margin of error is caused by the algorithm itself but can also be forced by any participant that claims to have all hashes. One can imagine the complexity of the algorithm rising very fast if random hashes are used in the algorithm.

Nevertheless, each element in the supposed intersection HS_{AB+} is used as input in the second protocol. The parties mutually prove possession here by sending each other challenges C_A and C_B for each element in HS_{AB+}, such that $|C_A|=|C_B|=|HS_{AB+}|$. The response for each $h_1 \in HS_{AB+}$ is calculated with the help of the other party's challenge $c \in C_Q$, the original $i \in I$ for which $h_1=\mathrm{H}(i, N)$ holds, the nonce N and their own identity q using the formula: $h_2 = \mathrm{H}(i,N,q,c)$, each participant forms a set of responses for themselves (their own proofs) and the other (the other's proofs) to be able to answer and verify requests. For each of the responses all the bits of each index are send until the last index has been transmitted. During this exchange each participant is able to see whether the other one was lying about possession and will start sending nonsense as soon as the first mistake is found.

4.2.3 SECURITY

At this point enough is known about the protocols to make a security analysis. The protocol was designed to be robust against the malicious party and to be very efficient at communicating the intersection well.

Confidentiality:
The amount of secrets shared is secured against either the HBC or malicious participant. The algorithm allows participants to input random hashes into their set for set size obscurity, but the hashes should be of such a length that it is very unlikely for hashes to match by accident for any set size. The hashes that do match are very likely to be in the intersection.

It is however allowed to put every hash possible into his or her set, since the algorithm allows this, this applies to the HBC participant as well. The second protocol then makes sure that only those references for which both parties have proven their possession are in the intersection. This does obscure the amount of actual references exchanged, but it is very unlikely that one party is able to request most of the possible reference combinations since the hashes are so large and this would be very costly to perform and the behavior of sending many references is easily noticed.

The secrets behind the references remain protected as long as the information abides by the input requirements. Nobody should be able to reconstruct $i \in I_Q$ from $h_1 = H(i,N)$ where N and h_1 are known.

Fairness principle:
The fairness principle requires that no protocol participant is able to learn information whilst preventing any other from learning the same. Both protocols leak information at each iteration by design. It might be true that at most, the references are the only information learned from the other participant and that is only possible if one participant decides to include a small amount of fake references while the other participant puts most of the possible references in his set. If two parties don't include a large amount of the possible references in their sets, they alternate sending one bit more than the other. Both the HBC and the malicious participants are able to break this.

There is another way to break the fairness principle, this is only applicable to the malicious party since it requires breaking the protocols rules. Both sub protocols state that a reference is considered to be part of the intersection if the complete reference or proof has been communicated. However, in a normal execution of the algorithm where the amount of hashes in the sets is far below the maximum amount of hashes, it becomes apparent very fast whether the hashes are going to end up in the intersection. Let's say after three rounds of no change in the amount of possible prefixes where all prefixes refer to unique references, the malicious participant is able to stop sending information about references he does have. Hereby he is able to infer with a high likelihood that the particular reference is part of the other participant's information set, but since the honest participant doesn't try to infer such information, the malicious participant does learn more than the other.

Privacy:
For any information $x \in I_Q$, only the result of a function $f(x)$ is communicated for which it is unfeasible to construct x from $f(x)$ alone. Property holds in both the attacker models.

The algorithm fails to be security and thereby isn't a decent private set matching algorithm. What the algorithm does do, is provide participants with an intersection which has been proven to be part of the intersection. The intersection determined by the algorithm can therefore still be a subset of the real intersection of the inputs in case there is a malicious party involved. This is still useful when there is more related information to be shared between the parties.

In the case of the HBC participant, the intersection at the end of the execution with be equal to the actual intersection of the inputs. The references the HBC participant is able to learn don't provide him with the opportunity to reconstruct the original information. However, by storing the secret nonce N used to generate h_1 and computing an h_1 over every new information combined with N, the malicious participant is able to see which information was shared by another party in the past.

4.2.4 PROTOCOL RULES

The sub protocols described below are described to such an extent that it is possible to infer the amount of data communicated, which will be covered in chapter 5.

4.2.4.1 Intersection sub protocol

The intersection sub protocol uses only two types of messages. An *ask* message for claiming possession and at the same time asking for a reply. A *refuse* message to deny possession and to stop any further communication regarding possession of information as visible in Table 2.

MESSAGE	*MEANING*
ask(p_1)	A party asks the other party to look whether he has any references to information that have p_1 as its prefix and at the same time, the party states that he is in possession of p_1. "I claim to have hashes with p_1 as prefix, so do you have any hashes with p_1 as a prefix?"
refuse(p_1)	A party tells the other party he does not claim possession of any references to information that have p_1 as its prefix "I claim not to have any hashes with p1 as prefix, so ignore your hashes that do have p_1 as a prefix."

Table 2 *- Basic messages used in the intersection sub protocol. The prefixes p_x can be of any length between 0 and $|h_x|$.*

The protocol starts by party A sending *ask*(ε), which means nothing more than: "I have hashes, do you?". Whenever the receiving party receives an *ask*(p_1) message, he is required to reply unless $|h_1|=|p_1|$. There must be a reply for both p_1+'0' and p_1+'1'. Depending on whether the receiver has corresponding hashes for the prefix p_1+x, he replies with either an *ask*(p_1+x) or a refuse(p_1+x) message.

A simple general example, adapted from [Tee06], is displayed in Table 3 below. Without knowing either set, it is clear that only references with 1011 as a prefix might still be in the intersection.

Step	*Party*	p_1	***MESSAGES***	***MEANING***
1	A		*ask(ε)*	Start of the protocol
2	B	ε	*ask(0)*	I've got references with prefix 0
			ask(1)	I've got references with prefix 1
3	A	0	*refuse(00)*	No references with the prefix 00; stop asking
			ask(01)	I've got references with prefix 01
		1	*ask(10)*	I've got references with prefix 10
			refuse(11)	No references with the prefix 11; stop asking
4	B	01	*refuse(010)*	No references with the prefix 010; stop asking
			refuse(011)	No references with the prefix 011; stop asking
		10	*refuse(100)*	No references with the prefix 100; stop asking
			ask(101)	I've got references with prefix 101
5	A	101	*ask(1010)*	I've got references with prefix 1010
			refuse(1011)	No references with the prefix 1011; stop asking

Table 3 *- A basic execution of the protocol for determining the intersection of two parties their hash sets. In this particular execution there is an immediate response to every ask message, i.e. the ask/refuse responses are not delayed to another step. The p_1 messages represent the new prefix which followed from the other party's.*

The protocol executions data can be visualized in Figure 4. Both parties take turns in sending the next bit for each prefix still in the running. Each node here can be encoded by one bit since each answer is expected, the only question is whether the reply is an *ask* or *refuse* message. An *ask* message will always require a reply and a *refuse* message never requires a reply; with the exception that the *ask* message doesn't require a reply when $|p|=|h_1|$. Finally, since each message is expected, we also know which node in the tree is considered and therefore what prefix it represents.

PARTY A (HI_A)	***PARTY B (HI_B)***
0101	0011
0111	<u>1010</u>
<u>1010</u>	1011

Table 4 *– The two small sets of parties A and B.*

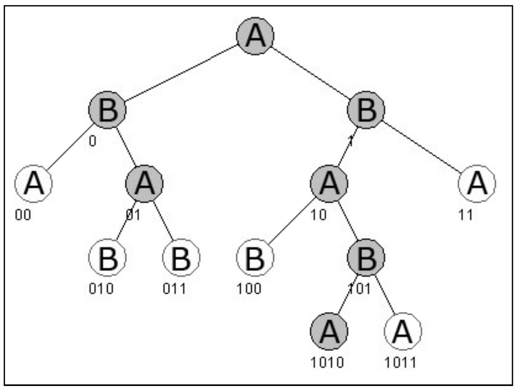


Figure 4 – *A visual representation of a protocol execution. The letter signifies the party sending a message and the color whether the message was an ask or refuse message*

The ask message can be represented by a '1' and a refuse message by a '0'. If a batch message is represented by Q{s} where Q is the identity and 's' is a binary string with the answers for a level in the intersection tree, then the example execution with a communication cost of 13 bits can be represented by:

A{1}, B{11}, A{0110}, B{0001} and A{10}

This communication cost consists of the amount of *refuse* and *ask* messages. However, some of the ask messages are not part of the actual intersection. Since A has the reference 0100 and B has the reference 0000, A's reply A{0$\underline{1}$10} to B{$\underline{1}$1} contains an ask message which doesn't correspond to the actual intersection (prefix 01) and requires two more messages then when B would have replied. These type of extra messages are called spurious nodes '***S***'. The intersection tree without these spurious nodes is called $\boldsymbol{T_{AB}}$. The communication cost is therefore $|T_{AB}|+|S|$. The details of the situations and overall communication cost of these spurious nodes are covered in section 5.4.1.

4.2.4.2 Proving sub protocol

Both parties have now obtained HS_{AB+} using the previous sub protocol. An overview for the variables used in this section is adapted from [Tee06]:

VAR	*MEANING*
$\boldsymbol{h_1}$	The hash value that denotes the secret for which the sub protocol has been invoked, $h_1 = H(i \in I_Q, N)$
$\boldsymbol{C_A}$	The challenge chosen by Alice. Effectively it's a special number generated by Alice for a single h_1 to be proven by another party
$\boldsymbol{C_B}$	Analogously Bob's challenge
$\boldsymbol{h_{2A}}$	The hash value that constitutes Alice her proof, $h_{2A} = H(i \in I_A, N, A, C_B)$
$\boldsymbol{h_{2B}}$	The hash value that constitutes Bob's proof, $h_{2B} = H(i \in I_B, N, B, C_A)$
$\boldsymbol{p_2}$	A prefix of h_{2Q} which at most will have a length of $\lvert h_{2Q} \rvert$

The last sub protocol also uses two messages to perform its actions. A *challenge* message to provide a challenge C_Q, which is a unique random number, used to construct a proof h_{2Q}. And a *prove* message to communicate prefixes p_2 for the proof h_{2Q}.

MESSAGE	*MEANING*
challenge(h_1, C_Q)	A principal asks the other party to prove possession of a secret with hash value h_1, using challenge C_Q "Use the challenge C_Q to construct your proof h_{2Q} for possessing the information related to h_1"
prove(h_1, p_2)	A principal partially proves possession of the secret, whose hash reference is by h_1, by presenting the prefix p_2, which is a prefix for the proof h_{2Q}. "Using p_2, I provide a part of my proof that I possess the information related to h_1"

Table 5 *- Basic messages used in the T-2 algorithm.*

No party can construct h_{2Q} unless the party has the information reference $i \in I_Q$ corresponding to the hash reference h_1 and the nonce N. The nonce N by itself is no longer useful to construct a proof of possession. And since the protocol should be robust against eavesdropping, a repeat attack from a malicious party must prevented; extensive details about the security properties of this protocol can be found in Teepe's dissertation [Tee06]. The combination of the identity Q of the other party and the challenge C_Q is chosen to ensure that the party Q can ensure the protocol doesn't use a challenge a second time and to ensure that another pair of parties with the same dataset and nonce N don't accidentally create the same hash. Simply put, if a party can produce h_{2Q} he has proven possession of the related $i \in I_Q$.

The protocol starts by sending all the *challenge* messages C_Q in lexicographical order of $h_1 \in HS_{AB+}$. Here every C_Q has a fixed binary string length and the challenges can therefore be concatenated without any marker or index. Subsequently, every *prove* message is sent in the same order as the challenges but a bit at a time.

Since it is known what prefix p_2 and which new bit is expected, the h_1 is not sent for the *prove* messages and only the new bits for p_2 are send until $p_2 = h_{2Q}$.

PARTY	H_1	***MSG***	***MSG DECODED***
A	1	0100	*Challenge(1 , 0100)*
B	1	0010	*Challenge(1 , 0010)*
A	2	0001	*Challenge(2 , 0001)*
B	2	0100	*Challenge(2 , 0100)*
A	1	11	*Prove(1 , 11)*
B	1	11	*Prove(1 , 11)*
A	2	01	*Prove(2 , 01)*
B	2	01	*Prove(2 , 01)*
A	1	01	*Prove(1 , 01)*
B	1	00	*Prove(1 , 00);* FALSE
A	2	11	** random bits **

Table 6 *- Example run of the protocol with an erroneous message from party B.*

4.3 CONCLUSION

The intersection protocol uses *ask* and *refuse* messages. In the configuration used, both players are honest, send all and only the information they began with and abide by the algorithms rules. In the end $T_{AB}+S$ is produced, which contains the intersection HS_{AB+} as well as the amount of bits communicated.

The prove protocol uses *challenge* and *prove* messages. First all challenges are send by all parties; both send one for each h_1 in HS_{AB+} . Then the proof prefixes are sent by each party until all proofs are complete.

The algorithm itself is not secure and cannot be considered a decent private matching algorithm since the fairness principle has been breached. It is possible for a participant to learn more about the intersection than another. What the protocol does provide, is a means to establish an intersection about which parties are both certain that the information related to the intersection was in possession of both participants before the execution.

This concludes the section that is required to understand the next chapter.
Previous chapters explained:

- What is the purpose of the algorithm;
- What kind of input it requires;
- The inner workings of the protocol and its configuration;

The next chapter will actually calculate the number of bits per hash in the best case scenario, in which the intersection is empty.

5 CALCULATING THE BEST CASE COMMUNICATION COST

The calculations for the average communication cost can be outlined in a scheme, as depicted in Figure 5. The scheme roughly follows the same lines as the algorithm itself.

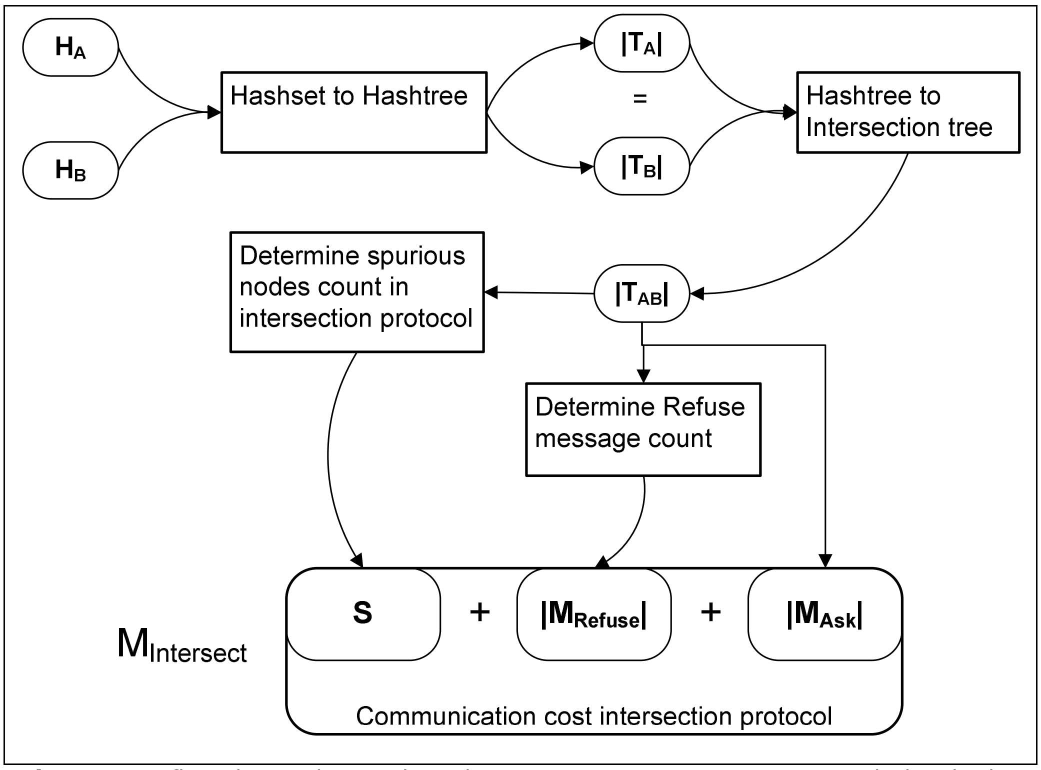
As we said before, this analysis assumes:

1. A two party setup (Parties A and B);
2. an empty intersection ($H_A \cap H_B = \varnothing$);
3. equally sized input sets ($|H_A| = |H_B|$);

By determining how the binary trees are filled with random hashes, we are able to determine the size of a hash tree for any depth by the number of hashes inputted alone. The size of a tree is determined by the number of nodes in them and since $|H_A| = |H_B|$, only one tree size needs to be determined.

The intersection protocol's cost can be derived from the mathematical intersection tree of the two input trees $T_{AB} = T_A \cap T_B$. By analyzing this mathematical intersection and taking spurious nodes into account , we are able to determine how many *ask* and *refuse* messages the sub protocol would send.

This brief explanation results in an average communication cost calculation which can be calculated by the input size alone.



H_A
H_B
Hashset to Hashtree
$|T_A|$
=
$|T_B|$
Hashtree to Intersection tree
Determine spurious nodes count in intersection protocol
$|T_{AB}|$
Determine Refuse message count
$M_{Intersect}$
S
+
$|M_{Refuse}|$
+
$|M_{Ask}|$
Communication cost intersection protocol

Figure 5 *- A flow chart indicating how the average communication cost is calculated. The rectangular shapes loosely indicate the type of calculation being performed on its input. The resulting rounding rectangular merely gives an idea and a handle on what has been calculated; as will become clear.*

An empty intersection of hashes is assumed because that situation is of most interest here. Note that an empty intersection of hashsets differs from the intersection of the trees. The mathematical intersection tree still intersects on a lot of hash prefixes, but since an empty hashset intersection is assumed, every intersecting node in the tree is random. Furthermore since the hashsets do not intersect, the prove sub protocol has no influence on the calculated communication cost. Nevertheless, the formula to calculate the communication cost for the proving protocol in case the intersection is not empty is still described. Even though we have stated the intersection of hashes to be empty and assuming the hashes have been chosen randomly, there is a very small chance that two hashes are the same. Considering that this chance is so small and therewith the negligible effect of such an occurrence on the communication cost, it will not be taken into account from here on.

The actual structure of this chapter is as follows. In order to prove that the communication cost is below 3 bits per inputted hash, the limit hypothesized by Wouter Teepe. Section 5.1 introduces formulas and lemma's based on the protocol properties described in chapter 4, which are used to calculate the communication cost later on. In sections 5.2, the binary tree data structure is examined in relation to the individual processing of hashes. Section 5.3 will then use those individual trees and look at how the intersection is constructed using the protocol. Section 5.3.3 will cover what problems would be faced when the shared hashes would be taken into account. Then in Section 5.3.5, using the protocol rules and the binary trees, a proof is provided that states that the average communication cost is in linear relation to the amount of hashes inputted, effectively this cost will be shown to be one node per inputted hash. Section 5.4.1 will deal with the spurious nodes. Finally, using calculations, Sections 5.5 and 5.6 try to construct the same average communication cost as Teepe's simulations have produced which will result in 2,446.

5.1 SUB PROTOCOL RELATED FORMULAS

By analyzing the algorithm using a top down approach, we are able to understand the problem by incrementally defining more detailed sub problems. In this section lemma's are defined which add the number of messages up so the total amount of messages is reached.

The lemma's require some new notations:

C_X	The cost for all X messages X∈ {Total, Prove, Intersect, Challenge, ProveMsg}
M_X	The set of all X messages X∈ { Ask, Refuse}

Using these notations, lemma's are used to outline what the average communication cost of the algorithm is. The first lemma states that the average communication cost of the algorithm is equal to the cost of the two distinct sub protocols.

Lemma 5.1.a
The total communication cost of the algorithm: $C_{Total} = C_{Intersect} + C_{Prove}$

Proof of Lemma 5.1.a
The cost of the algorithm is calculated by the amount of bits sent. Furthermore, the intersection protocol's messages are known to be one bit long each. The message size of the proving sub protocol is not constant, but as mentioned before $C_{Prove} = 0$. ■

Lemma 5.1.b
$C_{Intersect} = |M_{Ask}| + |M_{Refuse}|$

Proof of Lemma 5.1.b
The cost for the intersection protocol can be determined by the amount of messages since each message cost is exactly one bit. ■

Lemma 5.1.c
$C_{Prove} = C_{Challenge} + C_{ProveMsg} = 0$

Proof of Lemma 5.1.c
The cost for the proving protocol can be determined by adding up the cost of each of the two types of messages. ■

5.1.1 INTERSECTION PROTOCOL COST

The amount of ask messages used in Lemma 5.1.b, can be determined using lemma.

Lemma 5.1.d
$|M_{Ask}| = |T_{AB*}|$

Proof of Lemma 5.1.d
Every *ask* message is a claim of possession by the sender of the message, therefore the intersection tree T_{AB*} represents the message count for the ask messages. The intersection protocol builds a tree by communication, its size is determined by:

1. The behavior of random hashes partially overlapping;
2. The protocol related overhead (spurious nodes).

The first is a determent for mathematical intersection T_{AB}. The second is a determent for the communicated intersection T_{AB*}. The star in T_{AB*} represents the overhead caused by the spurious nodes, therefore we can state $T_{AB} \subseteq T_{AB*}$. The spurious nodes will be examined further on but are determined by the shape of T_{AB} caused by the random behavior of distribution hashes. ■

Lemma 5.1.e
$|M_{Refuse}| = (|M_{Ask}|+1) - (2 \cdot |HS_{AB+}|)$

Proof of Lemma 5.1.e
In layman terms, this lemma states that for every *ask* message a *refuse* message is sent, plus one. But since the shared hashes (which are not present) use up the length of the protocol, no *refuse* messages need to be send at the end.

It is easy to see that for every $h_1 \in HS_{AB+}$, there is no need to send *refuse* messages when $|p_1|=|h_1|$ holds for the p_1 in *ask*(p_1). The protocol then assumes there is no further communication. However, the intersection is empty and therefore $|HS_{AB+}|=0$.

The two answers to every *ask*-node have only three possible situations, as depicted in Figure 6 in which the dark circles represent *ask* messages and the white circles represent *refuse* messages:

1. Two more *ask* messages are sent.
2. One *ask* message and one *refuse* message is sent.
3. Two *refuse* messages are sent.

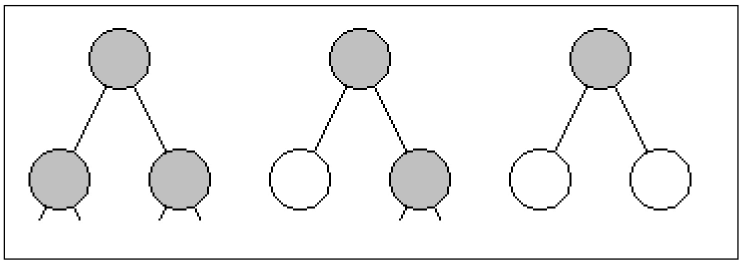


***Figure 6** - The dark nodes represent ask() messages which always end up with two follow-up questions, they can either be an ask() or a refuse() message. The first possibility is two ask() messages, the second one refuse() and one ask() message and the third are two refuse() messages.*

Every *refuse* message will ensure that no further communication will concern prefixes which have the *refuse* message as a prefix. The *ask* messages however can recursively allow further communication as described above. In order to prove that $|M_{Refuse}|=|M_{Ask}|+1$ when the communication has stopped because of the refuse messages,
a right linear context-free-grammar M_1 is constructed as shown in Table 7 below [Sud06]. This grammar produces the language L_1 and will be used to prove the "$(|T_{AB}| + 1)$" part of Lemma 5.1.e; i.e. the situation where $|HS_{AB+}|=0$;

Every grammar rule explains a part of how the tree is allowed look like. To understand the grammar, consider every ‘O’ as an ask node for which the type of the two children are still unknown. After the arrow follows new information, three choices in fact, where each lower case character is an immutable indication of the presence of either an ask or a refuse message. Any uppercase character represents a placeholder for any option after the corresponding arrow in the rules. Every ‘o’ is a representation of an *ask* message communicated and every ‘x’ is a representation for a communicated *refuse* message. This grammar shows that the tree can only end with *refuse* messages and it starts with an *ask* message. An example execution of the grammar is displayed in Table 8.

S’	→	oO
O	→	ooOO | oOX | XX
X	→	x

Table 7 *- Context-Free Grammer for the amount of nodes in the intersection tree T_{AB}. The ‘O’ rule depicts three situations: (1) Two ask message replies, (2) One ask and one refuse message reply and (3) two refuse message replies.*

GRAMMAR INPUT	*GRAMMAR RULES*
S’	S’ → oO
o**O**	O → ooOO
ooo**O**O	O → ooOO
ooooo**O**OO	O → ooOO
ooooooo**O**OOO	O → oOX
oooooooo**O**X**OOO**	O → XX ; 4 times
oooooooo**XXXXXXXXX**	X → x ; 9 times
ooooooooxxxxxxxxx	No. ’o’ = 8 ∧ No. ’x’ = 9

Table 8 *– The aforementioned grammar is applied on the input until all end states have been reached. The rule is applied for every* ***bold*** *part. The result of each rule is made visible using underline. The resulting word contains 8 ‘o’ characters and 9 ‘x’ characters.*

To prove that every ‘o’ is matched with a ‘x’ plus one, an invariant is constructed.

Lemma 5.1.f

For every $m \in (\Sigma(M_1))^*$ if $S' \rightarrow^{+} m$ then $'o'(m) + 1 = ('X'(m) + 'x'(m)) + 2 \cdot 'O'(m)$

Where $\Sigma(M_1)$ is the alphabet of the grammar M_1 (={S’, O, X, o, x}), $\rightarrow^{+}$ indicates a number of consecutive deductions equal or higher than 1 and ‘ω’(m) denotes the number of occurrences of ‘ω’ in m.

Proof of Lemma 5.1.f

Given

$m \in (\Sigma(M_1))^*$

Assume

$S' \rightarrow^{+} m$

Thus, there exists an *x* such that $S' \rightarrow^{x} m$. We will prove $S_1(m) = 'o'(m) + 1 = ('X'(m) + 'x'(m)) + 2 \cdot 'O'(m)$ by induction on *x*.

Base

Assume x=1 and $S' \rightarrow^{1} oO$, thus m=oO therefore $S_1(m)$ holds.

Induction

Assume x>1. Given m' such that $S' \rightarrow^{x-1} m' \rightarrow m$, by induction $S_1(m')$ holds.

We do a case distinction on the rules of grammar M_1 that generate m from m’.

Case (S' → oO)
From x>1 and the absence of S' in other rules follows that this rule has no effect on this statement.

Case (O → ooOO)
From $S_1(m')$ follows
$$'o'(m')\underline{+2} +1 = ('X'(m') + 'x'(m')) + 2\cdot('O'(m')\underline{+1}).$$
Therefore $'O'(m) \leq 'o'(m)$, thus $S_1(m)$.

Case (O → oOX)
From $S_1(m')$ follows
$$'o'(m')\underline{+1} +1 = ('X'(m')\underline{+1} + 'x'(m')) + 2\cdot'O'(m').$$
Therefore $'O'(m) \leq 'o'(m)$, thus $S_1(m)$.

Case (O → XX)
From $S_1(m')$ follows
$$'o'(m') +1 = ('X'(m')\underline{+2} + 'x'(m')) + 2\cdot('O'(m')\underline{-1}).$$
Therefore $'O'(m) \leq 'o'(m)$, thus $S_1(m)$.

Case (X → x)
From $S_1(m')$ follows
$$'o'(m') +1 = ('X'(m')\underline{-1} + 'x'(m')\underline{+1}) + 2\cdot'O'(m').$$
Therefore $'O'(m) \leq 'o'(m)$, thus $S_1(m)$. ■

Since for every $m \in L_1$, $m \in \Sigma\{o,x\}^*$ holds and Lemma 5.1.f holds, we can conclude that $|'o'|+1 = |'x'|$. Together with the fact that the intersection is empty, this proves that $|M_{Refuse}|=|M_{Ask}|+1$. ■

5.1.2 PROVE PROTOCOL COST

The prove sub protocol has HS_{AB+} as input and is only executed when $|HS_{AB+}|>0$. Even though we stated that the calculations do not take shared hashes into account, something can be said about the communication cost of the sub protocol.

Proof of Lemma 5.1.c

The actual cost of the proving sub protocol can be calculated

$$C_{Prove} = |M_{Challenge}| + |M_{ProveMsg}|$$
$$C_{Prove} = 2 \cdot |HS_{AB+}| \cdot |C_Q| + 2 \cdot |HS_{AB+}| \cdot |h_{2Q}|$$
$$C_{Prove} = 2 \cdot |HS_{AB+}| \cdot (|C_Q| + |h_{2Q}|)$$

As stated before, the configuration assumes two participating parties, which is where the two comes from in the formula. The two indicates that both players send the same amount of information. Each player must provide a challenge C_Q for every reference h_I and a proof h_{2Q} as well. ■

5.1.3 COST OF BOTH PROTOCOLS

Having provided the formulas for calculating the communication cost for both sub protocols. The total cost of the algorithm can be calculated.

Lemma 5.1.g
$C_{Total} = 2 \cdot |T_{AB*}| + 1$

Proof of Lemma 5.1.g
Using the previously mentioned lemmas and assuming $|HS_{AB+}|=0$ since it is highly unlikely that two distinct information references share the same hash, the final formula becomes.

$$C_{Total} = |M_{Intersect}| + 2 \cdot |HS_{AB+}| \cdot (|C_Q| + |h_{2Q}|)$$

Substituting $|M_{Intersect}|$

$$C_{Total} = |M_{Ask}| + |M_{Refuse}| + 2 \cdot |HS_{AB+}| \cdot (|C_Q| + |h_{2Q}|)$$

Substituting $|M_{Ask}|$ and $|M_{Refuse}|$

$$C_{Total} = |T_{AB*}| + ((|T_{AB*}| + 1) - (2 \cdot |HS_{AB+}|)) + 2 \cdot |HS_{AB+}| \cdot (|C_Q| + |h_{2Q}|)$$

Since $|HS_{AB+}|$ is as good as 0

$$C_{Total} = 2 \cdot |T_{AB*}| + 1$$ ■

5.2 DETERMINE THE SIZE OF THE INDIVIDUAL HASH TREES

The amount of messages send by the intersection protocol is determined by the mathematical intersection T_{AB} and the overhead caused by spurious nodes. This section will provide a mathematical function to calculate an individual hash tree's size $|T_X|$ with which we will be able to do the same for $|T_{AB}|$ in section 5.3.

The problem of determining the size of the individual tree is tackled by understanding that we can and need to look at each level individually (5.2.1). We will find that the average size of each level can then be determined using a function called G_1, which will be introduced in section 5.2.2.

5.2.1 LOOKING AT THE DISTRIBUTION TREE AS A WHOLE

Every node in the tree that represents a prefix for more than one hash, still only counts as one node. The total amount of nodes is the only thing of interest. If a perfect equal distribution over the tree was a given, i.e. hashes coincide as little as possible, the largest amount of nodes would be used. Say you have 256 hashes to distribute, two unlikely situations are if the hashes:

- **Are evenly distributed**
  Each node on the 8th level ($2^8=256$) would then represent exactly 1 one hash. The next level would still contain 256 nodes, but the tree level coverage would be 50% instead of 100%.
- **Have the largest shared prefix possible while still being unique**
  All the hashes would be the same, except for the very last part. In this situation the entire tree would be the size of a single hash length, plus some nodes to differentiate between the unique hashes.

The problem at hand is obviously a distribution problem.

The randomness of hashes sometimes create situations in which a few hashes share quite a big prefix. In order to find out at what level the level coverage stops being a full 100%, I started simulating the distribution of a large amount of nodes over each level. Looking at the standard deviation for each level, it showed the head of the jelly fish as the part where the standard deviation was zero. Then for the part where the tentacles partially overlap, the standard deviation increases until there is a very small chance for hash prefixes to overlap and it's back to zero again. In Figure 7 you can clearly see the head and the tentacles at the point where they don't overlap.

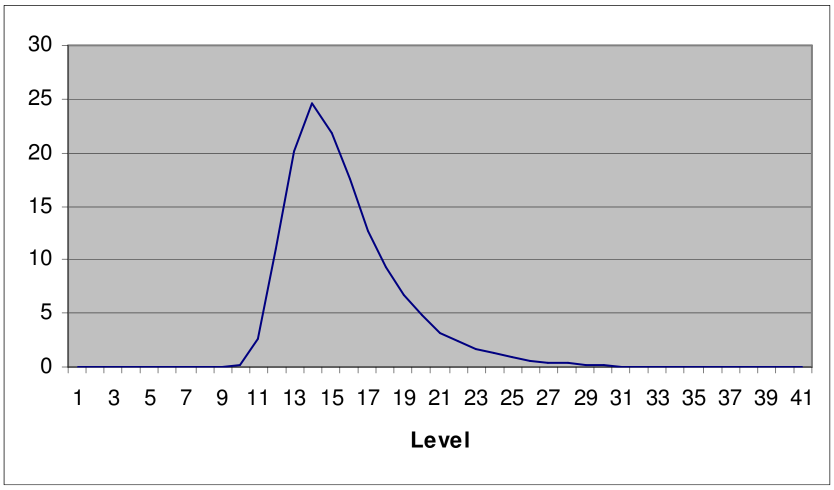


Figure 7 *- Standard deviation for the node count per level x in the individual tree. The amount of hashes was 5000 with 1000 runs. At level nine, a few runs ended with one less occupied node. At the next level there was not one run in which the entire level was occupied. From that moment on, a fast increasing amount of hashes will exclusively occupy unique nodes from that point until every next level will contain exactly 5000 nodes.*

Every prefix of a hash is random within the domain of that prefix. If some hash's prefix is equal to '1', there are 2^1 different possibilities for that prefix and a chance of 2^{-1} that the prefix became '1' instead of

‘0’. For a prefix with the length of two, there isn’t any knowledge about the amount of hashes which had ‘0’ or ‘1’ as a prefix. Since those numbers are random, the amount of hashes per prefix of the length two is just a s random. Therefore, every level in the tree can be viewed separately from the other in order to predict the level’s partition of nodes.

5.2.2 LOOKING AT OUR PROBLEM PER LEVEL

If an individual tree level is looked at as a set of *n* bins, then each bin represents a node. And *x* balls (hashes) are thrown in random bins, which can contain more than one ball; with replacement.
Within this definition, the number of bins with one or more balls are of interest. The function which produces this number is called $G_1(x,n)$.

Lemma 5.2.a

$$G_1(1,n) = 1$$

$$G_1(x,n) = G_1(x-1,n) + 1 - \frac{G_1(x-1,n)}{n}$$

Proof of Lemma 5.2.a

Let *C* and *H* be the sets of the bins and the balls respectively. And let *P(c, h)* be defined as the predicate function determining if the ball $h \in H$ is thrown into bin $c \in C$.

The average number of containers containing a node is then defined by:

$$G_1(|H|,\ |C|) = \left|\left\{\, c \mid c \in C \wedge \exists_{h\in H}\big(P(c,h)\big) \right\}\right|$$

BALLS (x)	*BINS (4)*	*OCCUPIED BINS*
1	**1** 0 0 0	1
2	**1 1** 0 0	2
	2 0 0 0	1
3	**1 1 1** 0	3
	2 1 0 0	2
	3 0 0 0	1

Table 9 *- This table displays the different type of outcomes when adding nodes to a set of containers.*

As can be seen in Table 9, $G_1(1,4)$, is obviously one when assigning one hash. With the second hash, the number of occupied bins can essentially be either one or two. The chance however, that one bin is occupied isn’t the same as the chance that two bins are occupied. To be more precise, the new expected amount of occupied bins, is the previous amount of occupied bins plus the chance that an unused bin gets filled:

$$G_1(2,4) = 1 + \frac{4-1}{4}$$

The same goes for the third, which looks more complicated:

$$G_1(3,4) = \left(1 \quad + \quad \frac{4-1}{4}\right) \quad + \quad \frac{4 - \left(1 + \frac{4-1}{4}\right)}{4}$$

$$G_1(3,4) = G_1(2,4) + \frac{4 - G_1(2,4)}{4}$$

$$G_1(3,4) = G_1(2,4) + 1 - \frac{G_1(2,4)}{4}$$

Basically, it can be read like: "The average amount of occupied bins (*$G_1(x,n)$*), is equal to the previous amount of occupied bins (*$G_1(x-1,n)$*) plus the chance that a hash has a different container as prefix than any of the previous set of occupied containers." ■

This recursive formula can be reduced to a normal, computational cheaper version.

Lemma 5.2.b

$$G_1(x,n) = n \cdot (1-(1-\frac{1}{n})^x)$$

Proof of Lemma 5.2.b

This recursive formula can first be rewritten as a summation formula which is then reduced to a regular function, proving the lemma.

Rewriting gives us a formula with a single recursive factor:

$$\begin{aligned} G_1(x,n) &= G_1(x-1,n) + 1 - \frac{G_1(x-1,n)}{n} \\ &= 1 - \frac{1}{n} * G_1(x-1,n) + G_1(x-1,n) \\ &= (1-\frac{1}{n}) * G_1(x-1,n) + 1 \end{aligned}$$

Substituting $G_1(x,n)$ with the previous formula renders:

$$\begin{aligned} G_1(x+1,n) &= (1-\frac{1}{n}) * G_1(x,n) + 1 \\ &= (1-\frac{1}{n}) * \left((1-\frac{1}{n}) * G_1(x-1,n) + 1 \right) + 1 \\ &= (1-\frac{1}{n})^2 * G_1(x-1,n) + (1-\frac{1}{n}) + 1 \end{aligned}$$

Substituting $G_1(x+1,n)$ with the previous formula. A pattern has emerged with a constant $G_1(x-1,n)$:

$$\begin{aligned} G_1(x+2,n) &= (1-\frac{1}{n}) * G_1(x+1,n) + 1 \\ &= (1-\frac{1}{n}) * \left((1-\frac{1}{n})^2 * G_1(x-1,n) + (1-\frac{1}{n}) + 1 \right) + 1 \\ &= (1-\frac{1}{n})^3 * G_1(x-1,n) + (1-\frac{1}{n})^2 + (1-\frac{1}{n})^1 + 1 \end{aligned}$$

Substituting $G_1(1,n)$ by one renders the summation:

$$\begin{array}{|ll}
G_1(1,n) & = 1 \\
G_1(2,n) & = (1-\frac{1}{n})^1 \cdot G_1(1,n) + 1 \\
 & = (1-\frac{1}{n})^1 \cdot 1 + 1 \\
G_1(3,n) & = (1-\frac{1}{n})^2 + (1-\frac{1}{n})^1 + 1 \\
G_1(4,n) & = (1-\frac{1}{n})^3 + (1-\frac{1}{n})^2 + (1-\frac{1}{n})^1 + (1-\frac{1}{n})^0 \\
G_1(x,n) & = \sum_{i=1}^{x} \left[(1-\frac{1}{n})^{i-1} \right]
\end{array}$$

Using this rule:

$$\left| \sum_{i=1}^{x} c^{i-1} \quad = \quad \frac{c^{x+1}}{c\cdot(c-1)} - \frac{1}{c-1} \quad = \quad \frac{c^x - 1}{c-1} \right.$$

A summation formula is constructed for $G_1(x,n)$.

$$\left| \begin{array}{llllll}
\sum_{i=1}^{x} (1-\frac{1}{n})^{i-1} & = & \dfrac{(1-\frac{1}{n})^x - 1}{(1-\frac{1}{n}) - 1} & = & \dfrac{(1-\frac{1}{n})^x - 1}{-\frac{1}{n}} & = \\
 & & n\cdot(1-(1-\frac{1}{n})^x) & & &
\end{array} \right.$$

■

The number of occupied containers, cannot exceed the number of available containers *n*, so the limit x->∞ of this formula needs to be *n*:

$$\lim \xrightarrow{x\to\infty} n\cdot(1-(1-\frac{1}{n})^x) = n$$

Since $\lim \xrightarrow{x\to\infty} (1-\frac{1}{n})^x = 0$ holds, a little bit of rewriting proves this limit by lowering the result with -1 results in having the entire relevant part of the formula under the x-axis. When the formula is flipped and stretched by -*n*, we get our new formula with the limit of *n*.
Every level has the same characteristics as they all have a number of containers or potential total amount of tree nodes called *n*. The existence of the nodes relies on the amount of nodes *n* and the amount of hashes *x* being provided as input.

The tree specific characteristic is that the number of hashes remain constant, while the total amount of different hash prefixes are doubled with each level.

Still, the intersection tree T_{AB} is the only tree that matters, and this shall be looked at in the next section.

5.3 DETERMINE THE SIZE OF THE INTERSECTION TREE

Having determined the average size of a level in an individual tree with G_1, this section will show that knowing the amount of nodes on a level is sufficient for determining the average intersection size of a level in the intersection tree. This function is defined as $G_2(x,n)$, where x represents the number of elements each party has and n is the amount of available spaces available in a certain level, and will be introduced in paragraph 5.3.2.

A level in the intersection tree T_{AB} can be viewed in a similar manner as a level in the individual distribution tree T_X in the previous section. Since distribution is individually random for each level, intersecting these levels doesn't require the context of the tree either.

If every tree level can be represented by a bit string in which each '1' represents a node and every '0' the absence of a node. The length of this string would represent the total amount of available nodes on that level. Since the two individual trees are considered equal, the same level of each party shares the same amount of existing nodes. For nodes to intersect, they should share the same tree-prefix or string index in the level representation.

Equal set sizes
There is a configuration independent variable which has a large influence on the average communication cost of the algorithm. Namely the proportion of the set sizes of the participating parties. We already stated that the set size are equal, the reason is that it generates the most communication for the same total amount of hashes. Since the average amount of bits communicated should never exceed three bits, the influence of this proportion should result in the largest $|T_{AB}|$ possible.

Consider sets of numbers which don't necessarily need to be unique. Comparing ten numbers with two numbers would take $10 \cdot 2 = 20$ comparisons. The same combined amount of numbers with equal set sizes, would render $6 \cdot 6 = 36$ comparisons.

Therefore, the tree size will be the largest when the two individual sets I_A and I_B are of equal size. This also gives a computational advantage since proportion of set sizes no longer count and the individual tree only needs to be calculated once.

5.3.1 PROBLEM DESCRIPTION

Below is a representation of an example in which two bit strings with the length of n=5 spaces and x=2 nodes each are depicted. Each bit in each bit string can be viewed as a ball which either has the value '1' or '0'. If the two bit strings were two ball sacks, the problem can be described as follows. And only for this section, x is defined as size of the partition with '1's.

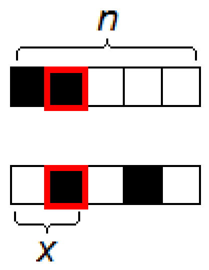


Figure 8 *- A representation of two binary strings being compared with a single node in common. Black represents a '1', white represents a '0'.*

If we were to take a ball out of each sack simultaneously without the possibility of putting it back, then the number of times two '1's were drawn is of interest; i.e. the intersection.

Essentially a ball drawn from the second sack, only has meaning when the first ball is a '1'. The chance that the second ball is also a '1', is independent of the order in which the balls from the first sack are drawn. To illustrate this using the example, imagine the second last ball from the first sack being the first '1' drawn. Without knowledge about the previous draws, the chance that the second ball is a '1' as well, is x/n or 2/5.

5.3.2 HYPERGEOMETRIC DISTRIBUTION

Lemma 5.3.a

$$G_2(x,n) = n \cdot ((1-\frac{1}{n})^x - 1)^2$$

Proof of Lemma 5.3.a

Hypergeometric distribution was found to be applicable to the problem description. It is a discrete probability distribution used to determine the number of successful draws out of a certain population without replacement.

Essentially it says: "If you have a bag with N balls of which m are potentially successful draws, the Hypergeometric function describes the number of successful draws when n balls are drawn without replacement".

As such we are interested in the average number of intersections on a certain level, with a certain amount of nodes n and certain amount of used hash prefixes x for both the individual participants.
Following the notation in the table below we can apply the hypergeometric distribution to our problem. For each existing node n *(x hash prefixes)* in one level, a random node in the other level of size N (*n nodes*) is checked and will not be checked a second time. If this node exists, we have a successful draw k.

The Hypergeometric distribution function is defined with these variables:

	Drawn	**Not drawn**	**Total**
Successes	k	$m-k$	m
Failures	$n-k$	$N+k-n-m$	$N-m$
Total	n	$N-n$	N

The hypergeometric distribution is designed to calculate the chance of getting a certain amount of successful draws. The predefined mean for the hypergeometric distribution however, is able get the average amount of successful draws and is defined as $m \cdot n/N$. After substitution, this mean is equal to $|x|^2/n$.

With the two formulas for calculating the overlap of nodes and the corresponding intersection of that level, the new formula for a single level is defined as:

$$G_2(x,n) = \frac{\left(n \cdot (1-(1-\frac{1}{n})^x)\right)^2}{n} = \frac{n^2 \cdot (1-(1-\frac{1}{n})^x)^2}{n} = n \cdot (1-(1-\frac{1}{n})^x)^2$$

■

5.3.3 THE INTERSECTION OF THE HASHSETS TAKEN INTO ACCOUNT

As mentioned earlier, the intersection of inputted hashes is assumed to be empty for the communication complexity analysis. There are two reasons for assuming an empty intersection. One is that [Tee06] assumes the intersection is empty, and the other is that it is a very difficult property to take into account.

Consider two distinct sets HI_A and HI_B, both are the same size and have respective trees T_A and T_B. When looking at their mathematical intersection tree T_{AB} with HS_{AB+}=0, it is as good as certain that the hashes will stop coinciding before the last tree level is reached.

Every node in T_A has the same chance of intersecting any node on the same level in T_B. But if suddenly one hash was shared, it is certain that one node on each level will intersect with another, these nodes don't behave the same way as the other truly random nodes. The other random nodes can still intersect with these predefined nodes, reducing the communication cost of the shared hash, but at some level where random hashes generally stop intersecting, the shared hashes will add to the communication cost because they will intersect till the last tree level. Determining where the level is from which the shared hashes start to add communication cost is very difficult and therefore the shared hashes are not part of this thesis.

5.3.4 RELATION BETWEEN TREE SIZES

What we are trying to prove is that whatever the tree size is, the number of bits communicated per hash remains below three bits. In other words, the scalability of the communication cost needs to be proven. In laymen terms, if the number of shared hashes are doubled, so must the tree size. Hereby proving that the amount of bits per hash in the mathematical intersection tree has a limit. This problem is attacked by first proving this for any tree level after which we will show that therefore the same property holds for the tree as well.

Lemma 5.3.b
For any natural number c:

$$G_2(x,n) \approx \frac{G_2(c \cdot x, c \cdot n)}{c}$$

Proof of Lemma 5.3.b
This lemma simply states that the answer to the intersection function for a single level stands in linear proportion to the original parameters. It is not possible to cancel out the multiplying factor *c*. Therefore we will determine the limit for *c*:

$$\lim \xrightarrow{c \to \infty} \frac{cn \cdot (1-(1-\frac{1}{cn})^{cx})^2}{c} = n \cdot (1-\mathrm{e}^{-\frac{x}{n}})^2$$

Therefore:

$$n \cdot (1-(1-\frac{1}{n})^{x})^2 \approx n \cdot (1-\mathrm{e}^{-\frac{x}{n}})^2$$

This formula depends on the fact that $(1-1/n)^x$ has a limit and is only inaccurate with very small sizes but proves that those differences will only get smaller with a greater constant. ■

The scalable tree
When the parameters *x* and *n* are doubled, we are calculating the level which is one higher in the tree with a doubled amount of input hashes and result. Effectively when doubling *x*, we should get a tree twice the original size.

The proven Lemma 5.3.b, states that this holds: $G_2(c \cdot x, 2^{l+c}) \approx c \cdot G_2(x, 2^l)$.

Lemma 5.3.c

$$\sum_{l=0}^{\infty} G_2(2 \cdot x, 2^l) \approx 2 \cdot \sum_{l=0}^{\infty} G_2(x, 2^l) + 1$$

Proof of Lemma 5.3.c
The calculation for a single level can be written as a summation over multiple levels:

$$(G_2(2 \cdot x, 2 \cdot n) \approx 2 \cdot G_2(x,n)) \Rightarrow$$

$$(\sum_{l=1}^{\infty} G_2(2 \cdot x, 2^{l+1}) \approx 2 \cdot \sum_{l=0}^{\infty} G_2(x, 2^l))$$

Counting from level 1 is not the same as doubling the tree size, there is a small inaccuracy

$$\sum_{l=0}^{\infty} G_2(2\cdot x, 2^{l+1}) - G_2(2\cdot x, 2^0) \approx 2\cdot\sum_{l=0}^{\infty} G_2(x, 2^l) \equiv$$

$$\sum_{l=0}^{\infty} G_2(2\cdot x, 2^{l+1}) \approx 2\cdot\sum_{l=0}^{\infty} G_2(x, 2^l) + 2^0$$

■

5.3.5 INTERSECTION TREE SIZE

With a very small inaccuracy, the size of the tree approximately doubles when the number of hashes is doubled. Furthermore, the small inaccuracy becomes negligible when the tree size grows. This section will give a decent indication for this fact as well using algebra.

Lemma 5.3.d

$$\frac{|T_{AB}|}{|HI_A| + |HI_B|} = 1$$

Proof of Lemma 5.3.d

The number tree nodes in $|T_{AB}|$ per hash is formulated as follows:

$$\frac{\sum_{l=0}^{\infty} G_2(x, 2^l)}{|HI_A| + |HI_B|} = \frac{\sum_{l=0}^{\infty} 2^l \cdot (1 - (1 - 2^{-l})^x)^2}{2\cdot x}$$

The same formula using the limit is defined in a similar fashion:

$$\frac{\sum_{l=0}^{\infty} 2^l \cdot (1 - \mathrm{e}^{-\frac{x}{2^l}})^2}{2\cdot x}$$

In the plot in Figure 9 of these functions below, the *x*-axis stands for the total amount of hashes.
The *y*-axis represents the number of nodes in T_{AB} per total amount of hashes. The top formula is the one using the actual tree size formula, but the bottom formula using the limit is increasingly accurate.

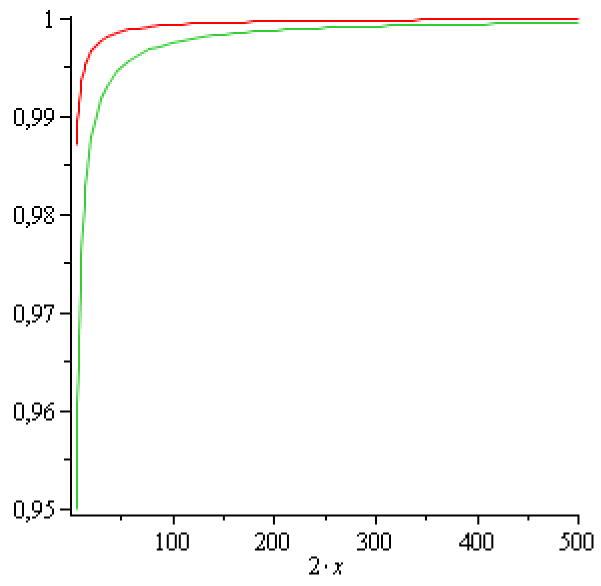


***Figure 9** - The top plot is the regular more accurate function and the bottom plot is the formula which incorporates the limit exp(-x/n). Both calculate the number of nodes per hash in the tree.*

The limit of both formula's in Figure 9 is one. This implies one tree node per hash, so the mathematical intersection tree T_{AB} contains one node per hash for the total amount of hashes. ■

5.4 INTERSECTION PROTOCOL

With $|M_{Ask}| = |T_{AB^*}|$, Lemma 5.1.d stated that the amount of *ask* messages is determined by the mathematical tree, plus the influence of spurious nodes. The size of the intersection tree $|T_{AB}|$ has been determined but the amount of spurious nodes has not. Section 5.4.1 produces a formula to determine the amount of spurious nodes for any level, called G_{S1}.

5.4.1 SPURIOUS NODES

Some of the nodes in the communicated intersection T_{AB^*} exist as a consequence of the very way the sub protocol has been setup. Those nodes are the so called spurious nodes as explained in Section 4.2.4.1. What has not yet been explained is the large influence they have on the communication complexity.

In short, a spurious node emerges whenever party A claims that she is in possession of hashes corresponding to a prefix p_1 but party B didn't have hashes with that prefix at all. The ask message, sent by party A, is nevertheless sent and therefore counts as a node in the communicated intersection T_{AB^*}.

Let $n \in T_{AB^*}$, let a be the binary answer whether a party sent an *ask* (1) or *refuse* (0) message and let b be the binary answer whether the other player has the node (1) or not (0). The predicate $P(n,a,b)$ would then describe whether a situation, as described in Table 10 below is true.

PREDICATE ($n \in T_{AB^*}$)	***MEANING***	***# MSG***
$P(n,1,1)$	Both parties have the node n	3+
$P(n,0,0)$	Neither party has the node n	1
$P(n,1,0)$	The message sending party claims to have n which the other party does not have; spurious node.	3
$P(n,0,1)$	The message sending party states that it doesn't have n	1

***Table 10** – A list of the meanings for different kinds of input for P(n,a,b).*

Here $P(n,1,0)$ is of interest, since it describes the only nodes for which ($n \in T_{AB^*} \wedge n \notin T_{AB}$) holds. Such a situation should have been communicated as $P(n,0,0)$ and therefore costs two bits more. The situation in which one of the two parties has the node is denoted as a **spurious situation**.

To illustrate the number of spurious nodes, consider that after every node in de mathematical intersection tree T_{AB} that has one or zero children, there is are one or two possible spurious situation. If every spurious situation would be an actual spurious node, the amount of spurious nodes would be equal to the size of T_{AB} itself.

Lemma 5.4.a
The amount of spurious nodes send, is half the amount of spurious situations.

Proof of Lemma 5.4.a
Which is the sending party, alternates per level. In a 5 level deep tree in which every node is send, party A sends 1+4+16=21 *ask* messages, whereas party B sends 2+8=10 *ask* messages. On every level however, the number of P(n,1,0) situations is equal to the number of P(n,0,1) situations since both parties have equally sized sets. Therefore half of the spurious situations is a spurious node. If the input sizes would differ from each other, it would be harder to calculate. ■

Problem description

To determine the amount of spurious nodes a formula can be constructed.

Let $G_S(x,n)$ be the formula which describes the average amount of spurious situations on any level of size n.

The same problem description for determining the intersection in Section 5.3.1 can be used to describe a spurious node. We still have two sets of bins of equal amount n, however the number of balls differ.
The first set still has x balls, corresponding to the number of nodes in the individual distribution tree. The second set contains n-x balls instead of x, describing the absence of nodes in the other identical individual distribution tree. Effectively we are counting the amount of spurious situations for one party, which is equal to dividing all the spurious situations by two.

A spurious situation can only occur if the parent of the spurious situation resides in the mathematical T_{AB}. This is because the participants take turns on communicating hash prefixes per level. A spurious node can therefore not have another spurious node as a parent.
The spurious situations with parent are defined as the set S_1, the set of spurious situations without parent is defined as S_0. In order to determine $|S_1|$, we first look at the amount of spurious situations $|S_0|+|S_1|$ disregarding whether the parent is in the intersection or not. We then add a formula to take the parents into account.

Solution to problem

Lemma 5.4.b

$$G_S(x,n) = \; |S_0| + |S_1| \; = \; n \cdot ((1-\frac{1}{n})^x - (1-\frac{1}{n})^{2x})$$

Proof of Lemma 5.4.b
Using the Hypergeometric distribution function again:

	DRAWN	*NOT DRAWN*	*TOTAL*
SUCCESSES	k	$m-k$	m
FAILURES	$n-k$	$N+k-n-m$	$N-m$
TOTAL	n	$N-n$	N

A successful draw here, is when one of the filled bins x in the first set is matched with one of the non-filled bins (n-x) in the second set. The predefined mean $m \cdot n/N$ now becomes $x \cdot (n-x)/n$ by substituting variables and returns the average amount of spurious situations given x and n.

Omitting the details of rewriting, the formula retrieved by substituting x by G_1 :

$$G_S(x,n) = \frac{G_1(x,n)\cdot(n - G_1(x,n))}{n} \equiv$$

$$G_S(x,n) = \quad n\cdot((1-\frac{1}{n})^x - (1-\frac{1}{n})^{2x})$$

■

Now the amount of nodes with or without parent has been determined, the portion of those which have a parent will now be determined, the function is hereby defined as $G_{S1}(x,n)$.

Lemma 5.4.c

$$G_{S1}(x,n) = \frac{2\cdot G_2(x,n/2) - G_2(x,n)}{n - G_2(x,n)}\cdot G_S(x,n)$$

The different situations which take the presence of the parent into account are depicted in the table below:

#	*SITUATION*	*DESCRIPTION*
1	$p\in T_{AB}$	The nodes with a parent
2	$p\in T_{AB} \wedge (P(n,1,0) \vee P(n,0,1))$	The actual spurious nodes S_1
3	$p\in T_{AB} \wedge (P(n,1,1))$	The intersection nodes with parent
4	$(P(n,0,0) \vee P(n,1,0) \vee P(n,0,1))$	The potential spurious nodes $S_0 \cup S_1$

Table 11 *– This table depicts the different type of partitions on level m that are necessary to construct G_{S1}.*

Since the situation $P(n,1,1)$ cannot occur if $p\notin T_{AB}$ is assumed, the amount of bins the intersection nodes of level m are spread over is:

(1) $\qquad\qquad\qquad\qquad$ $p\in T_{AB}$

$$2\cdot G_2(x,2^{m-1})$$

Which defines the amount of nodes at level m which have a parent. Only the nodes in this section which are not part of the intersection at level m $G_2(x,2^m)$ can be part of S_1 and since all intersecting nodes on level m have a parent:

(1)-(3) $\qquad\qquad\qquad\qquad$ $p\in T_{AB} \wedge (P(n,0,0) \vee P(n,1,0) \vee P(n,0,1))$

$$2\cdot G_2(x,2^{m-1}) - G_2(x,2^m)$$

The amount of nodes that isn't part of the intersection at level m (i.e. 2^m - $G_2(x,2^m)$), is available for the spurious situations in S_1 and S_0. The amount of spurious situations defined as S_1 becomes:

((1)-(3))/(4) * ($|S_0|+|S_1|$) $\qquad\qquad\qquad\qquad$ $p\in T_{AB} \wedge (P(n,1,0) \vee P(n,0,1))$

$$\frac{2\cdot G_2(x,2^{m-1}) - G_2(x,2^m)}{2^m - G_2(x,2^m)}\cdot G_S(x,2^m)$$

■

Plotting this graph for 20 and 2000 hashes renders Figure 10 below:

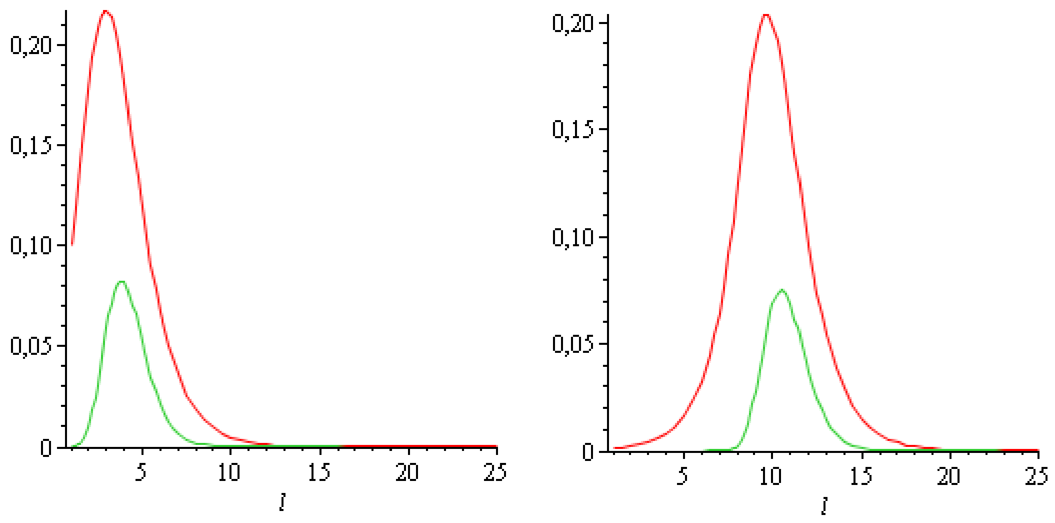


Figure 10 *- Two graphs each with 2 different plots indicating the number of spurious nodes per hash (bottom plot) in relation to the number of intersection nodes per hash (top plot) at each level l. The left graph is calculated for 20 hashes and the right for 2000 hashes.*

Calculating the average amount of spurious nodes per hash for different total amounts of hashes with a summation for each level renders:

$$\lim_{x \to \infty} \frac{\sum_{l=0}^{\infty} G_{S1}(x,2^{l})}{2 \cdot x} \approx 0,223$$

renders a little over 0,223 nodes per hash. Even though it is no proof, the graph is depicted in Figure 11 below.

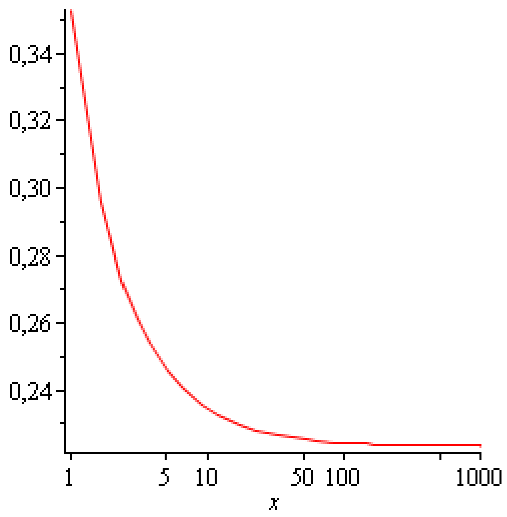


Figure 11 *- A graph which calculates the number of spurious nodes per hash. Where x denoted the number of hashes of one party.*

5.5 COST OF SUB PROTOCOLS

The total cost of the algorithm can now be determined using Lemma 5.1.g, $|T_{AB}|$ and the amount of spurious nodes S.

The total cost of the algorithm is determined by:

$$C_{Total} = 2 \cdot |T_{AB*}| + 1$$

The communicated intersection size $|T_{AB*}|$ is calculated by adding the average amount of spurious nodes to the mathematical intersection size $|T_{AB}|$, which has been determined to be equal to $2{\cdot}x$. The latter two have now been defined as constants per total number of hashes $2{\cdot}x$.

$$|T_{AB*}| = |T_{AB}| + |S|$$

$$\frac{|T_{AB*}|}{2 \cdot x} = \frac{|T_{AB}|}{2 \cdot x} + \frac{|S|}{2 \cdot x}$$

$$\frac{|T_{AB*}|}{2 \cdot x} = 1 + 0{,}223$$

By writing the formula for the total cost as the total cost per hash, we obtain:

$$C_{Total} = 2 \cdot |T_{AB*}| + 1$$

$$\frac{C_{Total}}{2 \cdot x} = \frac{2 \cdot |T_{AB*}|}{2 \cdot x} + \frac{1}{2 \cdot x}$$

By substituting $|T_{AB*}|$ by our new constant which is inaccurate for small amounts of hashes:

$$\frac{C_{Total}}{2 \cdot x} \approx 2 \cdot (1 + 0{,}223)$$

$$\frac{C_{Total}}{2 \cdot x} \approx 2{,}446$$

5.6 COMPARING THE RESULTS

The dissertation by Teepe presented an average amount of nodes per hash which had been produced by simulating a lot of T-2 executions in which:

- Two parties used the cooperative configuration with efficient encoding;
- the amount of nodes was equal for all parties;
- the amount of nodes for the different kind of simulations were 1, 10, 100 and 1000;
- each simulation consisted of 1000 executions to get a approximately accurate indication of the average sizes.
- The actual intersection is empty; $|HS_{AB}|=0$.

We produced the averages for the same inputs using our proven formula for calculating the intersection tree size and the formula for spurious nodes. If x is still the amount of nodes entered by a single party:

$$\frac{C_{Total}}{2 \cdot x} \approx 2{,}446$$

In Table 10, the average amount of bits communicated per total amount of hashes are depicted:

$\lvert HI_Q \rvert$	Teepe's result *Simulation*	Our result *Simulation*	*Calculation*
1	2,986	3,456	3,207
10	2,783	2,798	2,492
100	2,756	2,765	2,451
1000	2,755	2,755	2,447

***Table 12** - The calculated average amount of bits per total amount hashes for two parties from two different approaches.*

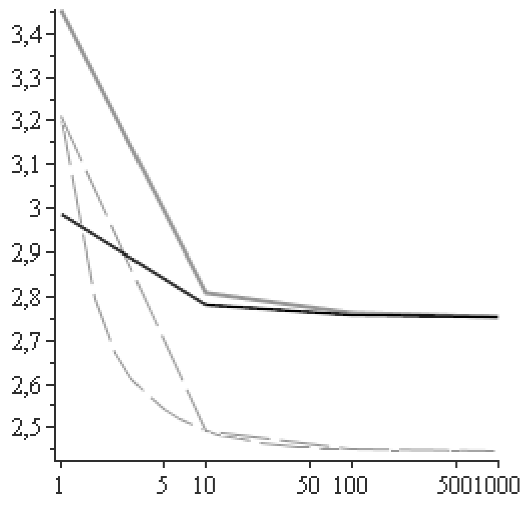


***Figure 12** - The graph shows the different results between our simulations, our calculations and Teepe's simulations. The black line is Teepe's simulation and the gray line is ours. The dashed plots represent our calculations.*

Even though the two simulations differ in the beginning, they end up with the same number of nodes per hash. Our calculations have the same shape as our simulations, but have a different result in number of bits per hash. Why there is a different shape is hard to tell. The difference between calculations and simulations can most likely be attributed to the spurious node formula G_{S1}, since the intersection formula is checked against the simulations. When the thesis was not yet finished, the first error we found in G_{S1} was related to the dependency on the parent. And I believe I have found the answer to that problem, so I believe the answer must lie somewhere else.

6 CONCLUSION

The sole purpose of this thesis, was to prove that by using the algorithm T-2, the number of bits communicated per total amount of hashes, is below three bits on average; assuming each hash represents a unique reference to some information. In order to answer this question we needed to understand how the algorithm worked and why it worked. Of course the latter is something you come across when trying to understand how the algorithm works.

INFORMATION INPUT

The algorithm accepts unique sets of confidential information. Each piece of confidential information is converted into a hash, since there should be unique references. How the information is converted or what information is provided as input was not the concern of this thesis. Nevertheless, in section 2.2 we learn that the input should not be susceptible to dictionary attacks.

STEPS IN THE ALGORITHM

Step 1 - Converting the information into hash trees

The first step of the algorithm is to convert the input into hashes followed by converting these hashes into individual binary hash trees; as explained in chapter 3.2. The algorithm makes use of the binary tree since it is fast and efficient to communicate and compare. Binary trees have a certain behavior when being filled with hashes. In the first couple of levels, every node in the tree is used and represents one or more hashes at once. If another party doesn't use that node, the corresponding hashes are easily disregarded. So every node in the tree is essentially a prefix for zero or multiple hashes. As soon as a level contains a node which doesn't represent any hashes, an empty node, this level and levels above may start to show prefixes or nodes which only represent one node. Those so called strings represent unique hashes. With a suitable hash length the very top of the tree will only have filled nodes representing one hash left.

Step 2 - Comparing the list of hashes

With the intersection trees from the previous step the algorithm starts to intersect them using two protocols as explained in chapter 4.2.4. One is actually for intersecting the individual trees, the other is for proving possession of shared hashes. Both use a configuration which is both efficient and easy to understand. Rules for the protocol and their cost have been copied from Teepe's dissertation.

COMMUNICATION COST

Individual tree size

The individual tree size is only determined by the number of hashes. It is in section 5.2 where we find that looking at the problem per level is possible and easier. The tree size is then calculated for each level.

Intersection tree size

Building from the formula found for the individual tree size, the number of nodes per level in the intersection tree is found after simulating and applying the hyper geometric distribution. Apparently, the number of shared hashes is very difficult to work with and fortunately not necessary in order to prove that the amount of bits per hash communicated stays does not exceed three bits. Since the amount of shared hashes is zero, the second protocol hereby becomes irrelevant.

For the actual intersection tree, we prove in section 5.3.4 that whatever the amount of hashes is, if the total amount of hashes is doubled, the corresponding tree size is doubled as well, with the side note that there is a margin of error which decreases as the number of hashes increase.

This is where the thesis has already answered the most important question. Nevertheless, we were still interested in the actual cost of the intersection protocol.

Actual intersection tree size

We needed to know what the amount of spurious nodes is and what the actual tree size is. The actual tree size was quickly calculated by simply executing the formula which was made for each level. The spurious node however, is a special case. Here, one party sends an *ask* message for a node which the other party does not have. The amount of these nodes is difficult to calculate, but apparently they account for a relevant amount of messages sent.

The formulas suggest that with a large amount of hashes, the number of bits communicated per hash will ultimately be 2,446. This number does not coincide with the results of the simulations. We have tried to determine what might cause these differences, but we have failed in doing so. Fortunately, those results have nothing to do since whatever the number was, the thesis proved that the number of bits per hash communicated decreases as the amount of hashes inputted increases.

LITERATURE

[DY83] Danny Yolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198-208. March 1983

[Sud06] Thomas A. Sudkamp. Languages and Machines. 2006

[Tee06] Wouter Teepe. Reconciling information exchange and confidentiality. A formal approach. 2006