Bachelor thesis
Computing Science

Radboud University

Java Implementation and Analysis of Multi-party Private Set Intersection Protocols

Author:
Michael de Jong
s4788451

First supervisor/assessor:
dr. Simona Samardjiska
simonas@cs.ru.nl

Second assessor:
dr. Asli Bay
A.Bay@cs.ru.nl

June 24, 2020

Abstract

Multi-party private set intersection (MPSI) is a secure multi-party computation technique that allows a number of parties holding a private dataset to compute the set intersection of these datasets. None of the parties reveal any other information to the other parties except the elements in the intersection. In this thesis we investigate a newly proposed MPSI protocol based on Bloom filters and two other relevant MPSI protocols based on Bloom filters and oblivious polynomial evaluation. We implement these protocols using the Java programming language and analyze the computational and communication performances of these protocols by measuring the execution time and communication required and comparing the results of our benchmark. The findings of this analysis indicate that the newly proposed MPSI protocol performs better than the other two protocols in terms of computational time, but requires more communication than the protocol based on oblivious polynomial evaluation.

Contents

Chapter 1

Introduction

Multi-party private set intersection (MPSI) is a useful secure multi-party computation technique that forms the basis for many important privacy-preserving applications and can be applied in many real-world situations. Therefore, the research community is constantly investigating, improving and extending protocols that implement MPSI. In this thesis, we will implement and analyze a newly proposed MPSI protocol that claims to be the most efficient among existing protocols that are based on public-key techniques. We will compare it to two other relevant protocols found in the literature and test the practical performances in terms of computational costs and amount of communication required.

The future of autonomous cars is bright and will change the lives of everyone. However, in this age of increasing processing power and data availability, information privacy is becoming progressively important. In this thesis, we will use the example of autonomous cars in a system of smart roads. An important requirement of such a system is the ability to track specific vehicles without infringing the privacy of other road users in the case of urgency or criminal activity. In order to learn the set of vehicles following a certain path, we can reformalize the problem as an MPSI problem: Each smart road holding a set of vehicles represents a party. These parties compute the private set intersection of their sets. Each party will only reveal the vehicles in the intersection and nothing else. Each smart road could track its vehicles by using smart cameras for example. We aim to solve this problem by utilizing an efficient MPSI protocol.

Our research question is:
"**How does the newly proposed MPSI protocol perform compared to relevant MPSI protocols?**" We will find the answer to this question based on the results we obtain with our measurement framework and Java implementations of the various protocols.

1.1 Overview of this thesis

In chapter 2 the preliminaries that are needed in order to implement the protocols will be made clear. Chapter 3 is about the newly proposed MPSI protocol and its implementation details. Chapter 4 and 5 are about two other relevant MPSI protocols. Chapter 6 deals with the implementation details of the MPSI protocols and our measurement framework. Lastly, in chapter 7 we will compare the three protocols using the practical example of tracking vehicles on smart roads and analyze the performances.

Chapter 2

Preliminaries

This chapter describes all relevant background information and building blocks needed to understand the various MPSI protocols. In this thesis we analyze the newly proposed MPSI protocol [1] based on Bloom filters. The protocol proposed in [2], which is also based on Bloom filters. And a polynomial based protocol proposed in [3].

2.1 Multi-party private set intersection

Multi-party private set intersection (MPSI) is a generalization of two-party private set intersection (PSI) to multiple parties. PSI is a cryptographic technique in the area of secure multi-party computation. Let there be n parties. Each party i has its own set of elements S_i. The goal of MPSI is to securely compute $\bigcap_{i=1}^{n} S_i$ without leaking any additional information, e.g. the other elements in S, to other parties, except the cardinality of S_i.

2.2 Threshold public key encryption schemes

A threshold public key encryption scheme is a type of public key encryption scheme (PKE) where the private key is distributed among n parties such that at least k parties have to cooperate and provide partial decryptions that have to be combined in order to obtain a full decryption of the ciphertext. Formally, given a security parameter k and two finite sets $\mathcal{M}, \mathcal{R} \in \{0,1\}^*$, a threshold PKE scheme $T\Pi$ is a tuple of (probabilistic) algorithms $(KGen, Enc, ShDec, Comb)$ described as follows:

KGen(k, n, t, r): Key-generation algorithm that given security parameter k, number of private keys n, threshold t and random string r outputs $(pk, sk_1, sk_2, ..., sk_n)$ where pk is the public key and sk_i is the secret key share of party i.

Enc(pk, m, r): Encryption algorithm that given public key pk, message $M \in \mathcal{M}$ and random string $r \in \mathcal{R}$ outputs $Enc(pk, M, r)$, the ciphertext of message M under public key pk with randomness r.

ShDec(sk_i, C): Shared decryption algorithm that given a secret key sk_i and ciphertext C outputs the decryption share C_i.

Comb$(pk, \{C_1, ..., C_k\})$: Combining algorithm that given public key pk and a set of k decryption shares $\{C_1, ..., C_k\}$ where k $\geq$ t, outputs a plaintext message $M \in \mathcal{M}$ or $\perp \notin \mathcal{M}$ to indicate an invalid ciphertext.

2.3 Additively homomorphic encryption

A PKE scheme with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is said to be additively homomorphic if for all $(sk, pk) \leftarrow$ **KGen**(), all $M_1, M_2 \in \mathcal{M}$ and arbitrary scalar α, there exists an efficient homomorphic operation $+_H$ over $\mathcal{C}$ that satisfies two properties:

- Dec(sk, Enc(pk, M_1) $+_H$ Enc(pk,M_2)) = $M_1 + M_2$
- Dec(sk, αEnc(pk,αM_1)) = αM_1

Conclusively, the above properties let us define an algorithm that we can use to rerandomize a given ciphertext C without changing the plaintext of C. This is especially useful for security purposes that we will elaborate later in the various protocols.

Algorithm 1: Ciphertext rerandomization algorithm

```
1 Rerand (C);
2 return C +_H Enc(pk, 0)
```

Another algorithm that we will need is the decryption-to-zero variant of the shared decryption algorithm, which will let us compute a decryption share that will lead to a randomized decrypted value if the plaintext is different than zero. This way, other parties will only be able to tell if the decrypted value is zero and not learn anything else. This can be achieved by randomizing the ciphertext by each of the involved parties, combining the results in a new value, and jointly decrypting this obtained value. The decryption will result in a decryption of 0 if the ciphertext was an encryption of 0 and a random value otherwise.

2.4 Threshold Paillier encryption

A typical threshold additively homomorphic PKE scheme that we will use to implement MPSI is Threshold Paillier encryption [4], which is an extension to the original Paillier cryptosystem proposed by Pascal Paillier [5]. A useful property for this cryptosystem is that we can use it in our implementations to perform homomorphic computations on the ciphertexts and use group-wise decryption. Why this is important will be explained in more detail later for each specific protocol.

The Threshold Paillier cryptosystem as proposed in [4] is a probabilistic asymmetric algorithm for public key cryptography with computations in the group $\mathbb{Z}^*_{n^2}$ where n is an RSA modulus consisting of two large primes $n = pq$ and plaintext space $\mathbb{Z}_n$.

Key generation: Find four unique primes p, p', q, q' such that $p = 2p' + 1$ and $q = 2q' + 1$. We then compute $n = pq$ and $m = p'q'$. In our implementation we set $s = 1$ such that the plaintext space will be $\mathbb{Z}_n$. We choose d such that $d = 0 \; mod \; m$ and $d = 1 \; mod \; n^s$. Also pick $g \in \mathbb{Z}^*_{n^{s+1}}$ such that $g = (1+n)^j x \; mod \; n^{s+1}$ for a known j relatively prime to n and $x \in H$ where H is isomorphic to $\mathbb{Z}^*_n$. Then compute the polynomial $f(X) = \sum_{i=1}^{k-1} \alpha_i X^i \; mod \; n^s m$ where α_i is chosen randomly from $\{0, ..., n^s m - 1\}$ and $\alpha_0 = d$. The private key of the ith party is $s_i = f(i)$ and the public key is n.

Encryption: In order to encrypt a message $M \in \mathbb{Z}_n$, choose a random $r \in \mathbb{Z}^*_{n^{s+1}}$ and compute the ciphertext c as $c = g^M r^{n^s} \; mod \; n^{s+1}$.

Shared decryption: Given a ciphertext c, the decryption share of the ith party will be computed as $c_i = c^{2\Delta s_i}$ where $\Delta = l!$ with l being the number of decryption servers.

Combining: Given k or more decryption shares, we can combine these in order to get a decryption of c as follows. Let S be a subset of the k decryption shares. Then we compute:

$$c' = \prod_{i \in S} c_i^{2\lambda^S_{0,i}} \qquad where \qquad \lambda^S_{0,i} = \Delta \prod_{i' \in S \setminus i} \frac{-i}{i - i'} \in \mathbb{Z}$$

Conclusively, c' will be of the form $c' = c^{4\Delta^2 f(0)} = c^{4\Delta^2 d}$ and $4\Delta^2 d = 0 \; mod \; \lambda$ and $4\Delta^2 d = 4\Delta^2 \; mod \; n^s$ where $\lambda = 4M$. We conclude:

$$c' = (1+n)^{4\Delta^2 M} \; mod \; n^{s+1}$$

Where M is the plaintext of ciphertext c. For more details how M can be computed from c' we refer to the original paper [4].

2.5 Bloom filter

The Bloom filter is a vital data structure for the MPSI protocols in [1] and [2]. It is a probabilistic data structure designed with speed and space-efficiency in mind and was introduced by Bloom in [6]. A bloom filter can be seen as a space-efficient representation of a certain dataset and can be used to test whether an element belongs to that set with some false positive probability. False negatives are not possible.

The bloom filter (**BF**) consists of a bit array of length m:
$\mathbf{BF} = (\mathbf{BF}[0], \ldots, \mathbf{BF}[m-1])$ and k independent hash functions such that $H_i : \{0,1\}^* \to \{0, \ldots, m-1\}\ \forall i :\ 1 \leq i \leq k$.

We define 3 algorithms for the Bloom filter: initialization, element insertion and element membership check:

Algorithm 2: Initializes an empty Bloom filter

```
Initialize ();
BF ← bit array of size m
for i ← 0 to m − 1 do
    BF[i] ← 0
end
return BF
```

Algorithm 3: Inserts an element to the Bloom filter

```
Insert (BF, e);
for j ← 1 to k do
    i ← H_j(e)
    BF[i] ← 1
end
return BF
```

Algorithm 4: Checks whether the Bloom filter includes an element

```
Check (BF, e);
for j ← 1 to k do
    i ← H_j(e)
    if BF[i] = 0 then
        return False
end
return True
```

We can compute the optimal number of bits m and number of hash

functions k for a certain false positive rate ε as follows:

$$m = -\frac{n\, ln\, \varepsilon}{(ln2)^2}$$

$$k = \frac{m}{n} ln\, 2$$

2.5.1 Inverted Bloom filter

The inverted version of a Bloom filter is simply the Bloom filter where the bits of the bit array are flipped.

Algorithm 5: Inverts a given Bloom filter

```
1  Invert (BF);
2  IBF ← BF
3  for i ← 0 to m − 1 do
4  |   if IBF[i] = 1 then
5  |   |   IBF[i] = 0
6  |   else
7  |   |   IBF[i] = 1
8  |   end
9  end
10 return IBF
```

2.5.2 Encrypted Bloom filter

The encryption of a Bloom filter is an array where every entry of the bit array is encrypted using a public key pk.

Algorithm 6: Encrypts a given Bloom filter

```
1 Encrypt (BF, pk);
2 EBF ← array of size m
3 for i ← 0 to m − 1 do
4 |   EBF[i] = Enc_pk(BF[i])
5 end
6 return EBF
```

2.6 Oblivious polynomial evaluation

Oblivious polynomial evaluation is a fundamental technique that is used in the MPSI protocol in [3]. The generic oblivious polynomial evaluation

protocol is first described in [7] and deals with two parties P_1 and P_2 where P_1 holds a polynomial $Q(\cdot)$ over some field F and P_2 holds an element $t \in F$. The goal of the protocol is that P_2 learns $Q(t)$ and nothing else, while P_1 learns nothing at all.

A variant of the oblivious polynomial evaluation protocol that is used in [3] to perform a two-party private set intersection is described in [8]:

1. Party C holds set $X = \{x_1, \ldots, x_{k_C}\}$ and party S holds set $Y = \{y_1, \ldots, y_{k_S}\}$ consisting of elements in $\mathbb{Z}_n$. Moreover, C holds a secret key of a homomorphic encryption scheme of which the public key is known to S.

2. C then computes the coefficients of the polynomial

$$P(y) = \sum_{u=0}^{k_C} \alpha_u y^u$$

   of degree k_C with roots $\{x_1, \ldots, x_{k_C}\}$. Next C encrypts each of the α_u and sends $Enc_{pk}(\alpha_u)$ to S.

3. Then S applies the properties of the homomorphic encryption scheme to compute for every $y \in Y$: $Enc_{pk}(rP(y)+y) = r \cdot Enc_{pk}(\sum_{u=0}^{k_C} \alpha_u y^u) +_H Enc_{pk}(y)$ where r is a random value and $+_H$ the homomorphic operation. S sends the encrypted values to C.

4. Next C computes $X \cap Y$ by decrypting the received values and adding each $x \in X$ to the intersection for which there is a corresponding decrypted value.

2.7 Semi-honest security model

The security model defines how adversaries are assumed to deviate from the protocol specification. We differentiate between two security models: Semi-honest and Malicious security. In the semi-honest security model adversaries do not deviate from the protocol specification, but merely cooperate to infer information about honest parties. In the malicious security model adversaries may arbitrarily deviate from the protocol and change inputs and outputs in order to infer information about the other parties. The malicious security model provides a higher level of security, but is often much less efficient.

All of the MPSI protocols presented in this thesis are under the assumption of a semi-honest security setting. In our privacy preserving setting this is appropriate. However, for each protocol there are possible extensions that make them secure in the malicious setting at the cost of performance.

Chapter 3

New MPSI protocol

This chapter deals with the newly designed MPSI protocol in [1]. According to the paper, this new protocol has the lowest communication and computational complexity compared to other MPSI protocols found in the literature. The protocol is a continuation of the PSI protocol introduced in [9] and extends it from two-party to multi-party.

3.1 Description

The fundamental building blocks of the protocol are Bloom filters and threshold homomorphic PKE schemes. In this thesis we implemented the protocol using the threshold Paillier PKE, but other threshold schemes are possible as well. Let there be t parties involved: $P_1, \ldots, P_t$ where each party P_i holds a private dataset S_i of cardinality n_i. The protocol follows a client-server architecture where parties $P_1, \ldots, P_{t-1}$ will be considered the clients and party P_t will be considered the server. The final set intersection will be computed by the server.

3.1.1 Input

Each party P_i holds a public key pk and a shared decryption key sk_i of a threshold homomorphic PKE with decryption to zero algorithm $ShDec0$. The cardinality of each dataset n_i is also known to each individual party.

3.1.2 Sequence diagram

Figure 3.1 shows the sequence diagram of the protocol and describes the communication between server and client, together with all intermediate stages. The computational steps of each intermediate stage will be specified in detail below. The steps for each of the clients are identical and happen in parallel.

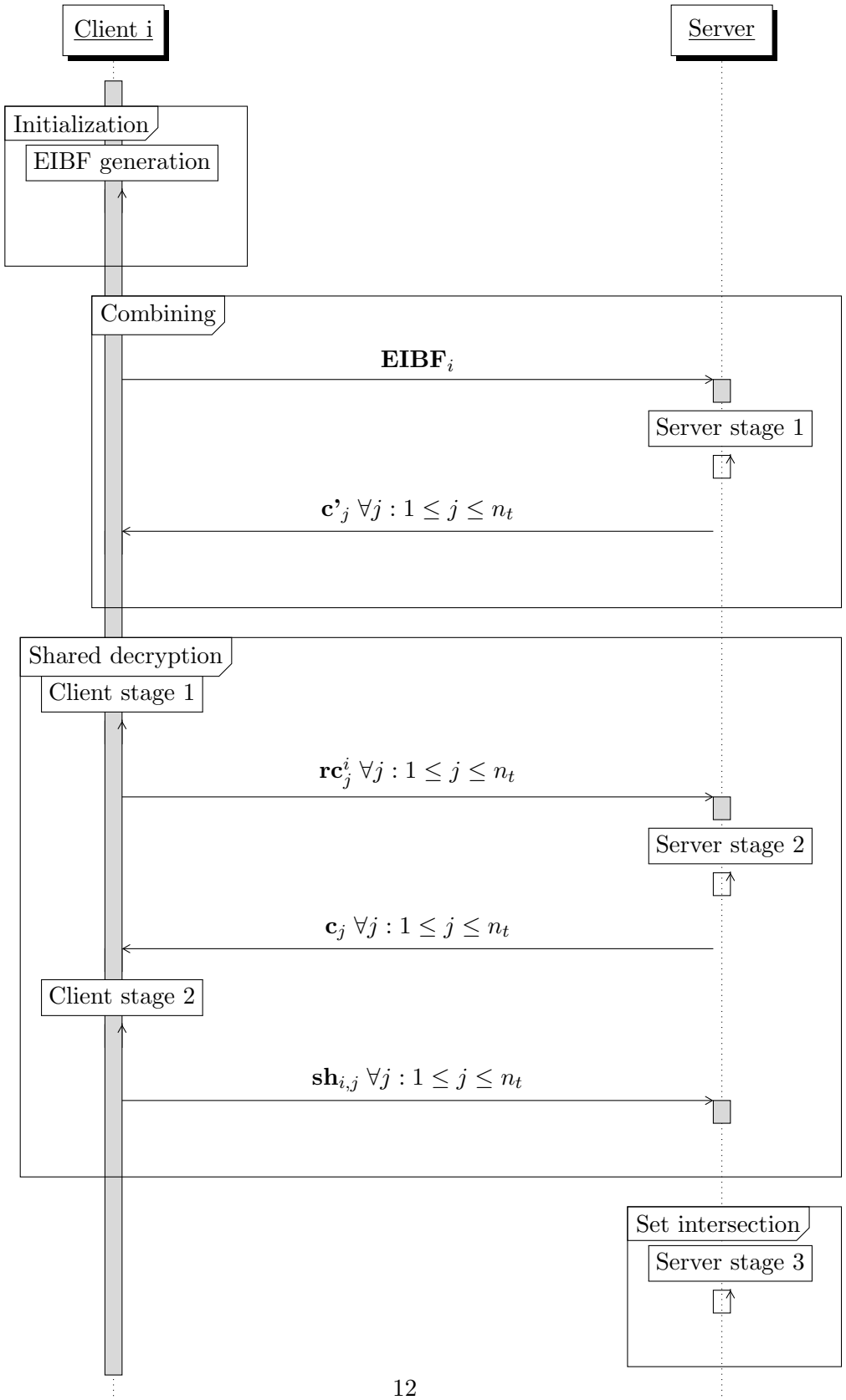

Figure 3.1: New MPSI sequence diagram

3.1.3 EIBF generation

Each client P_i where $1 \leq i \leq t-1$ computes the encrypted inverted Bloom filter ($\mathbf{EIBF}_i$) of their dataset S_i. This is done by first creating a new Bloom filter $\mathbf{BF}_i$ as in Algorithm 2. Then inserting each $e \in S_i$ to $\mathbf{BF}_i$ as in Algorithm 3. Next, the $\mathbf{BF}_i$ is inverted to obtain $\mathbf{IBF}_i$ using Algorithm 5 and then encrypted using the public key to obtain $\mathbf{EIBF}_i$ using Algorithm 6.

3.1.4 Server stage 1

let $+_H$ be the homomorphic addition operation of the threshold Paillier scheme, h_d represents the dth hash function of the Bloom filter, where k is the number of hash functions, y_j is the jth item in the server dataset.

1. Compute $\{C_1^{i,j}, \ldots, C_k^{i,j}\}\ \forall i,j$ where $C_d^{i,j} = \mathbf{EIBF}_i[h_d(y_j)]$.
2. $c_j^i = C_1^{i,j} +_H \ldots +_H C_k^{i,j}\ \ \forall i,j$
3. $c'_j = ReRand(c_j^1 +_H \ldots +_H c_j^{t-1})\ \ \forall j$

3.1.5 Client stage 1

Client i performs the first step of the $ShDec0$ algorithm and randomizes the received c'_j values by raising them by a random exponent such that the decrypted value is 0 if and only if the encrypted value is an encryption of 0:

1. Let r be a random exponent
2. $rc_j^i = ReRand((c'_j)^r)\ \ \forall j$

3.1.6 Server stage 2

The server sends the combined randomized c values back to the client.

1. $c_j = ReRand(rc_j^1 +_H \ldots +_H rc_j^{t-1})\ \ \forall j$

3.1.7 Client stage 2

The client computes its decryption shares:

1. $sh_{i,j} = ShDec(sk_i, c_j)\ \ \forall j$

3.1.8 Server stage 3

The final stage where the server computes the set intersection S.

1. $D(c_j) = Comb(pk, sh_{1,j}, \ldots, sh_{t-1,j})\ \ \forall j$
2. $S = \{y_j \mid D(c_j) = 0\}$

3.1.9 Remarks on protocol description and notation

Extra steps have been included to make the decryption-to-zero algorithm $ShDec0$ more explicit given the context of the threshold Paillier PKE. This includes raising the c'_j values by a random exponent to achieve the desired property of $ShDec0$. Also the protocol has been divided into various client and server stages, which describe the computations done by the corresponding party. This makes it easier to identify the correspondence with the java implementation.

3.2 Correctness

The correctness follows from the fact that if y_j is included in the intersection, then all encrypted inverted bloomfilter entries $\mathbf{EIBF}_i[h_d(y_j)]$ will clearly be an encryption of zero. Given the homomorphic properties of the encryption scheme, the sum and randomization of these values will still be an encryption of zero, hence all corresponding $\mathbf{c}_j^i$ decrypt to zero. If y_j is not in the intersection, then the decrypted values will be random.

Due to the possibility of false-positives in Bloom filters, the final set intersection has a neglectable small chance to include elements that are actually not in the true set intersection. However, Bloom filters have no false-negatives, so it cannot be the case that an element that should be in the set intersection is missing. The false-positive rate can be decreased by increasing the Bloom filter size at the cost of performance. See the Bloom filter section in the preliminaries chapter for more details.

3.3 Theoretical complexity

3.3.1 Theoretical communication complexity

For simplicity, we do not make a distinction between sending or receiving communication complexity.

The communication complexity of the client is dominated by the size of its EIBF containing m ciphertexts. Here, m is the size of the Bloom filter, which depends linearly on the size of the dataset n. Each client sends its EIBF to the server. Hence, the overall communication complexity of the client is $\mathcal{O}(n \cdot S)$ where S is the size of a ciphertext.

The server has to receive the EIBF of every client, hence the overall communication complexity of the server is $\mathcal{O}(n \cdot S \cdot t)$ where t is the number of clients.

3.3.2 Theoretical computational complexity

The computational complexity of the client in the initialization phase is primarily dominated by the size m of the Bloom filter. For each of the n_i elements in the client dataset, the client has to perform a constant amount of hash functions calls, and encrypt each of the m entries of the Bloom filter, leading to a complexity of $\mathcal{O}(n_i)$.

In the first and second stages, the client performs a constant number of computations for each of the n_t elements of the server dataset, leading to a complexity of $\mathcal{O}(n_t)$.

The server performs $\mathcal{O}(n_t \cdot t)$ hash computations in its first stage to compute the value $C_d^{i,j}$ for each client $i \leq t$ and each element $j \leq n_t$ in the server dataset. Furthermore, it performs $\mathcal{O}(n_t \cdot t)$ homomorphic additions.

The complexity of the server in stage 2 is dominated by the one in stage 1. Finally, in the third stage, it combines t decryption shares for each of the $j \leq n_t$ elements in the server dataset, resulting in a complexity of $\mathcal{O}(n_t \cdot t)$.

Chapter 4

MPSI Protocol (Miyaji, Nishida)

This chapter deals with the MPSI protocol presented in [2]. Similar to the new MPSI protocol of the previous chapter, this protocol also makes use of Bloom filters.

4.1 Description

The fundamental building blocks of the protocol are Bloom filters and threshold homomorphic PKE schemes. In this thesis we implemented the protocol using the threshold Paillier PKE, but other threshold schemes are possible as well. Let there be n parties involved: $P_1, ..., P_n$ where each party P_i holds a private dataset S_i of cardinality n_i. Additionaly, a dealer D is designated in order to reduce the computational complexity of the parties. Unlike in the new MPSI protocol in the previous chapter, the dealer is not to be confused with a party or a server that holds a dataset or discovers the set intersection.

The protocol follows a peer-to-peer architecture where every party will communicate with every other party and compute the set intersection.

4.1.1 Input

Each party P_i holds a public key pk and a shared decryption key sk_i of a threshold homomorphic PKE. The cardinality of each dataset n_i is not known to any individual party or dealer D.

4.1.2 Notation

The following notations are used in the protocol:

1. $\mathbf{BF}_{m,k}(S_i) = [\mathbf{BF}_i[0], ..., \mathbf{BF}_i[m-1]]$: Bloom filter on a set S_i

2. $\mathbf{IBF}_{m,k}(\cup S_i) = [\sum_{i=1}^{n} \mathbf{BF}_i[0], ..., \sum_{i=1}^{n} \mathbf{BF}_i[m-1]]$: integrated Bloom filter of n sets $\{S_i\}$, where $\sum_{i=1}^{n} \mathbf{BF}_i[j]$ represents the sum of all parties' array.

3. $\mathbf{IBF}_{m,k}(\cup S_i)\backslash l = [\sum_{i=1}^{n} \mathbf{BF}_i[0]-l, ..., \sum_{i=1}^{n} \mathbf{BF}_i[m-1]-l] (1 \leq l \leq n)$: l-subtraction from $\mathbf{IBF}_{m,k}(\cup S_i)$.

4.1.3 Sequence diagram

Figure 4.1 shows the sequence diagram of the protocol and describes the communication between an arbitrary party i and another party j, and the dealer D. The computational steps of each intermediate stage will be specified in detail below. The steps for each of the parties are identical and happen in parallel.

4.1.4 BF construction

Each party P_i where $1 \leq i \leq t$ computes the encrypted Bloom filter $Enc_{pk}(\mathbf{BF}_{m,k}(S_i))$ of their dataset S_i. This is done by first creating a new Bloom filter as in Algorithm 2. Then inserting each $e \in S_i$ to the Bloom filter as in Algorithm 3. Next, the obtained $\mathbf{BF}_{m,k}(S_i)$ is encrypted using the public key to obtain $Enc_{pk}(\mathbf{BF}_{m,k}(S_i))$ using Algorithm 6.

4.1.5 Dealer stage 1

1. Compute $Enc_{pk}(\mathbf{IBF}_{m,k}(\cup S_i)) = \prod_{i=1}^{n} \mathbf{BF}_{m,k}(S_i))$

2. Let r be a randomly chosen set of integers: $[r_0, ..., r_{m-1}] \in \mathbb{Z}_q^m$.

3. Compute $Enc_{pk}(r(\mathbf{IBF}_{m,k}(\cup S_i)\backslash \mathbf{n})) = (Enc_{pk}(\mathbf{IBF}(\cup S_i))) \cdot Enc_{pk}(-\mathbf{n}))^r$ where $Enc_{pk}(-\mathbf{n}) = [Enc_{pk}(-n), ..., Enc_{pk}(-n)]$

4.1.6 Party stage 1

The party computes its decryption shares:

1. Compute $\mathbf{sh}_i = [sh_i^0, ..., sh_i^{m-1}]$ where sh_i^j is the jth decryption share obtained after decrypting $Enc_{pk}(r(\mathbf{IBF}_{m,k}(\cup S_i)\backslash \mathbf{n}))$

4.1.7 Party stage 2

The party combines the other decryption shares it receives and computes the set intersection S:

1. Compute $\mathbf{d}^j = 1 \oplus Comb(sk, \mathbf{sh}_0^j, ..., \mathbf{sh}_n^j) \ \ \forall j$
   (The result is inverted, because an encryption of 0 should be decrypted to a 1)

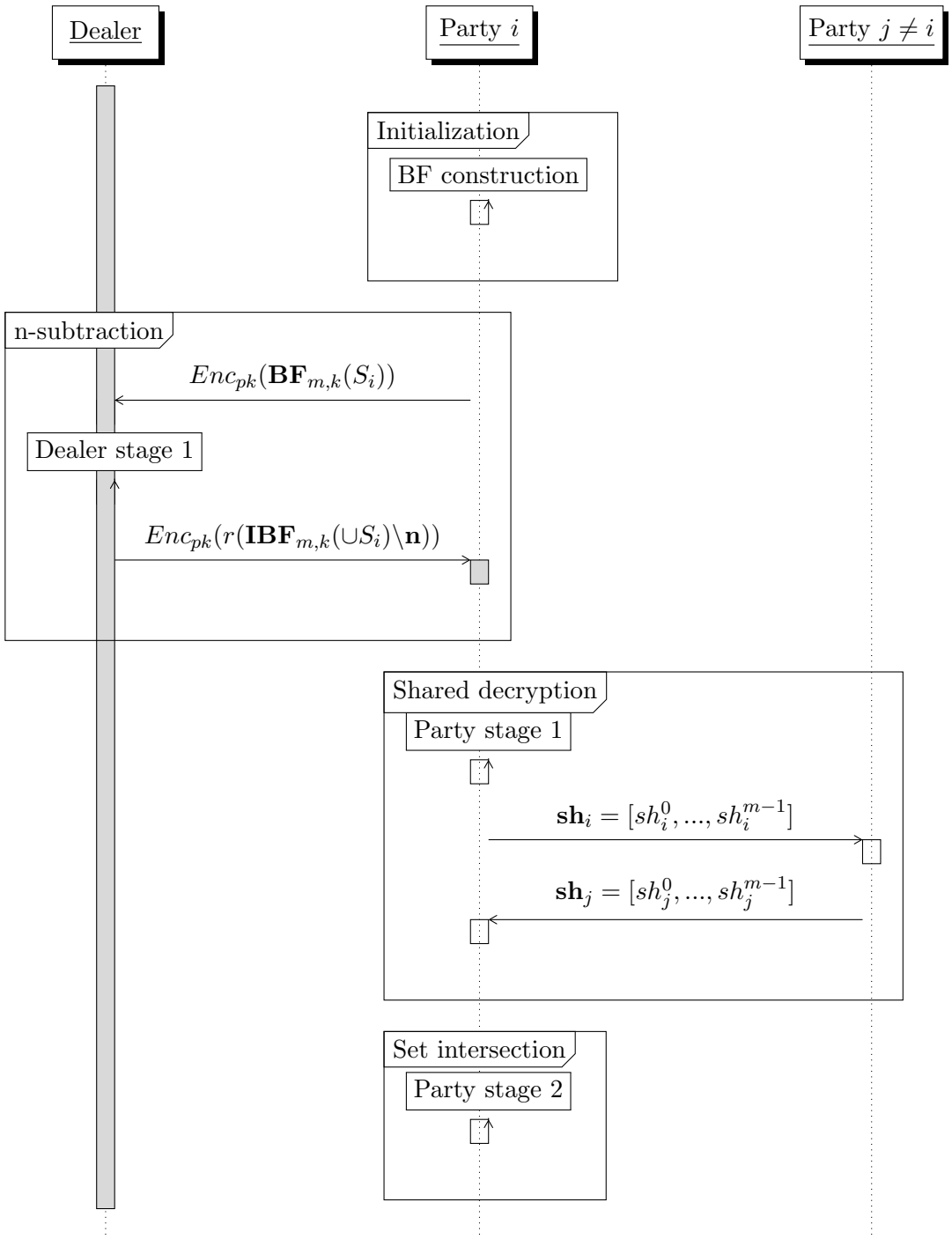

Figure 4.1: Miyaji & Nishida MPSI sequence diagram

2. Let $\mathbf{F}$ be a Bloom filter with the otained $\mathbf{d}^j$ values.

3. $S = \{y \mid y \in S_i \wedge Check(\mathbf{F}, y)\}$ where Check is the Bloom filter algorithm 4.

4.1.8 Remarks on protocol description and notation

The protocol has been divided into various party and dealer stages, which describe the computations done by the corresponding party. This makes it easier to identify the correspondence with the java implementation. Furthermore, in comparison to the original paper, the protocol has been translated into a sequence diagram representation to get a better overview and uniformity with the rest of this thesis.

4.2 Correctness

Given the homomorphic properties of the encryption scheme, the product $\prod_{i=1}^{n} \mathbf{BF}_{m,k}(S_i))$ will decrypt to a Bloom filter where all entries have been added up and thus corresponds with the Bloom filter of the set union $\cup S_i$. If an element y is in the set intersection, then all the entries mapped by the k hashes will be an encryption of n, and thus by performing the **n-subtraction**, the entries in the Bloom filter corresponding to the elements in the intersection will be an encryption of 0, which will be decrypted to 1, or a randomized value otherwise.

4.3 Theoretical complexity

4.3.1 Theoretical communication complexity

For simplicity, we do not make a distinction between sending or receiving communication complexity. The communication complexity is dominated by the size of the Bloom filter. Both the communication with the dealer and parties involve sending m ciphertexts where m is the size of the Bloom filter, which depends linearly on $|S_i|$.

The dealer sends and receives m ciphertexts to and from each party, hence the overall communication complexity of the dealer is $\mathcal{O}(|S_i| \cdot n \cdot S)$ where n is the number of parties and S is the size of a ciphertext.

The party also sends and receives m ciphertexts to and from each party, hence the overall communication complexity of the party is $\mathcal{O}(|S_i| \cdot n \cdot S)$.

4.3.2 Theoretical computational complexity

The computational complexity of the client in the initialization phase is primarily dominated by the size m of the Bloom filter. For each of the $|S_i|$

elements in the dataset, the client has to perform a constant amount of hash functions calls, and encrypt each of the m entries of the Bloom filter, leading to a complexity of $\mathcal{O}(|S_i|)$. The second stage of the party dominates its first stage: For each of the m entries of the Bloom filter, n decryption shares are combined. And then each of the $|S_i|$ elements are checked if they are in the decrypted Bloom filter, hence a total computational complexity of $\mathcal{O}(|S_i| \cdot n)$.

The computational complexity of the dealer is dominated by computing the product of m encrypted Bloom filter entries of n parties, thus a total complexity of $\mathcal{O}(|S_i| \cdot n)$.

Chapter 5

MPSI Protocol (Hazay, Venkitasubramaniam)

This chapter deals with the MPSI protocol presented in [3]. In comparison to the previous protocols, this protocol makes use of oblivious polynomial evaluation instead of Bloom filters.

5.1 Description

The fundamental building blocks of the protocol are oblivious polynomial evaluation and threshold homomorphic PKE schemes. In this thesis we implemented the protocol using the threshold Paillier PKE, but other threshold schemes are possible as well. Let there be n parties involved: $P_1, \ldots, P_n$ where each party P_i holds a private dataset X_i of cardinality m_i. The protocol follows a client-server architecture where parties $P_2, \ldots, P_t$ will be considered the clients and party P_1 will be considered the server. The final set intersection will be computed by the server.

5.1.1 Input

Each party P_i holds a public key PK and a shared decryption key SK_i of a threshold homomorphic PKE with decryption to zero algorithm $ShDec0$.

5.1.2 Sequence diagram

Figure 5.1 shows the sequence diagram of the protocol and describes the communication between an arbitrary party i and the server. The computational steps of each intermediate stage will be specified in detail below. The steps for each of the parties are identical and happen in parallel.

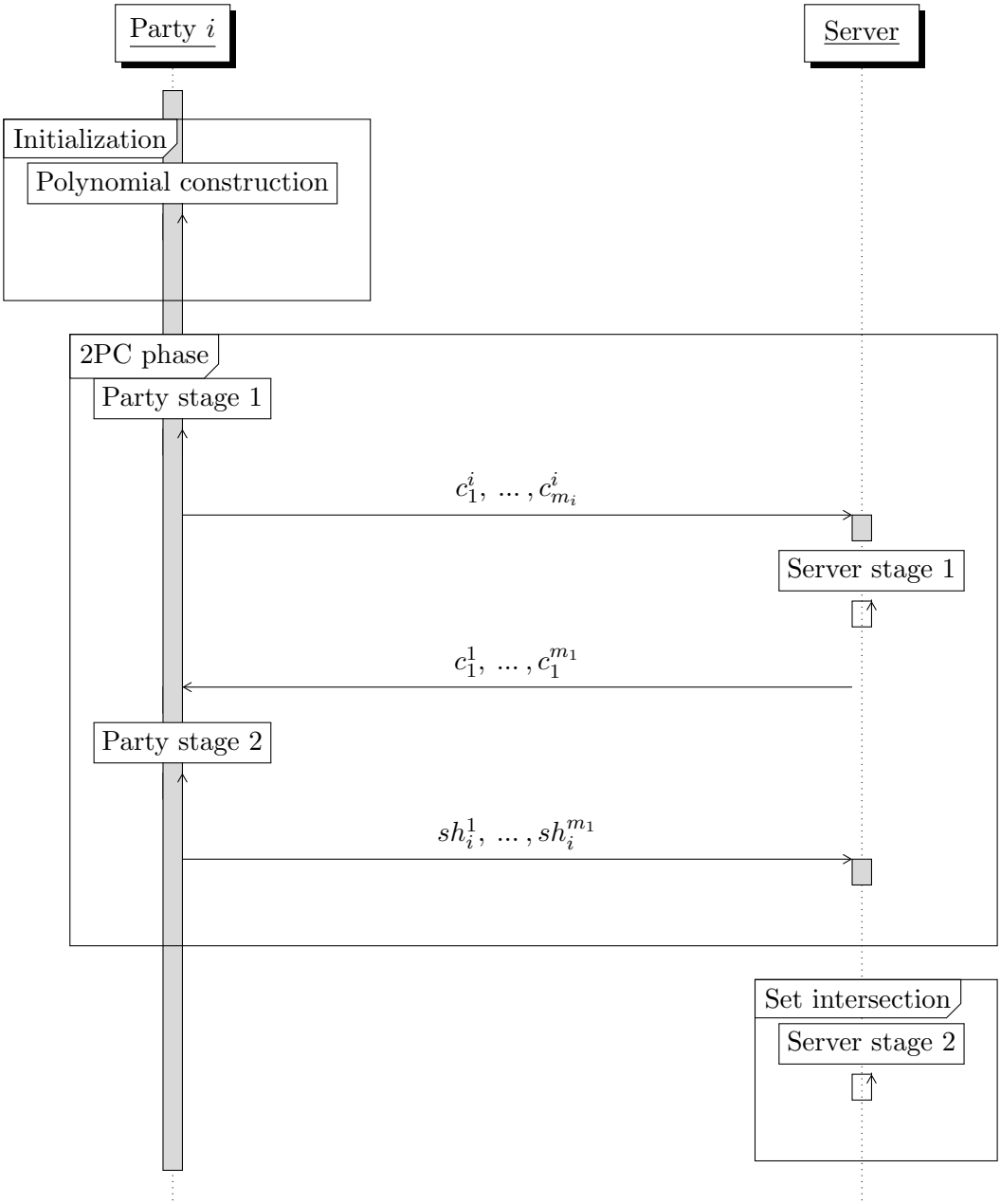

Figure 5.1: Hazay & Venkitasubramaniam MPSI sequence diagram

5.1.3 Polynomial construction

Each client P_i where $2 \leq i \leq t$ computes the coefficients of a polynomial $Q_i(\cdot)$ of degree m_i with roots set to the m_i elements of X_i.

5.1.4 Client stage 1

The client encrypts its coefficients:

1. Compute $c_j^i \ \ \forall j : j \leq |X_i|$

5.1.5 Server stage 1

1. Compute the combined polynomial $Q_1 = Q_2(\cdot) + ... + Q_n(\cdot)$:
   $c_1 = \prod_{i=2}^{n} c_1^i, ..., c_{mMAX} = \prod_{i=2}^{n} c_{mMAX}^i$ where $m_{MAX} = max(m_2, ..., m_n)$
2. Let r_i be a random value where $0 \leq i \leq m_1$
3. Compute for each $x_1^i \in X_1$ the randomized polynomial evaluations:
   $c_1^1 = r_1 \cdot Q_1(x_1^1), ..., c_1^{m_1} = r_{m_1} \cdot Q_1(x_1^{m_1})$

5.1.6 Client stage 2

The client computes its decryption shares:

1. Compute $sh_i^1 = \ \ ShDec(SK_i, c_1^1), ..., sh_i^{m_1} = \ \ ShDec(SK_i, c_1^{m_1})$

5.1.7 Server stage 2

The server combines the decryption shares it receives and computes the set intersection S:

1. Compute $d_1^1 = \ \ Comb(PK, sh_2^1, ..., sh_n^1), ..., d_1^{m_1} = \ \ Comb(PK, sh_2^{m_1}, ..., sh_n^{m_1})$
2. $S = \{x_1^j \mid d_1^j = 0\}$

5.1.8 Remarks on protocol description and notation

Like the other protocols, the protocol has been divided into various client and server stages, which describe the computations done by the corresponding party. This makes it easier to identify the correspondence with the java implementation. Furthermore, in comparison to the original paper, the protocol has been translated into a sequence diagram representation to get a better overview and uniformity with the rest of this thesis. Some steps from the protocol have been made more clear and explicit, because they were explained somewhat vague in the original paper. For example, the actual steps of the modified $(\pi_{FNP}^1, \pi_{FNP}^2)$ protocol mentioned in the paper are not made explicit.

5.2 Correctness

Correctness follows from the fact that if x_1^i is in the intersection, $Q_j(x_1^i)$ evaluates to 0 for all j. More specifically, $Q_1(x_1^i)$ evaluates to 0, because it is the combined polynomial where all coefficients are added up. Therefore, $r_i \cdot Q_1(x_1^i)$ evaluates to 0 if x_1^i is in the intersection or evaluates to a random value otherwise.

5.3 Theoretical complexity

5.3.1 Theoretical communication complexity

For simplicity, we do not make a distinction between sending or receiving communication complexity.

The client sends m_i ciphertexts to the server and receives m_1 ciphertexts from the server, hence the communication complexity is $\mathcal{O}(max(m_1, m_i) \cdot S)$ where S is the size of a ciphertext.

The server's communication complexity is dominated by receiving m_i ciphertexts from each client P_i, thus the communication complexity is $\mathcal{O}(m_{MAX} \cdot n \cdot S)$ where n is the number of clients and $m_{MAX} = max(m_2, ..., m_n)$.

5.3.2 Theoretical computational complexity

The initialization phase of the client consists of constructing a polynomial $Q_i(\cdot)$ of degree m_i with the roots set to the m_i elements. For this $\mathcal{O}({m_i}^2)$ polynomial multiplications are needed. The first stage of the client requires $\mathcal{O}(m_i)$ encryptions and the second stage requires $\mathcal{O}(m_1)$ decryptions. Hence, the client has an overall computational complexity of $\mathcal{O}({m_i}^2 + m_1)$.

The computational complexity of the server involves combining all polynomials $Q_i(\cdot)$, requiring $\mathcal{O}(m_{MAX} \cdot n)$ multiplications. And the $Q_1(\cdot)$ evaluation of m_1 elements, requiring a number of multiplications and exponentiations linear in $\mathcal{O}(m_{MAX} \cdot m_1)$. This leads to the overall computational complexity of $\mathcal{O}(m_{MAX} \cdot n + m_{MAX} \cdot m_1)$

Chapter 6

Implementation

In this chapter we will describe the details and design considerations of the implementations and data structures. The complete java implementation can be downloaded at [10] along with a documentation.

6.1 Protocol structure

Figure 6.1 shows the structure of the java implementation. The way that the protocol is implemented can be seen as a simulation. Instances of the server and clients are made and they are run and analyzed sequentially, while in a real-world application they would run in parallel. This however does not change the behaviour of the program. The controller manages the simulation. It drives the server and client stages and handles the network communication. The Server and Clients implement the actual computations done at the various stages.

6.2 Data structure

6.2.1 Bloom filter

Our Bloom filter implementation is kept as simple as possible to get the best possible performance. The **Bloomfilter** class implements all the algorithms

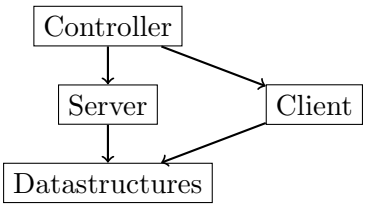

Figure 6.1: Implementation structure

described in section 2.5 with the exception that inversion and encryption are combined into a single method.

For hashing we use the standard Java $HashCode()$ implementation [11] for $Strings$ which computes the hash of a string s as

$$s[0] * 31^{n-1} + s[1] * 31^{n-2} + ... + s[n-1]$$

In order to compute the kth hash of an element e we chose to compute the hash of $k\|e$, that is the concatenation of the two. This way we need only one hash function.

While this is a very efficient way of hashing, it is not cryptographically secure. However, considering that the purpose of hashing in the context of the Bloom filter is not security, it is acceptable. Its main requirement is performance, and having an output of only 32 bits, collisions are possible.

Note that the 3 protocols analyzed in this thesis use the same hash function implementation, hence the choice of a hash function will not influence the performance differences between them.

6.2.2 Oblivious Polynomial Evaluation

The **Oblivious Polynomial Evaluation** (OPE) class provides methods for hashing arbitrary elements, interpolating a polynomial and evaluating a polynomial.

The elements represent the roots of the polynomial in $\mathbb{Z}_N$ where N is the public key of the encryption scheme. We use the same hash function for OPE as for Bloom filters, the only difference is that the result is computed modulo N. This makes it possible for the polynomial to represent any type of element.

The coefficients of the resulting polynomial representing a data set are obtained by computing a polynomial $P(x) = \prod_{i=0}^{k}(x - r_i)$ where r_i corresponds to the hash of the ith element.

In order to reduce the computational overhead of polynomial evaluation, we chose to apply Horner's rule in our implementation, which states that

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_0 = ((a_n x + a_{n-1})x + ...)x + a_0$$

This reduces the number of multiplications needed for polynomial evaluation from $2n - 1$ to n where n is the degree of the polynomial.

6.3 Threshold Paillier library

Our choice for the implementation of a threshold Paillier PKE is the *Paillier Threshold Encryption Toolbox* [12] created by the University of Texas at Dallas Data Security and Privacy Lab. The packages provide a Java implementation of the threshold variant of the Paillier encryption scheme as laid out in [4].

The library provides packages for Paillier encryption with thresholding, public and private key structures including key generation and non-interactive zero knowledge proofs.

The classes we are interested in are as follows:

- **PaillierThresholdKey** and **PaillierPrivateThresholdKey**

  Data structures for public and private keys

- **KeyGen**

  Generates the desired number of keys given a threshold, the number of bits for the prime factor of n and a random seed.

- **PaillierThreshold**

  Main Paillier class implementing encryption and threshold decryption

- **PartialDecryption**

  Data structure representing a partial decryption

One thing that we encountered while implementing the protocols was that the library did not have any functionality for ciphertext exponentiation. However, this was easily added to our own implementation.

6.4 Measurement framework

Our measurement framework drives the simulations given a certain range of parameters. It measures the execution time of specific phases of the protocol and keeps track of the sent and received data by each party. This way we can perform accurate measurements and compare the protocols as fairly as possible.

A simplified class structure of the framework is shown in figure 6.2. It shows the basic components and the interaction between them.

The core of this framework is the **Measurement** class. This class controls and runs the various MPSI **Implementations** with specific parameters obtained from the **Main** class. These parameters include a set of dataset sizes and number of parties to use for the tests. It calls the **DatasetGenerator** to obtain a randomized set for each party. After all tests are carried out, the **Measurement** class obtains the measurement results from the **Performance** and **Network** classes and stores them in the **ProtocolStats** for each protocol.

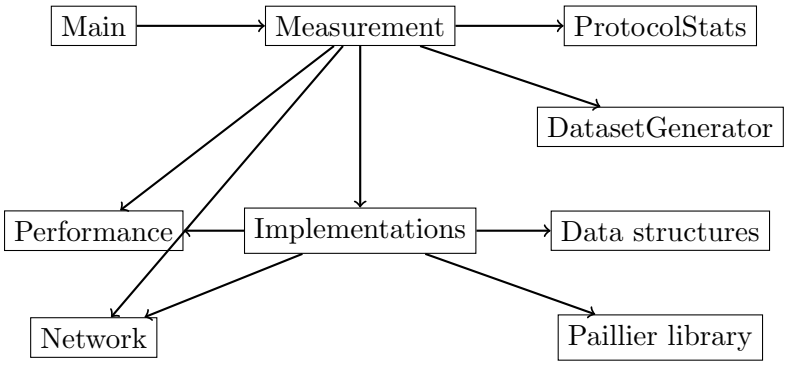

Figure 6.2: Framework structure

Chapter 7

Analysis

In this chapter we present the results of our performance analysis of the MPSI protocols in terms of computational and communication complexities. The results have been measured with the help of our measurement framework. Using the measurement results we do a comparison of the protocols and identify their main advantages and disadvantages.

7.1 Vehicle tracking

Our example of vehicle tracking will be applied in our comparisons. Each smart road will represent a party and holds its own dataset of vehicles. In our example, this dataset will consist of a list of strings representing the license plates of the vehicles. We wish to find those vehicles that have been registered with a set of smart roads, e.g. we compute the set intersection of the sets of vehicles for each road. We will perform measurements with varying number of roads and number of vehicles per road. The license plate of each vehicle will be represented as a randomly generated string that is 10 characters long.

7.2 Protocol comparison

Table 7.1 provides a comparison of the properties of the protocols. It shows the main properties for each protocol, such as which party learns the final set intersection, whether the size of the sets is being kept private, what network model is used, etc. The computational complexity (both for the initialization and interactive part) and the communication complexity have also been included.

	New MPSI		M&N MPSI		H&V MPSI	
	Client	*Server*	*Party*	*Dealer*	*Client*	*Server*
Learns intersection	✗	✓	✓	✗	✗	✓
Own dataset	✓	✓	✓	✗	✓	✓
Set size privacy	✓	✗	✓	-	✗	✗
Network model	client-server		peer-to-peer		client-server	
Security model	semi-honest		semi-honest		semi-honest	
Data structure	Bloom filter		Bloom filter		Polynomial	
Computational complexity (Initialization)	$\mathcal{O}(n)$	-	$\mathcal{O}(n)$	-	$\mathcal{O}(n^2)$	-
Computational complexity (Interactive)	$\mathcal{O}(n)$	$\mathcal{O}(nt)$	$\mathcal{O}(nt)$	$\mathcal{O}(nt)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2 + nt)$
Communication complexity	$\mathcal{O}(nS)$	$\mathcal{O}(nSt)$	$\mathcal{O}(nSt)$	$\mathcal{O}(nSt)$	$\mathcal{O}(nS)$	$\mathcal{O}(nSt)$

n is the set cardinality, t is the number of parties, S is the size of a ciphertext.

Table 7.1: Protocol comparisons

We put great effort in comparing the 3 MPSI protocols as fairly as possible. In doing so, we can reason about the different performances objectively and accurately, as they do not depend on different implementations or possible unknown variables that might affect performance. All of the 3 protocols are implemented in Java and are using the same Paillier encryption library. Also, the 3 protocols are simulated using the same simulation and measurement framework described in the previous chapter.

As the implementations are not optimized, paralellized or written in highly efficient languages such as C, the results that will be presented in this chapter do not represent the best possible results for each of the protocols. Furthermore, the level of optimization that we use in our implementation can affect the performance of each protocol differently.

The measurements are from simulated executions of the protocol under ideal circumstances and do not take extra overhead from real-world measurement results into consideration. For instance, this overhead could include extra network traffic.

7.3 Test criteria

For our comparisons we have a number of test criteria. Our measurements will be done for different number of parties and different number of set

elements. For instance, while we increase the number of parties according to the step size, we set the number of set elements constant to the default value. The minimum and maximum values, the step size and the default value for the variable parameters used in the measurements can be found in table 7.2. The constant parameters used can be found in table 7.3. For the communication measurements, the amount of bytes sent and received are combined and shown together. The results shown in this chapter have been obtained by running the measurement framework on an Intel i7-9750H with 16 GB of DDR4-2666 RAM.

Variables	**Minimum**	**Maximum**	**Step size**	**Default value**
Number of parties	5	95	2	5
Set size	500	10000	250	500

Table 7.2: Measurement variables

Constants	**Value**
Prime bits[1]	60
Randomness bits[2]	100
Element length	10 bytes
Intersection size	3 elements
Bloom filter FP rate	2^{-50}

[1] Number of bits required for the prime factor of n (public key)
[2] Number of bits required for random factors or exponents used in the protocols

Table 7.3: Measurement constants

7.4 Computational analysis

7.4.1 Measurement results

Figure 7.1 shows the measurement results of the total execution time from initialization to final computation of the set intersection with varying set sizes. The graph shows the execution time for each protocol and for both the server and the client (dealer and party). Figure 7.2 and figure 7.3 show the execution times for the initialization phase and interactive phase, respectively. Figure 7.4 shows the results of the total execution time with varying number of parties. We also included the measurements of the new MPSI protocol and the H&V protocol for a larger number of parties (see figure 7.5). Figure 7.6 and figure 7.7 show the execution times for the initialization phase and interactive phase, respectively. All measurement results correspond to an individual party. For some protocols, the corresponding plot can abruptly end. This is due to the fact that the computational complexity of that protocol grows much faster than the other protocols and we decided to stop the measurements of the protocol at that point. Also note that the servers and dealer are not found in figure 7.2 and figure 7.6, because they do not have an initialization phase. In the further course the Miyaji & Nishida protocol will be denoted as M&N, and the Hazay & Venkitasubramaniam protocol will be denoted as H&V.

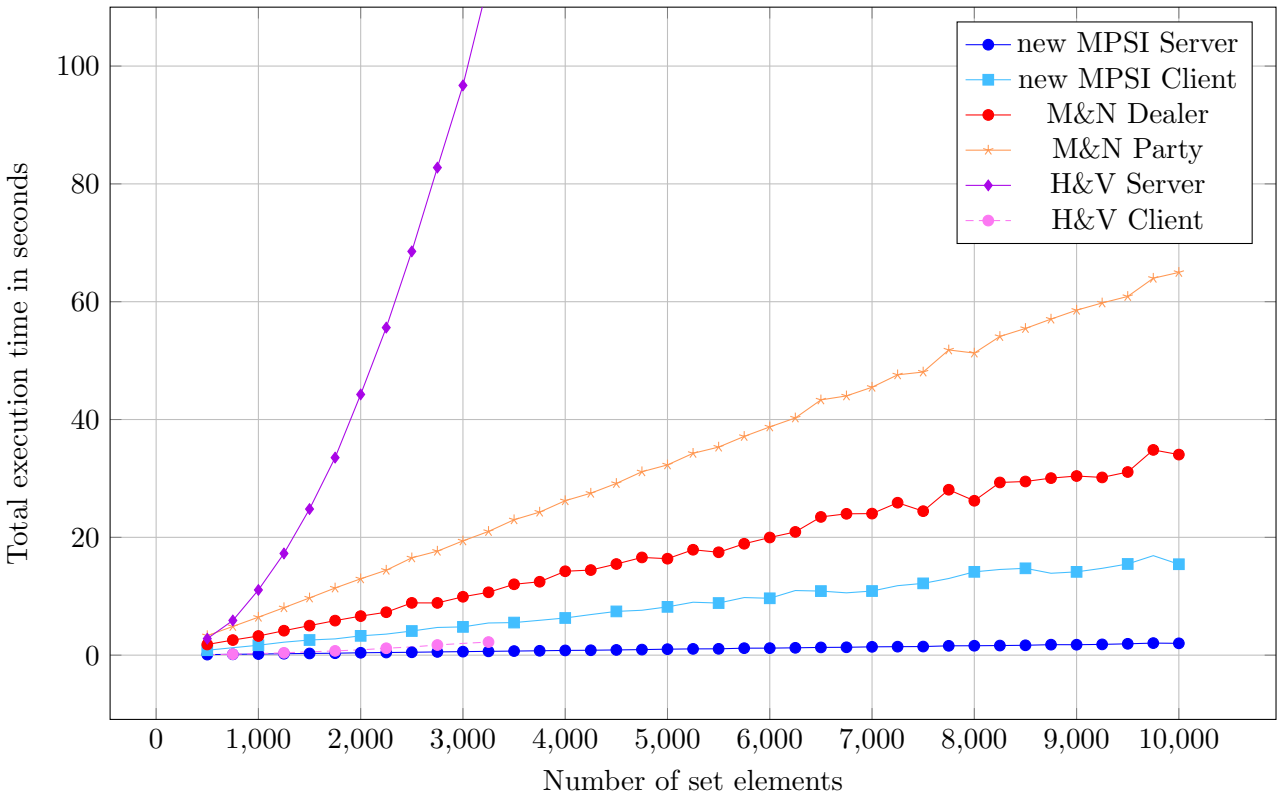

Figure 7.1: Total computation time with varying set sizes

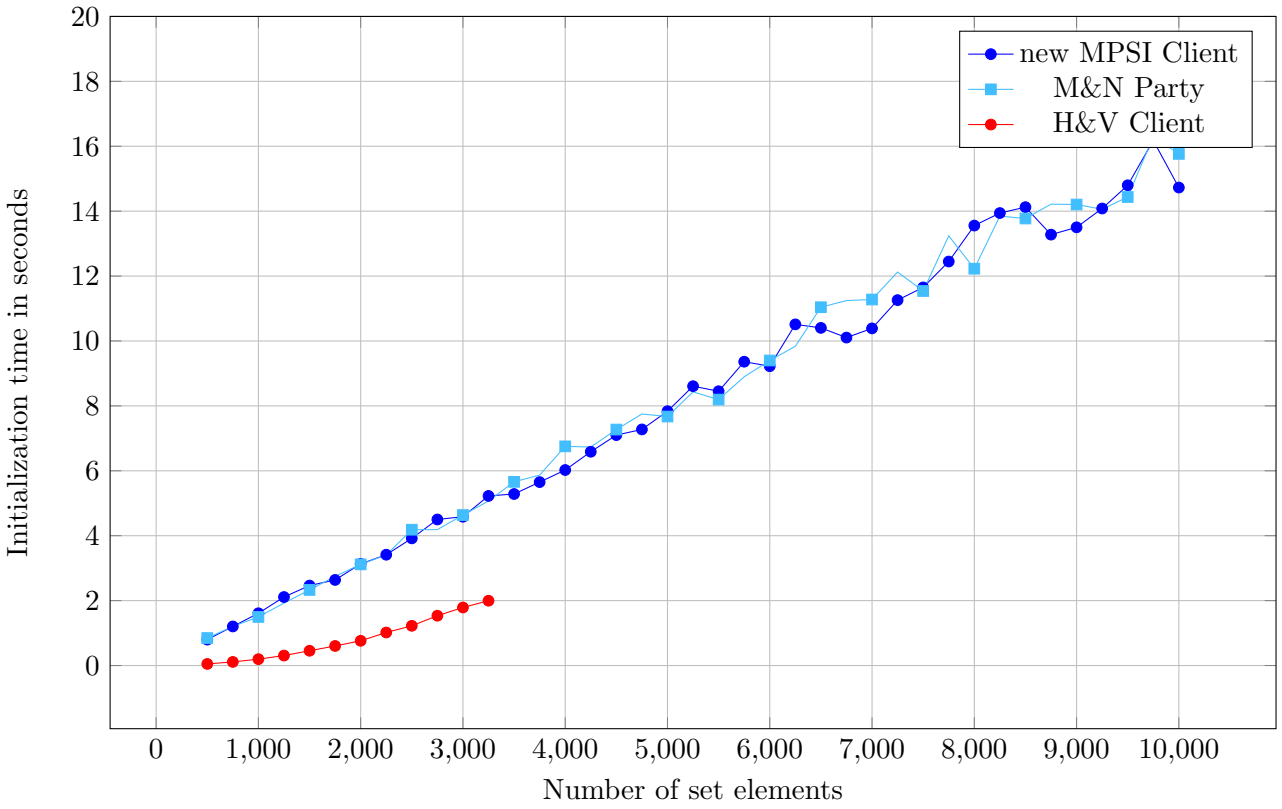

Figure 7.2: Initialization computation time with varying set sizes

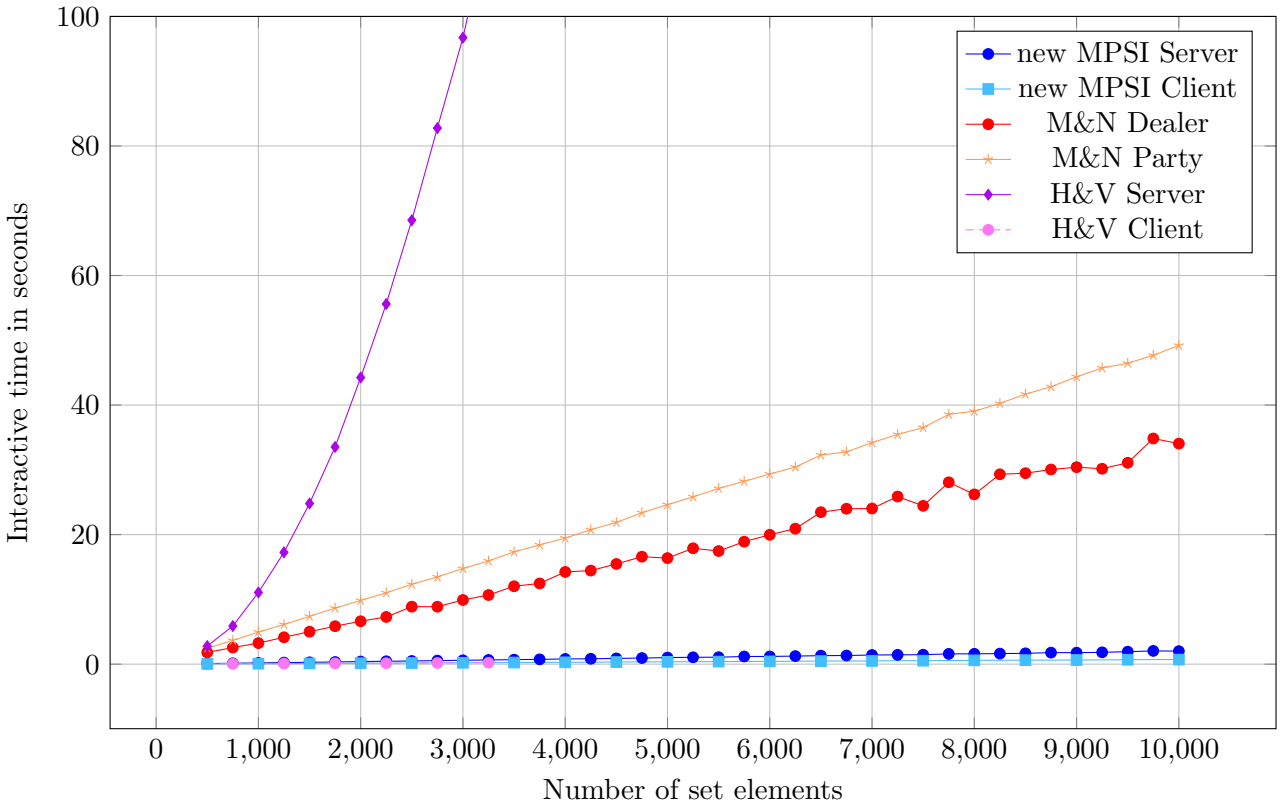

Figure 7.3: Interactive computation time with varying set sizes

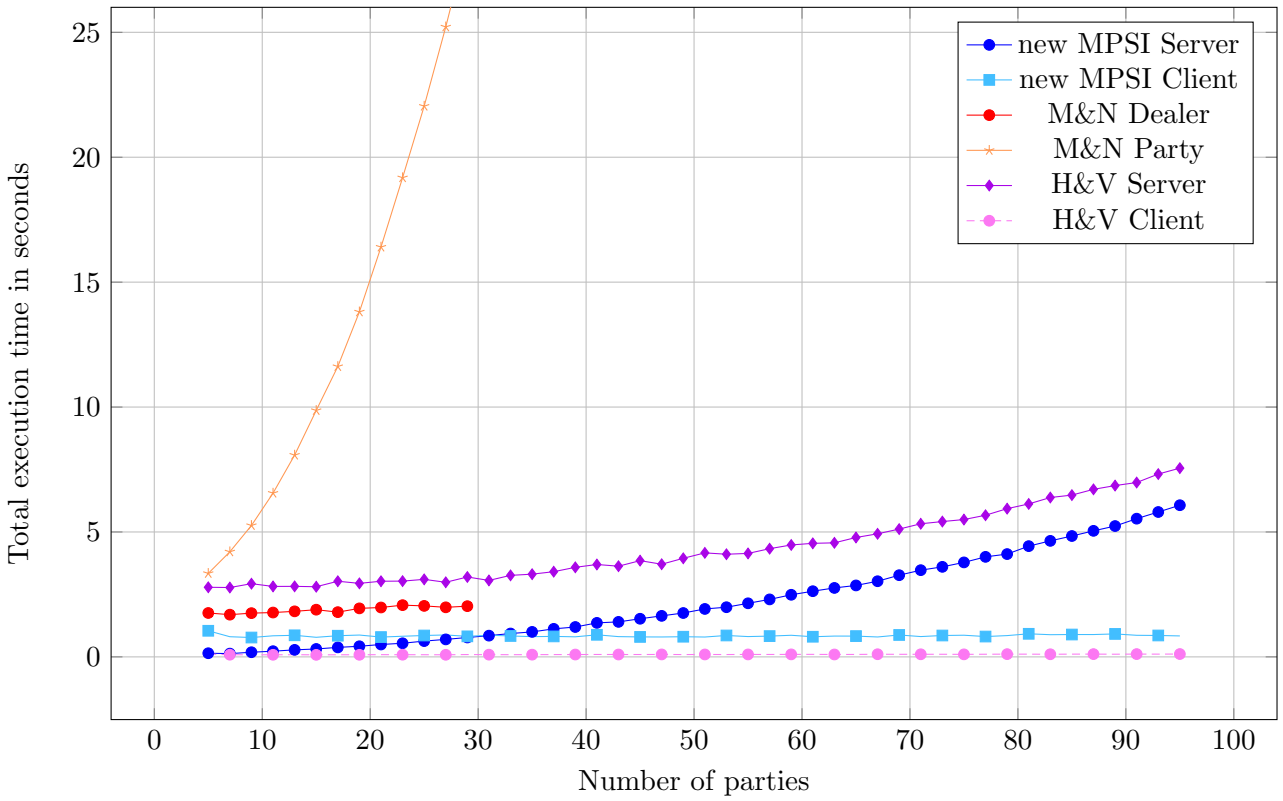

Figure 7.4: Total computation time with varying number of parties

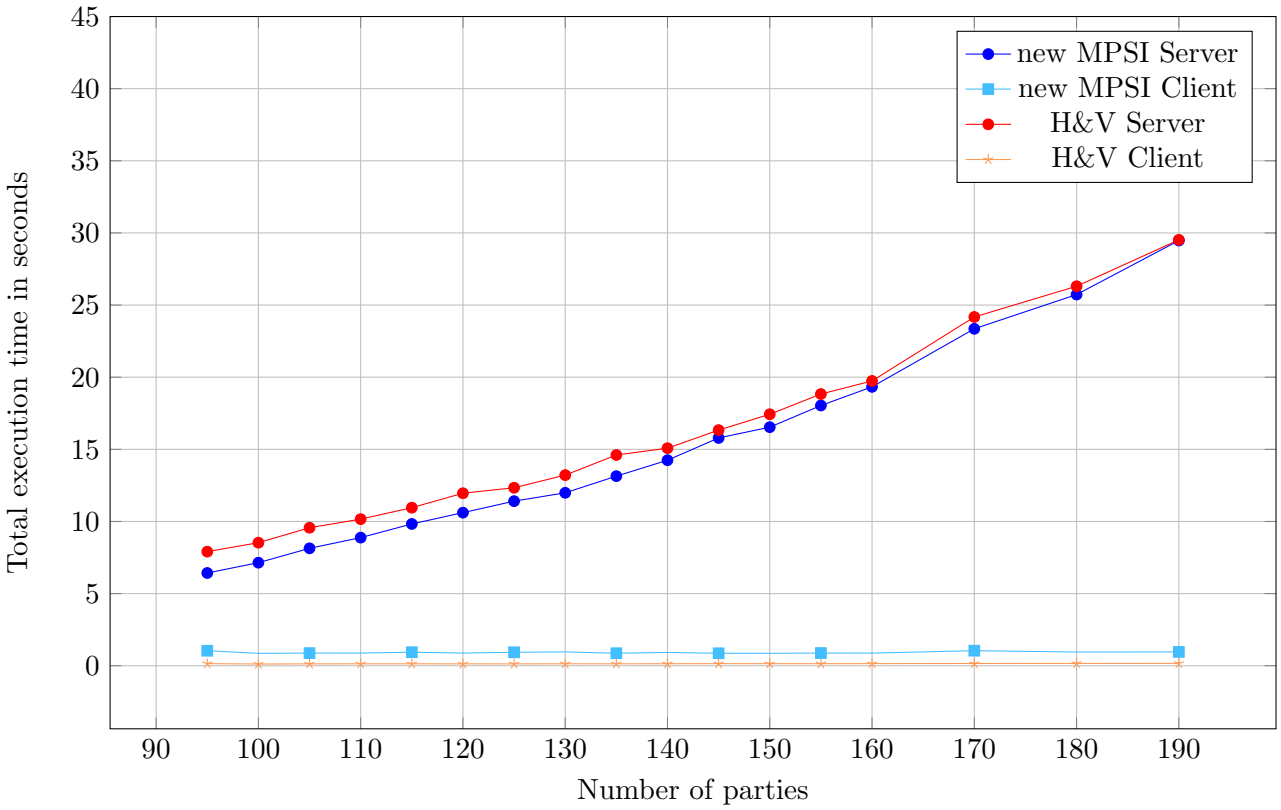

Figure 7.5: Comparison between new MPSI and H&V server with extended number of parties

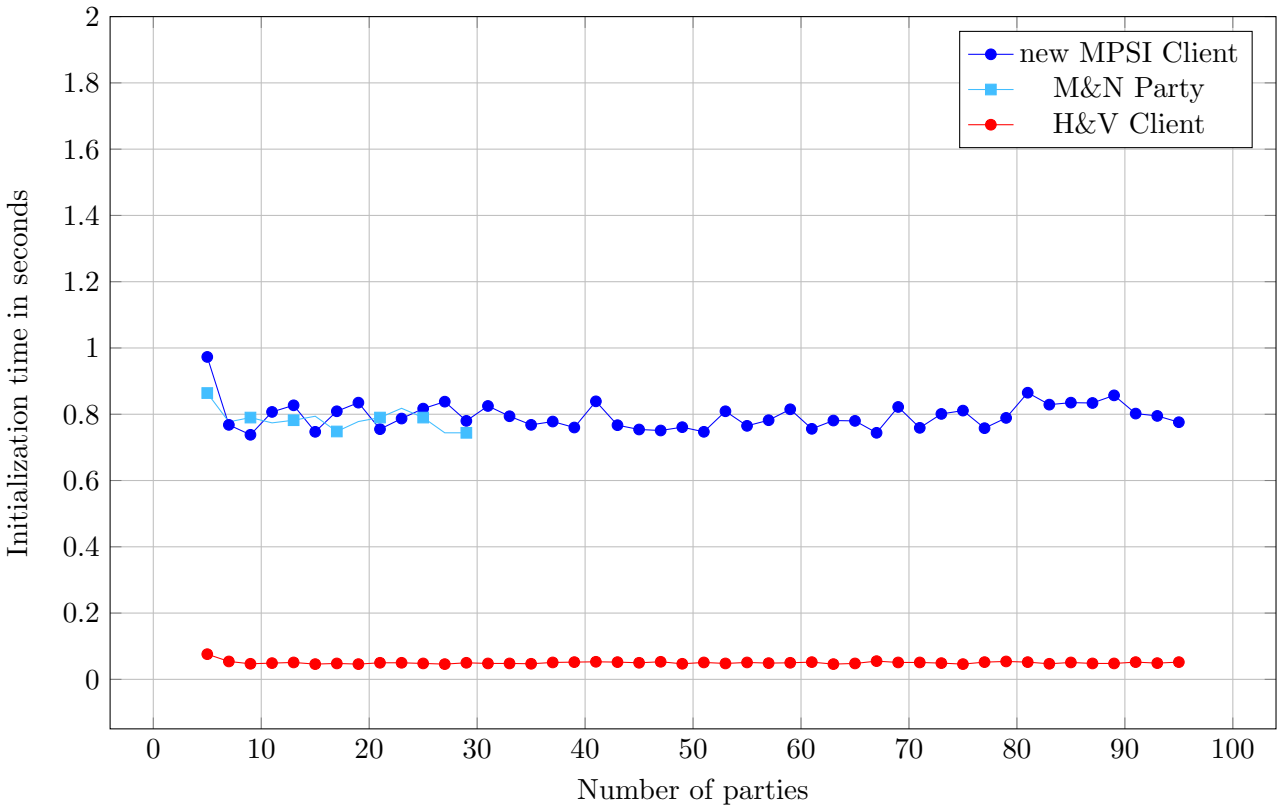

Figure 7.6: Initialization computation time with varying number of parties

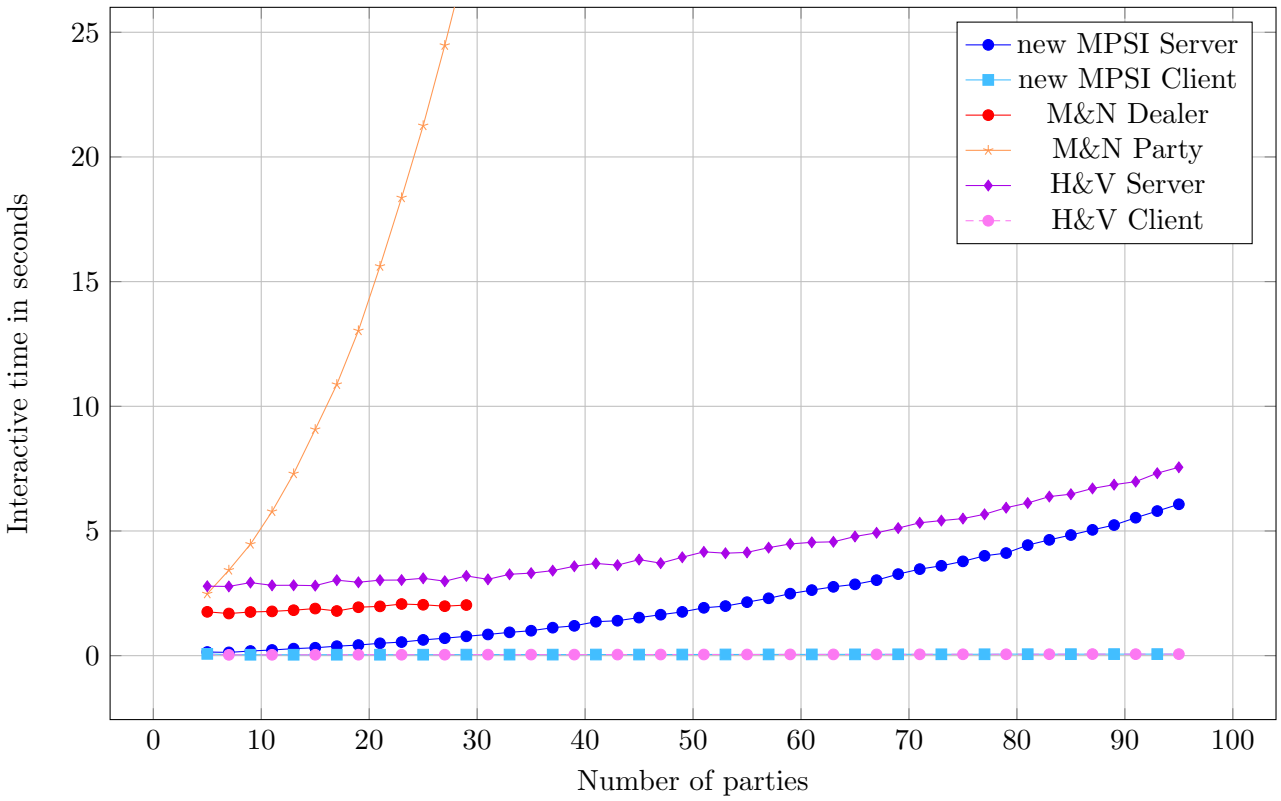

Figure 7.7: Interactive computation time with varying number of parties

7.4.2 Discussion

Varying set size

In this section we will analyse and discuss the previously obtained measurement results. When comparing the execution times in figure 7.1, we can immediately see that the H&V server grows relatively fast for increasing set sizes while the other protocols grow more linearly. The execution time of the H&V client remains very low, but due to the high server execution time necessary, the protocol will not be preferred for large set sizes. This already became evident in the theoretical quadratic time complexity of the protocol. The reason for this high growth in execution time is due to the polynomial evaluation, which has a quadratic time complexity in the set size. When comparing the new MPSI with M&N, we notice that the new MPSI has the lowest execution time for both the client and server. This difference is noticable when comparing the execution time of the new MPSI and M&N in the interactive part as can be seen in figure 7.3, where it is very low for the new MPSI protocol. The execution time during the initialization (see figure 7.2) part is similar, because both protocols compute and encrypt the Bloom filter here. The H&V protocol performs better in the initialization phase. This can be explained based on the fact that the Bloom filter will always be much larger than the actual size of the dataset. Overall, we can observe that the total execution time of the new MPSI server is remarkably low compared to the other servers.

Varying party size

We will now discuss the measurement results with varying number of parties as shown in figure 7.4. We observed that the execution time of the M&N party grows relatively fast for increasing number of parties, making the protocol not very suitable for applications with a high number of parties. We also see that the new MPSI server and the H&V server have superlinear growth in execution time. We found out that this is due to the fact that the complexity of the combining algorithm of the decryption shares is quadratic in the number of parties. The explanation for the more amplified growth of the M&N Party in comparison to the new MPSI server and H&V server is that in the M&N Party the combining algorithm is called for each entry of the Bloom filter. This becomes especially a problem when the false-positive rate is low and thus the Bloom filter size is high.

For lower number of parties, the new MPSI server performs slightly better than the H&V server. However, we extended the measurements for those two protocols to 190 parties and noticed that the plots of both servers grow increasingly closer (see figure 7.5). Unsurprisingly, the execution times of the new MPSI and H&V client are constant for different number of parties, because they do not depend on it. The plot of the M&N Dealer might

appear constant on first sight, but it does depend on the number of parties. However, these appear to be very lightweight computations, which is the reason why the execution time only grows very slow. The initialization time is constant for all 3 protocol clients/party (see figure 7.6). Once again, the protocols based on Bloom filters take more time to initialize.

All in all, we can conclude that the H&V protocol is less suited for large datasets and the M&N protocol is less suited for large numbers of parties. The new MPSI protocol performs the best when it comes to increasing set sizes and performs roughly as well as the H&V protocol when it comes to increasing party sizes.

7.5 Communication analysis

7.5.1 Measurement results

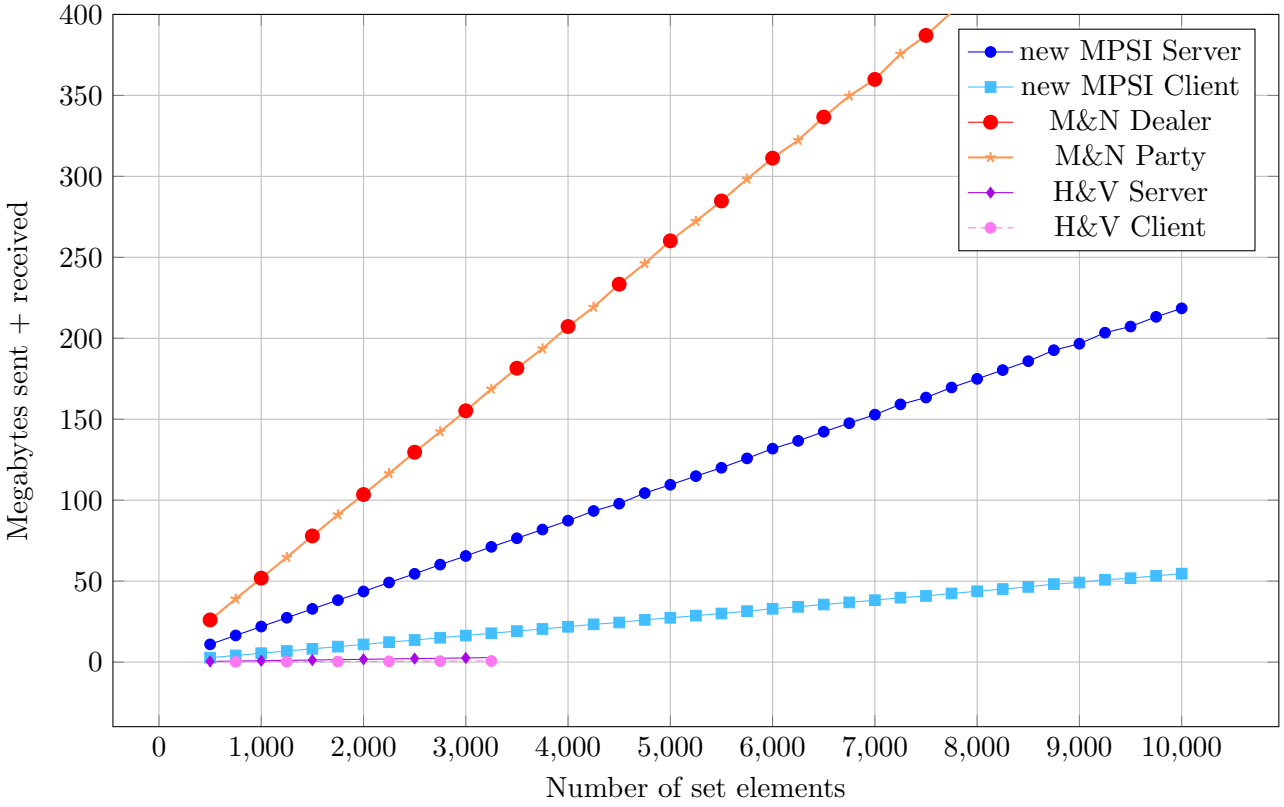

Figure 7.8: Communication with varying set sizes

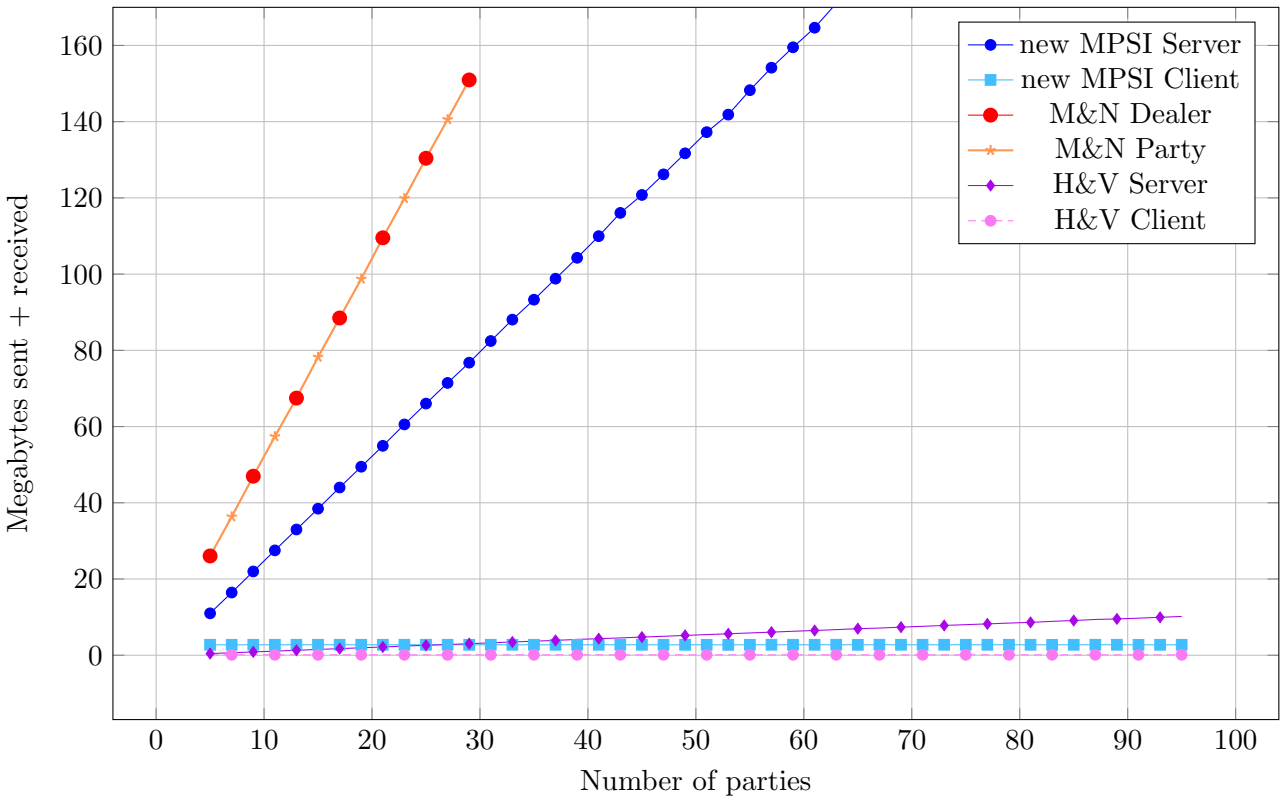

Figure 7.9: Communication with varying number of parties

7.5.2 Discussion

All three protocols show linear growth in the communicaton for both increasing set sizes and number of parties (see figure 7.8 and figure 7.9). Two exceptions are that figure 7.9 shows that the plots for the new MPSI client and H&V client remain constant for varying number of parties. This is not a surprise, because they do not depend on the number of parties directly as can be seen from the protocol specification. When we look at both results for varying set elements and number of parties, we notice that the protocols based on Bloom filters require a much higher communication. This makes the H&V protocol the most efficient when it comes to communication complexity. Out of the two Bloom filter based protocols, the new MPSI protocol requires less communication than the M&N protocol in both graphs. Moreover, we can observe that the required communication for the M&N protocol is identical for both the dealer and party, which is due to the peer-to-peer network model. On the other hand, the new MPSI client requires less network bandwidth than the server, especially as the number of parties increases.

7.6 Benchmark tests

In this section we perform 3 benchmark tests in which we set the same parameters for all 3 protocols and compare the differences in total execution time and communication required. Our benchmarks will be done with the same constants as in the previous measurements. The parameters are shown in table 7.4.

Test	**Party size**	**Set size**
1	10	1000
2	15	2000
3	20	3000

Table 7.4: Benchmark parameters

7.6.1 Measurement results

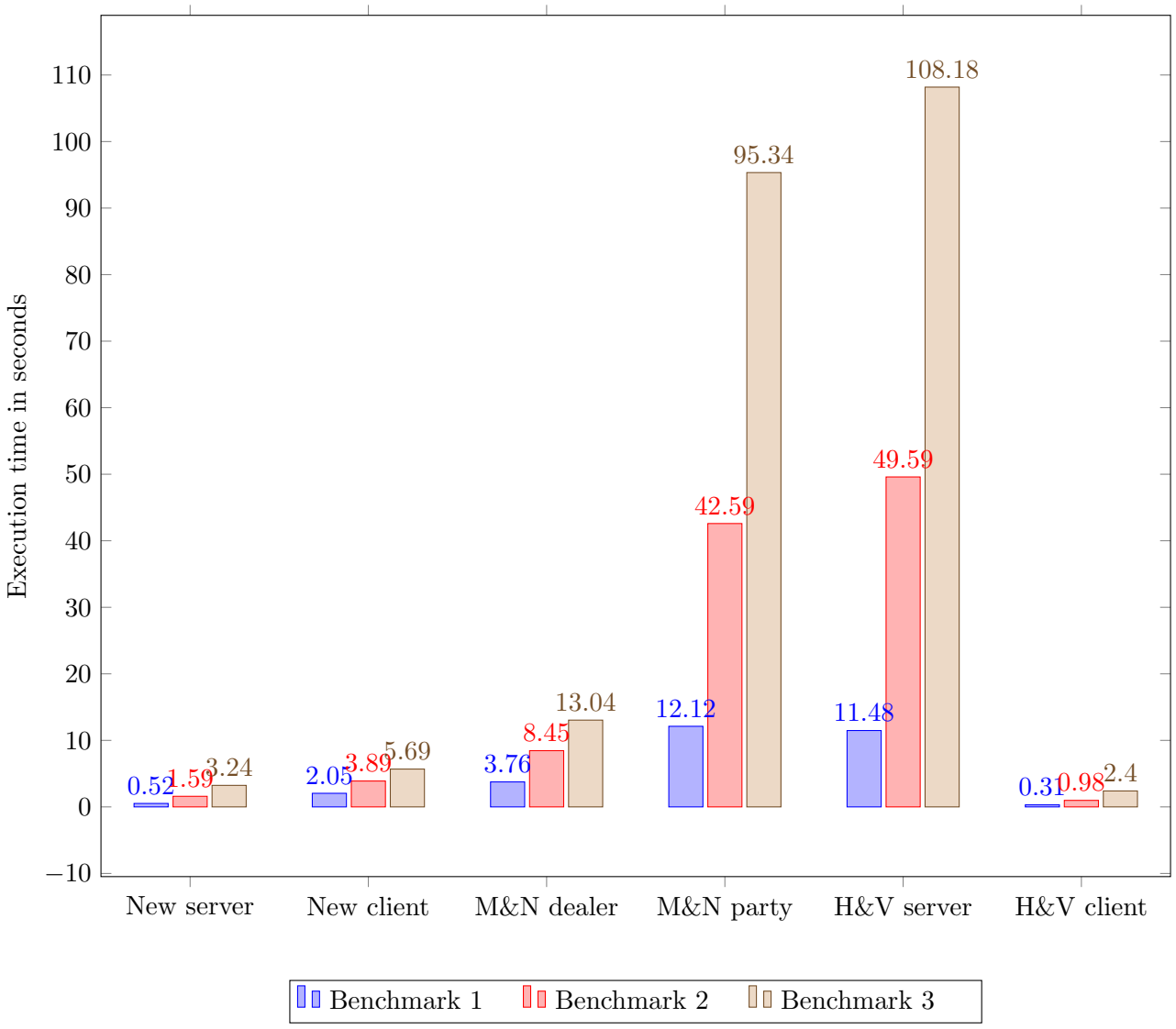

Figure 7.10: Computation benchmark results

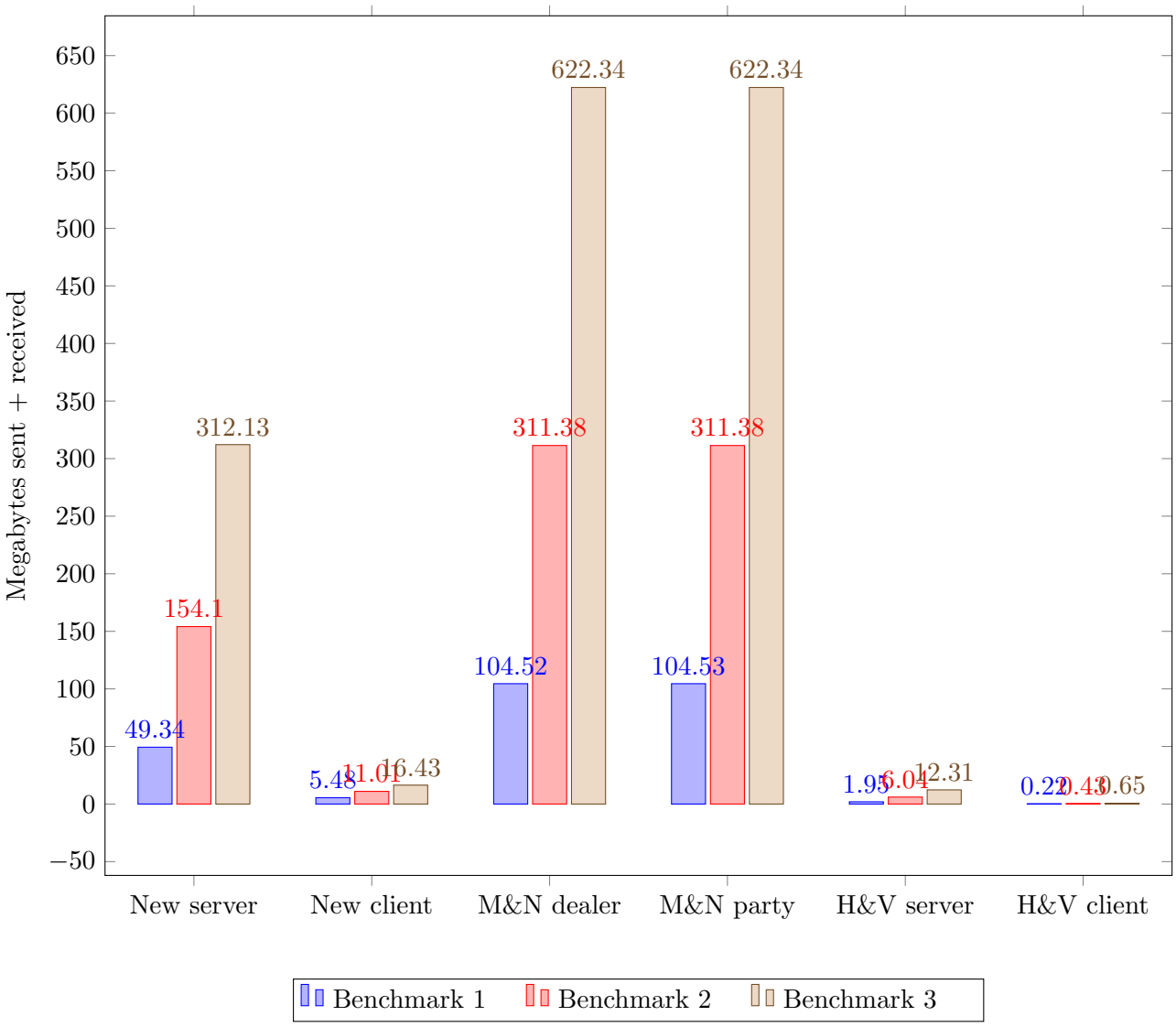

Figure 7.11: Communication benchmark results

		New MPSI Computational efficiency			
		Test 1	*Test 2*	*Test 3*	Average
M & N	*Dealer/Server*	+623%	+431%	+302%	+452%
	Party/Client	+491%	+995%	+1576%	+1021%
	Total	+518%	+831%	+1114%	+821%
H & V	*Server/Server*	+2108%	+3019%	+3239%	+8366%
	Client/Client	-85%	-75%	-57%	-72%
	Total	+359%	+823%	+1138%	+2320%

		New MPSI Communication efficiency			
		Test 1	*Test 2*	*Test 3*	Average
M & N	*Dealer/Server*	+112%	+102%	+99%	+104%
	Party/Client	+1807%	+2728%	+3688%	+2741%
	Total	+281%	+277%	+279%	+279%
H & V	*Server/Server*	-96%	-96%	-96%	-96%
	Client/Client	-96%	-96%	-96%	-96%
	Total	-96%	-96%	-96%	-96%

Table 7.5: Benchmark efficiency improvement of the newly proposed MPSI protocol compared to other protocols

7.6.2 Discussion

To wrap up this chapter and answer our initial research question: based on our results from section 7.4 and section 7.5, the newly proposed MPSI protocol is prefered over the other two MPSI protocols when it comes to computational complexity. This is especially the case when it comes to large datasets, where it clearly outperforms the other protocols. However, we observed that due to the nature of Bloom filters, the required communication will be higher compared than protocols based on oblivious polynomial evaluation.

In our final benchmark test results (see figure 7.5) we obtain that in terms of total execution time, the newly proposed MPSI protocol is about 821% more efficient than the M&N protocol and about 2320% more efficient than the H&V protocol. In terms of communication, we observed that the

newly proposed MPSI protocol is about 279% more efficient than the M&N protocol, but about 96% less efficient than the H&V protocol, based on the reasoning explained before. All in all, we conclude that in the general case, the newly proposed MPSI protocol is the preferred one out of the 3 as it scales well with both the number of parties and set sizes. However, it could be that in specific situations or protocol requirements, one of the other protocols might be preferred.

Chapter 8

Related Work

Over the last few decades, private set intersection has become an increasingly popular area in research and gradually more protocols for the multi-party case are being proposed. This chapter discusses relevant MPSI protocols found in the literature and briefly describes their characteristics.

Inbar, Omri and Pinkas propose a highly scalable MPSI protocol in [13] based on a variation of Bloom filters called a garbled Bloom filter. The protocol scales well with the number of parties and elements in the dataset, uses no cryptographic hardness assumptions and is based on the fact that the XOR of two garbled Bloom filters is a garbled Bloom filter of the intersection. The paper also shows performance measurement results based on their Java implementation of the protocol.

In [14] Kolesnikov, Matania, Pinkas, Rosulek and Trieu propose an MPSI protocol based on a new primitive, namely oblivious programmable pseudorandom functions. A Bloom filter-based, polynomial-based and a table-based construct of oblivious PRF are proposed which possess different trade-offs. The paper also provides a C++ implementation of their protocol.

Kissner and Song propose an MPSI protocol in [15] that is based on additive homomorphic encryption and oblivious polynomial evaluation.

In [16] Cheon and Jarecki propose an MPSI protocol that follows the basic idea of the protocol proposed in [15]. Unlike other protocols that require quadratic computational complexity when using the coefficient representation, this protocol achieves linear complexity by utilizing a new representation called point representation and operations over point representation.

Chapter 9

Conclusions

In this thesis we have implemented and analysed a newly proposed Bloom filter based MPSI protocol and two other relevant MPSI protocols. One of which is also based on Bloom filters while the other is based on oblivious polynomial evaluation. These protocols have been implemented in Java and their computational and communication performances were analyzed using our measurement framework. Our research question has been answered based on the measurement results that we analyzed. The findings of this analysis indicate that the newly proposed MPSI protocol performs better than the other two protocols in the general case. Our benchmark results show an average computational efficiency improvement of about 821% compared to the other protocol based on Bloom filteres and about 2320% compared to the protocol based on oblivious polynomial evaluation. The average communication efficiency improvement is 279% compared to the Bloom filter based protocol, but was much less efficient (96% decrease) compared to the protocol based on oblivious polynomial evaluation.

9.1 Future work

In future research we could look at some of the other available MPSI protocols found in the literature. Also, instead of using the Java programming language, one could focus on more optimized implementations such as C and analyze the best possible performances of the protocols one could achieve. In this thesis we focused solely on the Paillier encryption scheme, but other options are also possible, such as ElGamal encryption. Our performance criteria depended only on the execution time and communication required, but more factors can be taken into consideration, such as the total amount of memory used. Lastly, the protocols that we analyzed all assume the semi-honest security model. Future research could be spent analyzing extensions for the proposed protocols in the malicious setting.

Bibliography

[1] Asli Bay, Zekeriya Erkin, Jaap-Henk Hoepman, Simona Samardjiska. Efficient Multi-party Private Set Intersection Protocols.

[2] Atsuko Miyaji, Shohei Nishida. A scalable multiparty private set intersection. *NSS 2015: Network and System Security*, pages 376–385, November 2015. https://doi.org/10.1007/978-3-319-25645-0_26.

[3] Carmit Hazay, Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. *PKC 2017: Public-Key Cryptography – PKC 2017*, pages 175–203, February 2017. https://doi.org/10.1007/978-3-662-54365-8_8.

[4] Ivan Damgård, Mads Jurik. A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System. *PKC 2001: Public Key Cryptography*, pages 119–136, June 2001. https://doi.org/10.1007/3-540-44586-2_9.

[5] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, April 1999. https://doi.org/10.1007/3-540-48910-X_16.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, July 1970. https://doi.org/10.1145/362686.362692.

[7] Moni Naor, Benny Pinkas. Oblivious Transfer and Polynomial Evaluation. *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of Computing*, pages 245–254, May 1999. https://doi.org/10.1145/301250.301312.

[8] Michael J. Freedman, Kobbi Nissim, Benny Pinkas. Efficient Private Matching and Set Intersection. *EUROCRYPT 2004: Advances in Cryptology - EUROCRYPT 2004*, pages 1–19, April 2004. https://doi.org/10.1007/978-3-540-24676-3_1.

[9] Alex Davidson, Carlos Cid. Efficient Private Matching and Set Intersection. *ACISP 2017: Information Security and Privacy*, pages 261–278, May 2017. https://doi.org/10.1007/978-3-319-59870-3_15.

[10] Michael de Jong. Java source code. `https://github.com/mdejong10/mpsi`.

[11] String (Java Platform SE 7 Documentation) . `https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode()`.

[12] University of Texas at Dallas Data Security and Privacy Lab. Paillier Threshold Encryption Toolbox. `http://www.cs.utdallas.edu/dspl/cgi-bin/pailliertoolbox/`.

[13] Roi Inbar, Eran Omri, Benny Pinkas. Efficient scalable multiparty private set-intersection via garbled bloom filters. *SCN 2018: Security and Cryptography for Networks*, pages 235–252, August 2018. https://doi.org/10.1007/978-3-319-98113-0_13.

[14] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, Ni Trieu. Practical Multi-party Private Set Intersection from Symmetric-Key Techniques. *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 1257–1272, October 2017. https://doi.org/10.1145/3133956.3134065.

[15] Lea Kissner, Dawn Song. Privacy-preserving set operations. *CRYPTO'05: Proceedings of the 25th annual international conference on Advances in Cryptology*, pages 241–257, August 2005. https://doi.org/10.1007/11535218_15.

[16] Jung Hee Cheon, Stanislaw Jarecki, Jae Hong Seo. Multi-Party Privacy-Preserving Set Intersection with Quasi-Linear Complexity. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences 2012 Volume E95.A Issue 8*, pages 1366–1378, August 2012. https://doi.org/10.1587/transfun.E95.A.1366.