A PRACTICAL STUDY OF PSI PROTOCOLS: ANALYSIS, IMPLEMENTATION, AND EVALUATION WITH PROTOCOL ADAPTATIONS

EMIL LAMBAA LASSEN, 202206053
FREDERIK HVIDBJERG MIKKELSEN, 202208759
LASSE BJØRNHOLT HANSEN, 202208248

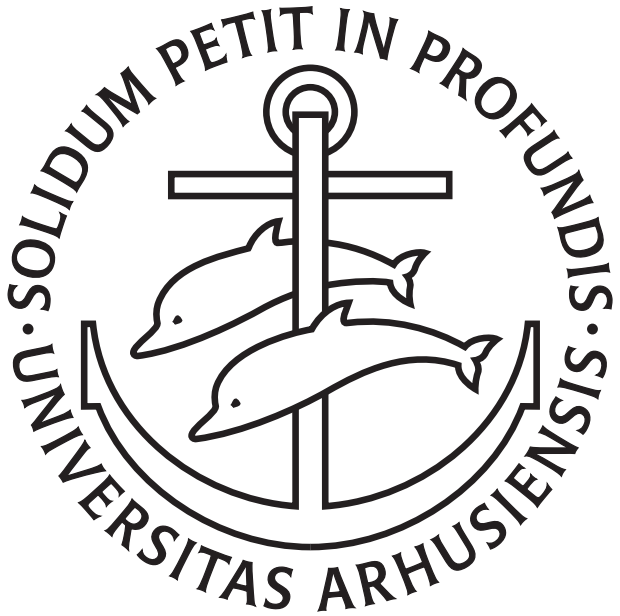
CRYPTOGRAPHY AND SECURITY

BACHELOR REPORT (15 ECTS) IN COMPUTER SCIENCE
Department of Computer Science, Aarhus University
June 9 2025
Advisor: Sophia Yakoubov
Supervisor: Aron Niels van Baarsen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

A PRACTICAL STUDY OF PSI PROTOCOLS: ANALYSIS, IMPLEMENTATION, AND EVALUATION WITH PROTOCOL ADAPTATIONS

HANSEN, LASSEN, MIKKELSEN



Bachelor report (15 ECTS) in Computer Science

Department of Computer Science
Faculty of Natural Sciences
Aarhus University

9 June 2025

Hansen, Lassen, Mikkelsen: *A Practical Study of PSI Protocols: Analysis, Implementation, and Evaluation with Protocol Adaptations,*
Bachelor report (15 ECTS) in Computer Science © 9 June 2025

CONTENTS

1 | ABSTRACT

This bachelor project presents a study and implementation of a Private Set Intersection (PSI) protocol. We begin with a literature review of [Rosulek and Trieu, 2021], examining the fundamental principles of a PSI protocol and the security mechanisms they employ. Here we have focused on understanding the properties of a PSI protocol, with focus on how to ensure malicious security. Following this, we describe the design and implementation of our own PSI protocol using classic Diffie-Hellman, rather than elliptic curve Diffie-Hellman (ECDH) approach used in [Rosulek and Trieu, 2021]. We then evaluate the runtime performance of our protocol and analyze how it differs from the protocol proposed in the article, considering both efficiency and security aspects. Finally, we discuss the trade-offs introduced by our design choices.

2 INTRODUCTION

Private Set Intersection (PSI) protocols enable two parties to compute the intersection of their private datasets without revealing any other information. This cryptographic primitive has become increasingly relevant in privacy-preserving applications such as contact discovery [Mezzour et al., 2009]. At the core of PSI lies a delicate balance between efficiency and security, where modern implementations often rely oblivious transfer (OT) extension [Pinkas, Schneider, and Zohner, 2014] and vector oblivious linear evaluation (VOLE) [Raghuraman and Rindal, 2022] to achieve both performance and strong privacy guarantees.

In this project, we present a study and implementation of a PSI protocol inspired by [Rosulek and Trieu, 2021] with a particular focus on ensuring the security properties this protocol provides. Our work begins with a literature review where we examine the PSI protocol proposed in the article and its underlying cryptographic assumptions. We then provide a detailed description of our own implementation, which deviates from the study's approach by using classical Diffie-Hellman. This choice was motivated by pedagogical and practical considerations, allowing us to explore the trade-offs involved in selecting different cryptographic primitives.

The implementation is followed by an experimental evaluation, where we measure runtime performance and analyze how the choice of cryptographic techniques impacts efficiency. Finally, we discuss the limitations of our approach, particularly in regard to the absence of elliptic curve cryptography (ECC) optimizations, hardware, and choice of programming language.

3 REVIEW OF LITERATURE

3.1 INTRODUCTION

Brief introduction to Private Set Intersection (PSI).
In Private Set Intersection (PSI), two parties each hold a private set of items denoted as X and Y, respectively. The goal is to compute the intersection $X \cap Y$ in a way that ensures neither party learns anything about the other's set beyond the common elements, with the goal of ensuring privacy-preserving comparisons of datasets.

Why PSI is important
Private Set Intersection (PSI) is important because it enables privacy-preserving data sharing across various real-world applications. Many scenarios require two parties to compare their datasets without revealing any additional information beyond the common elements.

An example of where this could be useful could be phone book apps. Here the app could allow users to check if any of their contacts are already on the platform. When a user grants access to their contact list, the app could compare it with its own existing user database to suggest connections. However, without privacy-preserving techniques like PSI, this process could expose sensitive information, such as the users complete contact list.

By using PSI, such can help users find contacts already on the platform while ensuring that neither party learns more than necessary.

Goal of the literature review
In this review, we take a look at the PSI protocol proposed in the article [Rosulek and Trieu, 2021] that works to improve the well-known Diffie-Hellman protocol, making it more efficient in smaller sets and achieving malicious security.

3.2 BACKGROUND ON PSI

General principles of PSI
There are different ways of approaching PSI, but the two most practical approaches include Diffie-Hellman and OT-extension based protocols. Here, Diffie-Hellman-based methods are particularly relevant to our work as the PSI protocol we are going to implement later is based on it.

Diffie-Hellman-based PSI leverages cryptographic key exchanges and hashing techniques to enable secure set comparison. In a typical construction, each party applies a randomized cryptographic function (such as modular exponentiation in a Diffie-Hellman setting) to their set elements. Since both parties perform operations that rely on the commutative property of modular exponentiation (i.e., $(g^a)^b \bmod p = (g^b)^a \bmod p$), they can independently compute a common value for elements in the intersection while preventing access to non-matching elements.

Categories of PSI protocols
The article discusses two variations of their protocol: one malicious and one semi-honest. The difference between these lies in the fact that a semi-honest PSI protocol assumes all parties

follow the protocol description, meaning an adversary could potentially exploit the protocol to infer additional information beyond what they were supposed to see. In contrast, a malicious PSI protocol protects against adversaries who may actively deviate from the protocol, manipulate inputs, or attempt to cheat. To counter such threats, malicious PSI protocols incorporate stronger cryptographic safeguards to ensure security.

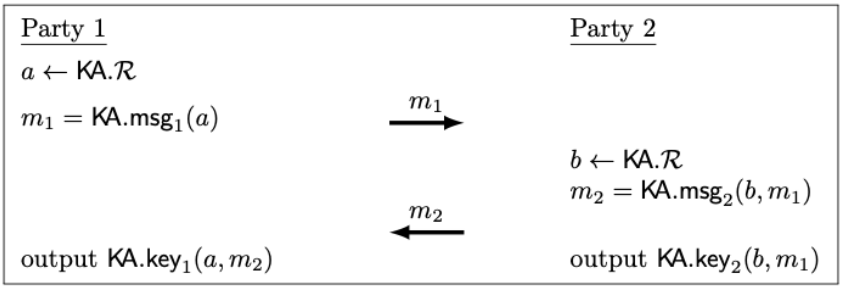
In our work, we focus solely on the malicious protocol, as this represents a more realistic and challenging case. This is also the approach used in the article [Rosulek and Trieu, 2021] where their improvements ensure security in real-world scenarios where adversaries cannot exploit weaknesses in the protocol.

3.3 REVIEW OF THE PSI PROTOCOL

Cryptographic techniques used

When implementing a PSI protocol, some cryptographic techniques are required in order to ensure security and functionality. In this section, we will look at what is needed in order to construct this maliciously secure PSI protocol.

Key agreement protocol
The PSI protocol is constructed on top of a 2-round key agreement protocol shown in the figure below.



The two-round key agreement protocol uses the following parameters:

- KA.R - which is the space of random bits shared among the two parties.
- KA.M - is the space of possible messages for Party 2.
- KA.K - is the space of possible output keys.

And a list of algorithms:

- KA.$msg_1(a)$
- KA.$msg_2(b)$
- KA.$key_1(a, m_2)$
- KA.$key_2(b, m_1)$

For this key agreement protocol to work in the proposed PSI protocol [Rosulek and Trieu, 2021], it is required that the space of possible protocol messages is in some finite field, and the key agreement protocol is required to satisfy the security definitions 1-4 described in [3.5].

Ideal permutation
In this protocol, all participants have access to a special black box (called an oracle) that applies a random but consistent permutation (denoted as Π) to any input from $\{0,1\}^n$. They can also use another oracle, which acts as the inverse of the first permutation, to undo the permutation (denoted as Π^{-1}). The relationship between Π and Π^{-1} is important because it ensures that any transformation applied by one party can be reversed by another. This is used for tasks like decryption, verifying data, or recovering original values in cryptographic protocols. It also plays a key role in security analysis, ensuring that the simulator can handle both forward and inverse queries consistently. This setup follows the ideal permutation model, which behaves as perfectly random and unpredictable transformations.

Random Oracles
The random oracles used are a theoretical function that behaves like a truly random function. It should provide random outputs for each unique input without giving any pattern to be detected. In the protocol, they use random oracles as hash functions:

$$H_1 : \{0,1\}^* \to \mathbb{F} \tag{3.1}$$
$$H_2 : \{0,1\}^* \times \mathbb{F} \to \{0,1\}^{2\kappa} \tag{3.2}$$

Here we have $\mathbb{F}$ as a finite field and the κ is equal to the bits of security used in the protocol.

Polynomial interpolation
For polynomial interpolation, meaning finding a polynomial that passes through predefined points.

The role of polynomial interpolation in the protocol is as follows:

Given a set of points

$$X = \{x_1, ..., x_n\} \quad \text{and} \quad Y = \{y_1, ..., y_n\}$$

we construct a polynomial P satisfying

$$P(x_i) = y_i$$

3.4 PROTOCOL

The PSI protocol described in [Rosulek and Trieu, 2021] builds upon a 2-round key agreement protocol KA (defined in Section 3.3), as well as access to an ideal permutation Π and its inverse Π^{-1} (Section 3.3). Additionally, both parties have access to two shared random oracles H_1 and H_2 (Section 3.3).

We describe the protocol in terms of an abstract sender and receiver, each holding an input set:

Sender: $X = \{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$
Receiver: $Y = \{y_1, \ldots, y_n\} \subseteq \{0,1\}^*$

The protocol proceeds as follows. The sender generates some random value $a \leftarrow KA.\mathcal{R}$ and computes the first message $m = KA.msg_1(a)$. The sender then sends m to the receiver.

For each element $y_i \in Y$, the receiver generates some random value $b_i \leftarrow KA.\mathcal{R}$ and computes a second-round message $m'_i = KA.msg_2(b_i, m)$. Each m'_i is then mapped through the inverse ideal permutation: $f_i = \Pi^{-1}(m'_i)$. The receiver now has pairs $(H_1(y_i), f_i)$ and constructs

a polynomial P of minimal degree that interpolates all these points. The receiver then sends P to the sender.

If the polynomial P has degree less than 1, the sender aborts. Otherwise, for each $x_i \in X$, the sender performs:

1. Compute $H_1(x_i)$ and evaluate the polynomial:

$$f_i = P(H_1(x_i))$$

2. Recover the corresponding second-round message:

$$m'_i = \Pi(f_i)$$

3. Compute the session key using the key agreement:

$$k_i = KA.key_1(a, m'_i)$$

4. Compute the masked key:

$$k'_i = H_2(x_i, k_i)$$

All k'_i values are added to a set K, which is then randomly shuffled and sent to the receiver.

For each $y_i \in Y$, the receiver computes $H_2(y_i, KA.key_2(b_i, m))$. If $H_2(y_i, KA.key_2(b_i, m)) \in K$, the receiver concludes that $y_i \in X \cap Y$.

Correctness.
Correctness follows from the key agreement protocol's correctness property, which guarantees that if $m = KA.msg_1(a)$ and $m'_i = KA.msg_2(b_i, m)$, then:

$$KA.key_1(a, m'_i) = KA.key_2(b_i, m)$$

Therefore, for any $x_i = y_i$, the sender and receiver compute the same shared key and consequently the same value under H_2:

$$H_2(x_i, KA.key_1(a, m'_i)) = H_2(y_i, KA.key_2(b_i, m))$$

This ensures that elements in the intersection $X \cap Y$ yield matching values in K, enabling correct intersection detection.

3.5 SECURITY, EFFICIENCY, AND SCALABILITY OF THE MALICIOUS PSI PROTOCOL

Security

For the malicious protocol to be secure, some security definitions must hold. Here the article argues the following security definition. The security definitions are written as in [**Rosulek and Trieu, 2021**].

- **Definition 1.** *A KA scheme is* ***correct*** *if, when executed honestly as shown in Figure 3, the two parties give identical output. In other words, for all* $a, b \in KA.R$.

$$\text{KA.key}_1(a, \text{KA.msg}_2(b, \text{KA.msg}_1(a))) = \text{KA.key}_2(b, \text{KA.msg}_1(a))$$

Essentially it should hold that given the correct input the protocol terminates with both parties outputting the same result following.

- **Definition 2.** *A KA scheme is* ***secure against an eavesdropper*** *if the following distributions are indistinguishable.*

$a, b \leftarrow \mathsf{KA}.\mathcal{R}$	$a, b \leftarrow \mathsf{KA}.\mathcal{R}$
$m_1 = \mathsf{KA.msg}_1(a)$	$m_1 = \mathsf{KA.msg}_1(a)$
$m_2 = \mathsf{KA.msg}_2(b, m_1)$	$m_2 = \mathsf{KA.msg}_2(b, m_1)$
$k = \mathsf{KA.key}_2(b, m_1)$	$k \leftarrow \mathsf{KA}.\mathcal{K}$
return (m_1, m_2, k)	return (m_1, m_2, k)

Figure 3.1: Security definition for a Key Agreement (KA) scheme against an eavesdropper. Taken from [Rosulek and Trieu, 2021].

This definition essentially states that a Key Agreement (KA) scheme is secure against an eavesdropper if the key k derived from an honest execution of the protocol is computationally indistinguishable from a key drawn randomly from the key space $KA.\mathcal{K}$.

In the figure, we see how the real key is constructed (the process on the left) using the $KA.key_2$ algorithm. On the right, we see a hypothetical key construction, where the key k is instead chosen randomly from the key space of all possible keys $KA.\mathcal{K}$.

The security definition holds as long as no adversary can distinguish, in probabilistic polynomial time, between the two cases when observing (m_1, m_2, k). If an adversary cannot differentiate between them, it means they gain no useful information about the key, ensuring the protocol remains secure.

- **Definition 3.** *A KA scheme is* ***non-malleable*** *if it is secure (in the sense of Definition 2) against an eavesdropper that has oracle access to* ***KA.key1(a, ·)****, provided the eavesdropper never queries the oracle on* m_2*. Formally, the following distributions are indistinguishable, for every PPT A that never queries its oracle on input* m_2.

$a, b \leftarrow \mathsf{KA}.\mathcal{R}$	$a, b \leftarrow \mathsf{KA}.\mathcal{R}$
$m_1 = \mathsf{KA.msg}_1(a)$	$m_1 = \mathsf{KA.msg}_1(a)$
$m_2 = \mathsf{KA.msg}_2(b, m_1)$	$m_2 = \mathsf{KA.msg}_2(b, m_1)$
$k = \mathsf{KA.key}_2(b, m_1)$	$k \leftarrow \mathsf{KA}.\mathcal{K}$
return $\mathcal{A}^{\mathsf{KA.key}_1(a,\cdot)}(m_1, m_2, k)$	return $\mathcal{A}^{\mathsf{KA.key}_1(a,\cdot)}(m_1, m_2, k)$

Figure 3.2: Security definition for a Key Agreement (KA) scheme that is non-malleable. Taken from [Rosulek and Trieu, 2021].

Let's imagine the communication between two parties, Alice (sender) and Bob (receiver). If Bob is corrupt and sends a manipulated second message, Alice (being honest) will still compute her key as if the protocol were followed correctly. Even though Bob might tamper with the message exchange, he will not learn anything useful from Alice's response, nor can he control or influence Alice's key in a meaningful way.

The core idea is that the adversary (potentially corrupt Bob) is given access to an oracle for key derivation, meaning he can compute certain keys based on manipulated messages.

However, Bob is restricted from querying the oracle on the exact honest message m_2 that Alice constructs. If Bob were able to query the oracle on m_2, he might gain an advantage by manipulating it to derive Alice's key. This restriction ensures that Bob cannot directly obtain Alice's key just by leveraging the oracle.

As shown in Figure 3.2, the security definition ensures that even if Bob sends a specific message, Alice will still execute the protocol and derive the key following Definition 2. This means that regardless of Bob's input, the output will still appear random and unpredictable to him. The security here follows the Diffie-Hellman Oracle assumption (DHO).

- **Definition 4.** *A KA scheme has* ***pseudorandom second messages*** *if m_2 is indistinguishable from random, even to someone who chooses m_1 adversarially. Formally, the following distributions are indistinguishable for all PPT A.*

$(view, \widetilde{m}_1) \leftarrow \mathcal{A}$ $b \leftarrow \mathsf{KA}.\mathcal{R}$ $m_2 = \mathsf{KA}.\mathsf{msg}_2(b, \widetilde{m}_1)$ return $(view, m_2)$	$(view, \widetilde{m}_1) \leftarrow \mathcal{A}$ $m_2 \leftarrow \mathsf{KA}.\mathcal{M}$ return $(view, m_2)$

Figure 3.3: Security definition for a Key Agreement (KA) scheme that has has pseudorandom second messages. Taken from [Rosulek and Trieu, 2021].

This definition says that a KA scheme has pseudorandom second messages if, for any PPT adversary A, the second message m_2 generated by an honest party is computationally indistinguishable from a uniformly random string of the same length, even when A adaptively chooses the first message m_1. Meaning that regardless of how an adversary tries to maliciously construct a message, the second message will never leak any information.

Efficiency (e.g., computational complexity, communication overhead)

The malicious PSI-protocol is best used in regards to Time vs. communication in small sets, meaning ≤ 1000 items. Even when comparing their malicious protocol to other semi-honest protocols their malicious protocol is 18-30% faster for n = 256 items (depending on the network speed) and uses 10% less communication than the next best (semi-honest) protocol.

Scalability (how well they perform with large datasets)

The protocol presented by [Rosulek and Trieu, 2021] is specifically optimized for small datasets, defined in their work as fewer than 1,000 items. Within this range, their approach outperforms existing PSI protocols in both computation and communication overhead. However, if dataset sizes grow beyond this threshold, the efficiency advantages may diminish, meaning this approach is only suitable for smaller sets.

3.6 CONCLUSION

Summary of key findings

The paper introduces an efficient and compact approach to maliciously secure Private Set Intersection (PSI), while being a Diffie-Hellman based PSI. Since this is a malicious PSI protocol, it is important that the security definitions hold as the protocol should still be secure and not

leak any information, even in the presence of a malicious party. This is usually the bottleneck for malicious protocols as computation time tends to suffer when introducing security measures, even then, the proposed protocol offers strong security guarantees while maintaining efficiency, making it a great example of a malicious PSI protocol.

One of the most significant advantages of this approach is its efficiency for small data sets. The protocol demonstrates performance improvements over even semi-honest PSI protocols in scenarios where the dataset size is limited. However, this efficiency does not extend as well to larger datasets, where other PSI techniques, particularly those based on OT extensions, may perform better. Another notable limitation is its dependency on network speed. For small datasets ($\leq$500 elements), the protocol consistently outperforms OT-extension-based approaches. However, for datasets approaching 1000 elements, its advantage diminishes, and it only remains superior under slow network conditions (e.g., 50 Mbps for both parties).

Despite these limitations, this work contributes to the field by providing a practical, maliciously secure PSI protocol optimized for small-scale applications. Future work could explore modifications to enhance scalability or investigate alternative optimizations that reduce reliance on network conditions while maintaining malicious security guarantees.

4 REVIEW OF OUR IMPLEMENTATION

4.1 INTRODUCTION TO OUR IMPLEMENTATION

When we began implementing the compact and maliciously secure private set intersection (PSI) protocol described in the paper, we quickly encountered a major obstacle: the original protocol relied heavily on elliptic curve cryptography (ECC) and Elligator mapping for encoding group elements. Due to the complexity of elliptic curves, and in agreement with our advisor, we decided to replace elliptic curve operations with modular arithmetic over a prime field, using Diffie-Hellman key exchange over large safe primes.

This change simplifies the mathematical tools required, making the protocol more accessible for implementation, though at the cost of reduced cryptographic efficiency and compactness. Despite this tradeoff, the core security guarantees of the PSI protocol, particularly those related to malicious security, are preserved by carefully adapting the structure and behavior of the protocol to fit within this new framework.

To ensure our implementation remains secure and functional under these changes, some modifications to the protocol were necessary. These adjustments include replacing group element encoding mechanisms and modifying the structure of cryptographic operations to align with modular arithmetic. The remainder of this section outlines the most notable of these changes, with detailed explanations provided throughout the report.

- **No Use of Elliptic Curves:** Since we do not use elliptic curves, our protocol instead relies on prime order cyclic subgroups of $\mathbb{Z}_p^*$ as in classical Diffie-Hellman key exchange.
- **Encoding of Group Elements:** To preserve the security condition that message elements $KA.msg_2$ (3.1) must be indistinguishable from uniformly random bit strings, we introduced a custom encoding scheme suited for modular arithmetic. This is necessary because we operate within a subgroup of $\mathbb{Z}_p^*$, which can be easily distinguished from the larger space.

Overall, we retained the same high-level structure of the original protocol, adapting specific steps where necessary to accommodate our use of standard number-theoretic tools. This approach allowed us to focus on implementing a secure and correct system without being hindered by the additional complexity of elliptic curve cryptography.

4.2 PACKAGES/LIBRARIES AND DEPENDENCIES

To implement our version of the cryptographic protocol and related functionality, we relied on a range of Python libraries. These libraries have been used for tasks such as cryptographic operations, network communication, polynomial computation etc.

- **Standard libraries** (`socket`, `struct`, `pickle`, `json`, `time`, `threading`, `random`, `string`, `fractions`, `sys`): Used for basic system operations, network communication, data encoding/decoding, and mathematical utilities.
- `typing`: Provides the Set type for our receiver and sender.

- **`Crypto (pycryptodome)`** (`AES`, `Random`, `Util.number`): Supports secure encryption, key generation, and number-theoretic functions.
- **`sympy`**, **`secrets`**, **`hashlib`**: Used for symbolic mathematics, secure random number generation, and cryptographic hashing, respectively.
- **`gmpy2`**: A C-coded Python extension module for high-performance multiple-precision arithmetic. Built on GMP, MPFR, and MPC libraries, it offers efficient support for integers, rationals, reals, and complex numbers with optimized memory and speed, ideal for cryptographic and mathematical computations.

4.3 SAFE PRIMES

In order to create our cyclic group, we will use *safe primes* to ensure the existence of a large prime-order subgroup, which is essential for cryptographic security in discrete logarithm-based protocols such as Diffie-Hellman. A **safe prime** is a prime number of the form $p = 2q + 1$, where q is also a prime. By doing this we ensure that the multiplicative group modulo p contains a large prime-order subgroup of size q.

4.4 CONSTRUCTING THE CYCLIC GROUP

Define $\mathbb{Z}_p^*$ and its order $p - 1 = 2q$.

$\mathbb{Z}_p^*$ is our cyclic subgroup, which is a multiplicative group of integer values modulo p, excluding 0. So this essentially means that we have:

- $\mathbb{Z}_p$ is the set of integers modulo p, i.e., the set of integers $\{0, 1, \ldots, p-1\}$
- $\mathbb{Z}_p^*$ refers to the subset of $\mathbb{Z}_p$ that consists of all the invertible elements (i.e., numbers that have a multiplicative inverse under modulo p which means they are coprime with p).

So what does it mean for two values to be coprime? Essentially, if the greatest common divisor between two values is 1, then they are coprime. For example.

$$gcd(8, 15) = 1$$
Coprime, as they only share the common factor 1
$$gcd(4, 8) = 4$$
Not coprime, as they also share the common factor 4

Now, for our cyclic subgroup $\mathbb{Z}_p^*$, we are only interested in the elements that are coprime with our prime p. Since p is a prime number, it is only divisible by 1 and itself. The only number in $\mathbb{Z}_p$ that is **not** coprime with p is 0, because 0 is divisible by every integer, including p. Therefore, all elements from 1 to $p-1$ are coprime with p, meaning our group is $\mathbb{Z}_p^* = \{1, 2, 3, \ldots, p-1\}$.

Thus, the **order** of $\mathbb{Z}_p^*$ (the number of elements in the group) is $p - 1$, which can also be written as $2q$ when p is a **safe prime** (i.e., $p = 2q + 1$ for some prime q).

Find a generator g such that the subgroup $G = \langle g^2 \rangle$ has order q.

When working with cryptography, we want to operate over structures that appear random and do not leak any information through patterns. This is exactly why we construct a new subgroup

from the full group $\mathbb{Z}_p^*$.

The reason why we do not want to perform Diffie-Hellman in $\mathbb{Z}_p^*$ follows from the eavesdropping case described in Definition 2 [3.1].

Assume we have an eavesdropper who receives either (g^a, g^b, g^{ab}) or (g^a, g^b, g^c), where c is sampled uniformly at random from $\mathbb{Z}_p$. Let a and b also be sampled uniformly at random from $\mathbb{Z}_p$. Then:

- a is even with probability approximately $1/2$.
- b is even with probability approximately $1/2$.
- Therefore, ab is odd only when both a and b are odd, which happens with probability approximately $1/4$.
- Thus, ab is even with probability approximately $3/4$.
- On the other hand, c (chosen uniformly) is even with probability $1/2$.

This means that when we are working with a generator g of the full group $\mathbb{Z}_p^*$, there is now an efficient way to distinguish between elements with even and odd exponents, since this translates to elements being squares or non-squares. An eavesdropper can now notice this difference in probability such that definition 2 [3.1] does not hold.

Instead in the group we are working with $G = \langle g^2 \rangle$, all the elements in this group are squares and so an eavesdropper can no longer distinguish between elements with an even or an odd exponent. This construction ensures that the Decisional Diffie-Hellman (DDH) assumption holds in the group $G = \langle g^2 \rangle$ thereby making sure definition 2 [3.1] holds.

To do this we will start with some $g \in \mathbb{Z}_p^*$ which will serve as a generator for our entire group. For the implementation to be secure and functional, we need to perform a few checks to ensure that the elements generated appear random and don't fall into small subgroups that could be easily attacked.

- $g \not\equiv 1 \mod p \rightarrow$ Prevents trivial generator. If $g = 1$, then everything generated is just 1, which we can't really use for anything.
- $g^2 \not\equiv 1 \mod p \rightarrow$ Ensures g doesn't collapse into a small subgroup. If $g^2 = 1$, then g is a square root of 1, which means $g = \pm 1 \mod p$, meaning we end up with a very small subgroup.
- $g^q \not\equiv 1 \mod p \rightarrow$ If this is true, then the order of g must be $2q$, since $g^{2q} \equiv 1 \mod p$ always holds in $\mathbb{Z}_p^*$ (because that's the group order), and $g^q \neq 1$ implies the order isn't q or a smaller divisor.

So if these checks hold, we know we will have a g of order $2q$. Now, we can use this g to generate our new subgroup $G = \langle g^2 \rangle$. Since we know g is an element of order $2q$ (i.e., $g^{2q} \equiv 1 \mod p$), the powers of g will cycle through all $2q$ elements of $\mathbb{Z}_p^*$.

By squaring g, we get g^2. The order of g^2 must divide the order of g (which is $2q$), as guaranteed by Lagrange's Theorem. This means the order of g^2 must divide $2q$, and the possible divisors are $1, 2, q, 2q$.

Now, because we know $g^q \not\equiv 1 \mod p$ (as checked earlier), the order of g^2 cannot be $2q$ or 1. Therefore, the only remaining possibility is q. Thus, we conclude that g^2 has order q, and it generates a cyclic subgroup of size q.

The subgroup $G = \langle g^2 \rangle$ is the cyclic group generated by g^2. This subgroup will have elements:

$$G = \{1, g^2, g^4, g^6, \ldots, g^{2(q-1)}\} \mod p$$

Since g^2 has order q, the subgroup G contains exactly q distinct elements. These elements are generated by the powers of g^2, and G forms a cyclic group of order q under multiplication modulo p.

Selecting a Random Element in the Subgroup
So now that we have our cyclic subgroup $G = \langle g^2 \rangle \subset \mathbb{Z}_p^*$ of order q, we can generate random elements using

$$x = (g^2)^r \mod p \quad \text{for } r \in [1, q]$$

Since g^2 has order q, this ensures each r maps to a unique non-identity element in G, giving us uniform randomness over the subgroup.

4.5 SECURITY IN OUR IMPLEMENTATION

The original PSI protocol [Rosulek and Trieu, 2021] work is designed to be secure against malicious parties, utilizing ECC and Elligator for efficient encoding of group elements. It ensures that:

- **Sender Privacy**: The sender does not learn anything in the execution of the protocol
- **Receiver Privacy**: The Receiver only learns the intersection and nothing more about the sender's set.
- **Malicious security**: The above privacy guarantees still hold if the parties deviate arbitrarily from the protocol

In contrast to the original protocol, our implementation does not rely on the Elligator mapping or ECC. Instead, we use classical Diffie-Hellman and custom encodings. As a result, some security aspects need to be looked over to determine whether our implementation can still guarantee the same level of malicious security.

Why is it secure?

In the article [Rosulek and Trieu, 2021] they prove that the PSI protocol has malicious security if the KA protocol satisfies definitions 1-4. We therefore argue that our implementation is also maliciously secure given we still satisfy the security definitions, with the changes we have made.

The security definitions can be found here: [3.5]

Security definition 1: This definition essentially says that given the protocol is executed honestly, then both parties should arrive at the same result. Given that we still follow the protocol proposed by Rosulek, M. and Trieu in regards to hashing and encrypting, the underlying structure has not changed; therefore, this definition should still hold.

Security definition 2: This definition says that a KA scheme is secure against an eavesdropper if an adversary cannot distinguish between two key distributions. As in the article, we also use Diffie-Hellman Key Agreement (DHKA) to establish a shared secret. The security of the hashed DHKA protocol against a passive eavesdropper (as discussed in Definition 2 [3.1]) is considered standard and relies on the assumption that the Computational Diffie-Hellman (CDH) problem is hard.

Security definition 3: For this definition, the question is whether our protocol satisfies the non-malleability property as described in Definition 3 [3.2]. Here the security follows the Oracle Diffie-Hellman assumption (ODH) just like in [Rosulek and Trieu, 2021], that says; in a protocol like Diffie-Hellman, even if an eavesdropper sees all public messages and has access to the random oracle, they cannot tell whether the session key is real or random. Thereby, following this assumption Definition 3 should hold.

Security definition 4: The mapping we've defined, where we map an element $x \in G$ to either $x \mod (p)$ or $u \cdot x \mod (p)$ with equal probability, ensures that the output is indistinguishable from a random string.

This mapping preserves the uniform distribution of elements in G, while introducing a random factor that prevents any discernible structure. Since the elements of G are uniformly distributed in $\mathbb{Z}_p^*$, the transformation either maps directly to $x \mod (p)$ or applies a randomization factor u (with $u^q \neq 1 \mod (p)$) to x.

By ensuring that $u^q \neq 1 \mod (p)$, we guarantee that multiplying x by u does not cause repetition or cycle through a limited set of values. If $u^q \equiv 1 \mod (p)$, we would just stay inside of G when multiplying with u, thereby not adding any additional randomness. In contrast, when $u^q \neq 1 \mod (p)$, multiplying x by u ensures that the transformation maps to uniformly random elements in $\mathbb{Z}_p^*$.

Thus, after applying the encoding, random elements of G are indistinguishable from random integers in [1,p], since $KA.msg_2$ is generated as g^b where b is sampled uniformly random in [1,q], giving us a uniformly random element in G. Meaning our mapping maintains the uniformity of G, while still ensuring that all elements appear uniformly random. Therefore, definition 4 should hold.

Visualization

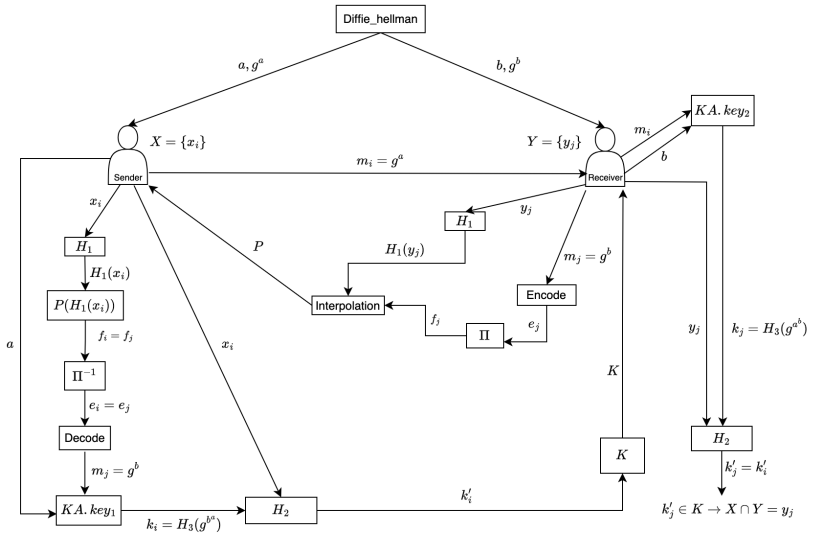


Figure 4.1: Illustration our PSI implementation flow. The receiver obtains confirmation that $y_j \in X \cap Y$ by learning that $x_i = y_j$ for some $x_i \in X$. The protocol ensures that no further information about the sender's input set X or the receiver's set Y is disclosed.

4.6 SUB-COMPONENTS

In this section, we will present a structured breakdown of the core sub-components and operations that make up our PSI implementation. We will talk about key security considerations that arise during implementation, as well as the workarounds and practical adaptations we employed to mitigate potential vulnerabilities or performance bottlenecks.

Generation of protocol parameters

We begin by selecting a random prime q using the `getPrime` function from the `sympy` library with parameter 2047, corresponding to a $\kappa \approx 128$ security level (https://www.keylength.com/en/compare/). This produces a random prime such that $2^{2046} \leq q < 2^{2047}$. Since we require a safe prime p in the range $2^{2047} \leq p < 2^{2048}$, we repeatedly generate new candidates for q until a pair (p, q) is found where $p = 2q + 1$.

A random field element $h \in \mathbb{Z}_p^*$ is sampled using the `secrets.randbelow` function from Python's `secrets` module. This value is then used to compute a generator g of the subgroup of order q, and we verify that it satisfies the required security conditions (e.g., $g \not\equiv 1 \mod p$, $g^2 \not\equiv 1 \mod p$, $g^q \equiv 1 \mod p$).

Finally, we generate the element $u \in \mathbb{Z}_p^* \setminus G$ using Python's `random.randint` function. The element u is chosen such that it lies outside the subgroup $G = \langle g \rangle$, i.e., $u^q \not\equiv 1 \mod p$, to ensure the correctness of the encoding scheme used later.

Finite Fields

Our implementation employs two different finite fields. The first is a prime field denoted $\mathbb{F}_p$, consisting of the elements

$$\{0, 1, \ldots, p-1\},$$

where p is a large safe prime. The second is a binary extension field denoted $\mathbb{F}_{2^{\lceil \log_2(p) \rceil}}$, which can be interpreted as bitstrings of length $\lceil \log_2(p) \rceil$, corresponding to integers in the range:

$$\{0, 1, \ldots, 2^{2048} - 1\}.$$

The motivation for using two distinct fields stems from a mismatch between the range of our permutation function and the arithmetic requirements of our polynomial interpolation. Specifically, the permutation function produces outputs that may exceed the safe prime p, whereas interpolation must be performed in a field to ensure the existence of modular inverses for all nonzero elements. The $\mathbb{F}_p$ field also needs the ability to make the key agreement messages pseudorandom (see definition 4: fig. [3.3]), which is not necessarily the case for $\mathbb{F}_{2^{\lceil \log_2(p) \rceil}}$. For this reason, we perform interpolation over the prime field $\mathbb{F}_p$, which guarantees that every nonzero element has a unique modular inverse,

As a result, we separate the responsibilities: the binary field is used for the permutation function's domain and range, while the prime field $\mathbb{F}_p$ is used for all arithmetic operations involving polynomial interpolation. This dual-field design ensures correctness throughout the protocol.

Diffie-hellman

We have implemented a basic Diffie–Hellman key pair generation. Given a generator $g \in \mathbb{Z}_p^*$ and a safe prime modulus $p = 2q + 1$, the function samples a private key uniformly at random from the interval $[1, q]$. It then computes the corresponding public key as $g^{\text{private\_key}} \bmod p$. The function returns the tuple $(\text{private\_key}, \text{public\_key})$, which can subsequently be used for secure key exchange in the Diffie–Hellman protocol.

Encoding

In our implementation of the maliciously secure PSI protocol, one crucial requirement is that elements transmitted as the second message ($m_i' = g^{b_i}$) in the Diffie-Hellman key exchange must be computationally indistinguishable from uniformly random bit strings of fixed length. This indistinguishability is essential for maintaining the security guarantees of the protocol, as defined in Definition 4 [3.3].

While the original paper uses elliptic curves and Elligator maps [Bernstein et al., 2013] to encode group elements as pseudorandom strings, we adopt a different strategy. Specifically, we replace elliptic curve operations with standard Diffie-Hellman over large safe primes, which simplifies implementation but introduces the challenge of encoding elements from a prime-order subgroup $G \subset \mathbb{Z}_p^*$ as pseudorandom bit strings.

The key issue is that the subgroup G has order q, whereas $\mathbb{Z}_p^*$ has order $2q$ (since $p = 2q+1$), meaning that a random element of G can be distinguished from a uniformly random string of length $\log(p)$ bits. To overcome this, we define a probabilistic encoding function `encode_group_element` that introduces blinding while preserving reversibility.

Encoding
Given:

- $x \in G \subset \mathbb{Z}_p^*$: a group element,
- p: a safe prime,
- $u \in \mathbb{Z}_p^* \setminus G$

the function proceeds as follows:

$$x \mapsto \begin{cases} x \bmod p & \text{with probability } \frac{1}{2}, \\ u \cdot x \bmod p & \text{with probability } \frac{1}{2}, \end{cases}$$

The resulting value is then encoded as a bit string of length 2048. This ensures that the output always is a random element in $\mathbb{F}_p$ with a fixed length.

Decoding
To recover the original group element, we define a decoding function that takes:

- `encoded`: the integer representation of the received bit string,
- p: the prime modulus,
- q: the order of the subgroup G,
- u: the non-generator element used in encoding,

The function first checks that the integer z corresponding to the bit string lies in the valid range $[1, p-1]$. Then, it checks whether $z^q \equiv 1 \mod p$:

- If so, then $z \in G$, and we return z as the original element.
- Otherwise, we compute and return $u^{-1} \cdot z \bmod p$, effectively reversing the random blinding.

This decoding process guarantees correctness regardless of the random choice made during encoding.

Permutation

Our permutation function implements a fixed-length permutation of bit strings by encrypting the input using the AES block cipher in CBC (Cipher Block Chaining) mode. The primary purpose of this function is to pseudorandomly permute a group element's bit representation, ensuring uniformity and unpredictability in the output while remaining invertible under known key and IV. The function takes the following arguments:

- `x_bitstring`: a binary string of fixed length (e.g., 2048 bits), representing the group element
- `key`: a secret AES key (256 bits)
- `iv`: a fixed or random initialization vector (IV) for CBC mode (128 bits)

The procedure is as follows:

1. The input bit string is converted into a byte string using a helper function `bitstring_to_bytes`.
2. The AES cipher is initialized with the given key and IV in CBC mode.

3. The input byte string is encrypted using AES, producing a ciphertext of the same length.
4. The ciphertext is interpreted as a large integer and re-encoded as a bit string of fixed length (e.g., 2048 bits) using Python's `format` function.

This process effectively defines a deterministic permutation on the space of all 2048-bit strings, parameterized by the key and IV. As AES is a pseudorandom permutation when the key is unknown, this transformation looks random but preserves invertibility for parties with access to the key and IV.

The function `inverse_permutation` implements the inverse of the permutation mapping. It reverses the pseudorandom permutation applied to a bit string by decrypting the AES-encrypted data using the same key and initialization vector (IV). The function takes the following arguments:

- `encrypted_bitstring`: a binary string of fixed length (e.g., 2048 bits), representing the output of the permutation mapping
- `key`: the same secret AES key used for encryption
- `iv`: the same initialization vector used in CBC mode during encryption

The procedure is as follows:

1. The input bit string is first converted back into a byte string using the helper function `bitstring_to_bytes`.
2. The AES cipher is initialized with the same key and IV in CBC mode.
3. The byte string is decrypted using AES, yielding the original plaintext byte string.
4. The plaintext is converted back into a binary string of fixed length (e.g., 2048 bits) using the `format` function.

Assuming the key and IV are identical to those used in the original permutation mapping, this function deterministically recovers the original input bit string. The correctness of this inverse relies on the fact that AES is a symmetric block cipher, and CBC mode with a fixed IV is reversible under decryption.

Permutation Workaround

To recover the permuted messages, the receiver performs polynomial interpolation over a set of points (x_i, y_i), where each y_i corresponds to a permuted message m'_i.

However, polynomial interpolation requires arithmetic in the field, $\mathbb{F}_p$. Since our messages are pseudorandom in the field $\mathbb{F}_p$.

Our permutation function produces outputs of 2048 bits. Consequently, it is possible that a permuted value f'_i may exceed the chosen prime p. Since interpolation requires all values to be within the field $\mathbb{F}_p$, we enforce the condition that all permuted values $m'_i < p$. To guarantee this, the receiver repeatedly permutes messages m'_i until the permuted f'_i satisfies $f'_i < p$, thereby ensuring the correctness of interpolation.

Hashing

Our implementation uses three different hash functions all modeled as random oracles:

hash function $\mathbf{H_1}$:

$$H_1 : \{0,1\}^* \to \mathbb{F}_p$$

This function maps arbitrary-length binary strings to elements of a finite field $\mathbb{F}_p$. It is defined as follows:

$$H_1(x) = \texttt{int.from\_bytes(SHA256}(x), \texttt{big-endian})$$

where SHA256 is from the hashlib library

Usage:

- On the *receiver* side, H_1 is used to derive the x-values for interpolation of the polynomial.
- On the *sender* side, H_1 is used to evaluate the polynomial at the hashed x-points.

hash function $\mathbf{H_2}$:

$$H_2 : \{0,1\}^* \times \mathbb{F} \to \{0,1\}^{2048}$$

This function maps a pair consisting of a string and a field element to a 2048-bit binary string. It is defined as:

$$H_2(x, k) = \texttt{SHA256}(x \parallel \text{" : "} \parallel k)$$

where $\parallel$ " : " $\parallel$ denotes string concatenation.

Usage:

- Used by the *sender* to construct the elements in the set K.
- Used by the *receiver* to verify membership of their own set elements in K.

hash function $\mathbf{H_3}$:

$$H_3 : \{0,1\}^* \to \mathbb{F}_p$$

This function is identical in structure to H_1, but returns the field element incremented by 1:

$$H_3(x) = \texttt{int.from\_bytes(SHA256}(x), \texttt{big-endian}) + 1$$

We use $+1$ to distinguish H_3 from H_1 **Usage:**

- Used to hash values such as g^{ab} and g^{ba} in key agreement.

Interpolation

For the interpolation, we have implemented a function `fast_modular_interpolation` that performs fast polynomial interpolation over the finite field $\mathbb{F}_p$, given a set of points (x_i, y_i). The algorithm returns the coefficients of a polynomial $f(x) \in \mathbb{F}_p$ such that $f(x_i) = y_i \mod p$ for all i, where p is our safe prime.

The implementation is based on a divide-and-conquer approach using Newton interpolation and we thereby achieve a complexity of $\mathcal{O}(n \log^2 n)$ field operations. It is particularly suited for large-scale interpolation over prime fields. The function takes in the following arguments:

- `x`: a list of distinct x-coordinates $x_0, \ldots, x_{n-1} \in \mathbb{F}_p$
- `y`: a list of corresponding y-coordinates $y_0, \ldots, y_{n-1} \in \mathbb{F}_p$

- `p`: a prime number specifying the modulus of the finite field $\mathbb{F}_p$

The interpolation procedure consists of the following steps:

1. **Divided Differences:** The function `fast_divided_differences` computes the coefficients of the interpolating polynomial in Newton form using modular arithmetic. These coefficients are derived via a recursive computation of divided differences:

$$f[x_i, \ldots, x_{i+j}] = \frac{f[x_{i+1}, \ldots, x_{i+j}] - f[x_i, \ldots, x_{i+j-1}]}{x_{i+j} - x_i} \mod p$$

   Modular division is performed via multiplicative inverses using Fermat's Little Theorem.

2. **Polynomial Construction:** The Newton-form coefficients are converted to the standard basis (monomial basis) by incrementally constructing the product terms $(x - x_0)(x - x_1) \ldots (x - x_{i-1})$ and combining them with the Newton coefficients. The conversion leverages modular polynomial multiplication, implemented in the helper function `poly_mult`.

3. **Modular Arithmetic:** All arithmetic operations (addition, multiplication, and inversion) are performed modulo p, ensuring correctness over $\mathbb{F}_p$.

The algorithm outputs the coefficient list $[a_0, a_1, \ldots, a_{n-1}]$ representing the interpolated polynomial

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \mod p$$

such that $f(x_i) = y_i$ for all $i \in \{0, \ldots, n-1\}$. To preserve the privacy of the input sets held by the sender and the receiver, it is essential that neither party learns anything about elements that are not part of the set intersection.

The receiver has interpolated a polynomial $P(x)$ over a finite field $\mathbb{F}_p$ such that

$$P(H_1(y_i)) < p \quad \text{for all } y_i \in Y$$

The use of a prime field $\mathbb{F}_p$ ensures that all polynomial evaluations and arithmetic operations take place within a finite field $\mathbb{F}_p$. By keeping all evaluations strictly within the field $\mathbb{F}_p$, we ensure that the range of possible outputs is uniformly distributed and bounded. This prevents the sender from distinguishing between elements inside and outside the receiver's set based on the structure or magnitude of $f(x)$ modulo p. Thus, the polynomial representation and evaluation over $\mathbb{F}_p$ serves as a cryptographic tool to guarantee that no information about $Y \setminus X$ is leaked during the protocol.

Multi-point evaluation

For the evaluation, we've implemented an algorithmic multi-point evaluation that should maintain cryptographic security. Traditional methods would evaluate a polynomial like $P(x) = a_0 + a_1 x + \ldots + a_n x^n$ at each point r_i independently, requiring $O(n^2)$ operations, which can be expensive for large-scale cryptographic protocols. Instead, we adopt a hierarchical approach inspired by polynomial modular arithmetic, reducing the complexity to $O(m + n \log^2 n)$ while ensuring all computations remain securely within the finite field $\mathbb{F}_p$.

The algorithm begins by constructing a *subproduct tree*, a binary tree structure where each leaf represents a linear polynomial $(x - r_i)$ corresponding to an evaluation point. These leaves are recursively multiplied in pairs up the tree, culminating in a root polynomial that is the product of all $(x - r_i)$ terms. This tree serves as a computational blueprint: by taking the original polynomial $P(x)$ and successively computing its remainder modulo each polynomial in the tree

(starting from the root), we decompose the problem into smaller subproblems. The Polynomial Remainder Theorem guarantees that

$$P(x) \bmod (x - r_i) = P(r_i),$$

meaning the final remainders at the tree's leaves are precisely the evaluations we seek. This divide-and-conquer strategy reduces redundant calculations and intermediate results are reused rather than recomputed, mirroring the efficiency gains of dynamic programming.

While our current implementation prioritizes clarity using fast multi-point evaluation ($O(m + n\log^2(n))$). The algorithms efficiency makes it useful for private set intersection protocols, where polynomial evaluations can dominate the computational cost. There are faster algorithms like Fast modular transform and Karatsuba which can run in $O(n \log n)$. But these potential improvements, however, would be larger and more difficult to implement with our large integers.

Working with the set K

The sender's construction of K
After receiving the polynomial, the sender constructs the set $K = \{K_1, \ldots, K_n\}$ our implementation works as follows:

1. Each element $x_i \in X$ is first hashed using H_1. Let $h_i = H_1(x_i) \bmod p$.
2. The sender evaluates a secret polynomial P at each h_i, computing $P(h_i) \in \mathbb{F}_p$.
3. Each evaluation result $P(h_i)$ is then encoded as a bitstring of fixed length and passed through our inverse permutation function. The permuted bitstring is interpreted as an integer modulo p, then decoded into a group element representation.
4. The decoded value is exponentiated with sender's private exponent $a \in \mathbb{F}_p$ to compute g^{b^a}, where g^b is implicitly embedded in the decoding.
5. The result is converted to a bitstring and hashed using a second random oracle $H_3 : \{0,1\}^* \to \mathbb{F}_p$ to derive a field element k_i.
6. Finally, the sender computes $K_i = H_2(x_i, k_i)$ using a third hash function H_2, which maps the pair (x_i, k_i) to a fixed-length output.
7. After all elements have been processed, the list $K = [K_1, \ldots, K_n]$ is randomly shuffled.

The receivers checking whether his elements are in K
Let $Y = \{y_1, y_2, \ldots, y_n\}$ denote the receiver's private input set, and let K be the (shuffled) list of ciphertexts received from the sender. The receiver holds an exponent b_i, and a common base group element g^a encoded as a bitstring $m \in \{0,1\}^{2048}$. The receiver tests each element $y_i \in Y$ to see if it lies in K as follows:

1. For each $i \in \{1, \ldots, n\}$, compute

$$g^{ab_i} = m^{b_i} \mod p,$$

   where the bitstring m is interpreted as an integer and p is the prime modulus of the field.

2. Represent g^{ab_i} as a 2048-bit string:

$$\text{bitstring}_i = \text{format}(g^{ab_i}, 02048\text{b}).$$

3. Hash this bitstring using the hash function $H_3 : \{0,1\}^* \to \mathbb{F}$ to obtain the shared key:

$$k_i = H_3(\text{bitstring}_i).$$

4. Use the hash function $H_2 : \{0,1\}^* \times \mathbb{F} \to \{0,1\}^{256}$ to compute:

$$h_i = H_2(y_i, k_i).$$

5. If $h_i \in \mathcal{K}$, then y_i is deemed to be a shared element, and is added to the receiver's output set:

$$\text{Output} := \text{Output} \cup \{y_i\}.$$

The output set thus contains exactly those elements of Y which were also encrypted into K by the sender, preserving privacy of both sets due to the use of random exponents and cryptographic hash functions.

5 RESULTS AND FINDINGS

5.1 IMPLEMENTATION CODE

To compare our modified version of the PSI protocol with the malicious protocol proposed in [Rosulek and Trieu, 2021], we have implemented our own version, which is available on GitHub: Here

5.2 RESULTS

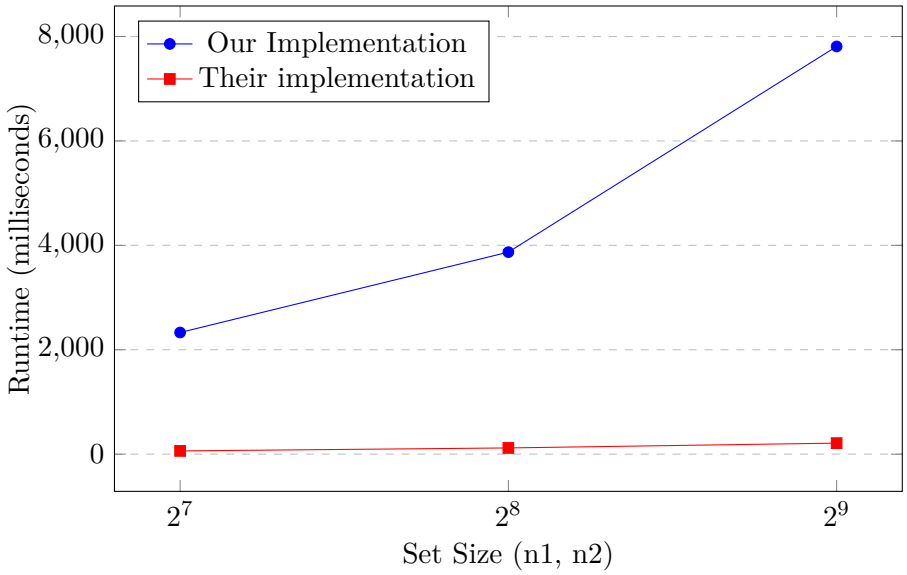


Figure 5.1: Comparison of our vs their protocol runtime. The x-axis represents set size, and the y-axis shows total runtime in milliseconds, n1 and n2 being the sender and receiver respectively. This figure works to show the difference in runtime for smaller set sizes in greater detail.

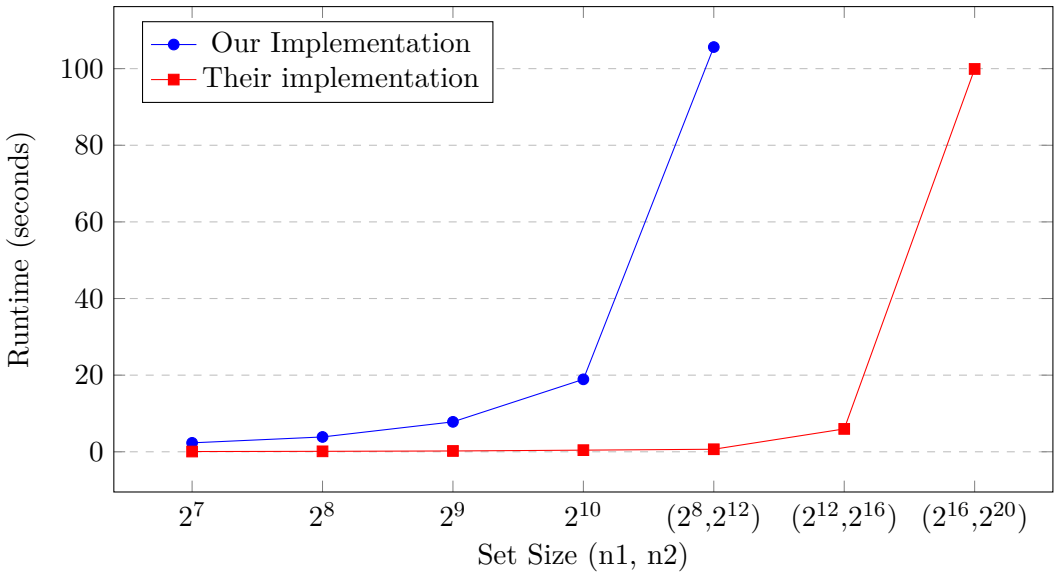


Figure 5.2: Comparison of our vs their protocol runtime, n1 and n2 being the sender and receiver respectively. The x-axis represents set size, and the y-axis shows total runtime in seconds.

Table 5.1: Runtime Comparison (Ours vs Theirs)

n_1, n_2	Runtime (s)		Runtime (ms)	
	Ours	Theirs	Ours	Theirs
2^7	2.33	0.0621	2330	62.1
2^8	3.87	0.1195	3870	119.5
2^9	7.81	0.2112	7810	211.2
2^{10}	18.91	0.4245	18910	424.5
$(2^8, 2^{12})$	105.63	0.66	105630	660
$(2^{12}, 2^{16})$	—	5.97	—	5970
$(2^{16}, 2^{20})$	—	99.91	—	99910

5.3 EXPERIMENTATION SETUP

We have implemented our code in Python, and ran our protocol on a single AMD Ryzen 7 5800X with 3801 MHz and 32 GB RAM. Considering that the runtime of our implementation is much slower, we have decided not to introduce artificial network delays, as network speed in this case will only make a marginal difference in the runtime. Taking this into consideration, to keep this as fair as possible, we are comparing our results to the runtime of the article's test when run on a 10 Gbps network.

Implementation evaluation

When examining the results, it becomes clear that our implementation does not match the efficiency of the protocol presented in [Rosulek and Trieu, 2021]]. As shown in Figures 5.1 [5.2] and 5.2 [5.2], while both implementations perform well on smaller sets, the runtime of our implementation increases significantly faster as the set size grows. The protocol proposed in the article maintains low runtimes for small sets and scales more gracefully overall.

A key point of divergence occurs around the set size $(2^8, 2^{12})$, or 256 and 4096 elements for the sender and receiver, respectively. At this point, our implementation begins to experience a steep increase in computation time, resulting in poor performance. In contrast, the article's protocol handles much larger inputs, up to approximately $(2^{16}, 2^{20})$, or 65,536 and 1,048,576 elements, before encountering similar performance degradation. This suggests their implementation is significantly more scalable and better optimized.

Looking at table 5.1 [5.2], we can also see that we have not included data for our implementation beyond $(2^8, 2^{12})$. This is due to scalability limitations, possibly stemming from higher algorithmic complexity, inefficient memory usage, or the absence of optimizations such as parallelism. A memory bottleneck may also be contributing to the prolonged runtimes. These results indicate the presence of a "performance cliff" in our implementation, where runtime increases dramatically past a certain threshold, making it infeasible to obtain results within a reasonable timeframe for larger input sizes.

6 DISCUSSION

Hardware

The article states that their experiments were conducted using a single Intel Xeon processor running at 2.30 GHz, alongside 256 GB of RAM. Given the publication year (2021), a likely candidate for the CPU model is the Xeon E5-2697 v4, which has been commonly used in high-memory server environments. In contrast, our experiments were conducted using an AMD Ryzen 7 5800X, which generally offers superior per-core performance and a more modern architecture. One key hardware-related factor that could influence performance, in regards to CPU, is memory bandwidth, particularly the difference in memory channel configurations. The Intel Xeon series, likely including the model used in the article, typically supports quad-channel memory, enabling higher data transfer rates between RAM and CPU through parallel channels. In contrast, our AMD Ryzen 7 5800X only supports dual-channel memory, which limits memory throughput. While this difference has little impact on small inputs that fit into CPU cache, it becomes increasingly important for larger set sizes, where data must frequently be fetched from RAM. In such cases, higher memory bandwidth helps avoid CPU stalls due to memory access delays, potentially explaining why our implementation degrades in performance more quickly as input size increases.

Another significant hardware-related difference lies in the available memory. They used 256 GB, while we only had access to 32 GB. This eightfold increase in RAM on their side could have a large effect on computation times, particularly for operations involving large input sets or memory-intensive tasks. However, for smaller input sets, memory usage is typically limited, so this difference would primarily impact performance at larger set sizes. Therefore, while hardware differences may contribute to some performance variation, they are unlikely to fully explain the results observed in the small to mid-size range.

Large numbers / using elliptic curve

When discussing the switch from elliptic curve cryptography (ECC) to large integers for the PSI protocol, there are several performance challenges being introduced. ECC allows for way smaller key sizes (256-bit) while offering the same security as much larger keys in traditional methods. In our case, we have used classic DHKA, which offers the same amount of security but with a much larger key of size 2048 bits. This results in theirs having faster computations and lower memory and bandwidth usage. In contrast, large integer operations, such as modular exponentiation or multiplication, become significantly slower as the key size increases. These operations are more computationally heavy and demand more memory and bandwidth, particularly when dealing with large datasets in the PSI protocol.

Additionally, the smaller key sizes in ECC directly contribute to better scalability, enabling the protocol to handle larger sets more efficiently. As integer-based methods require more complex operations and larger keys (e.g., 2048 or 4096 bits), slows down the protocol, increasing both runtime and communication cost. As [Rosulek and Trieu, 2021] predicts for an RSA protocol, the running time should be at least 20 times slower and the communication costs should be 8

times as expensive. Overall, while large integers offer flexibility, elliptic curves provide a more efficient and scalable solution for secure protocols like PSI, balancing security with computational efficiency.

Python vs C++

In our evaluation, the PSI (Private Set Intersection) protocol was implemented in Python, while the protocol from the article Rosulek and Trieu, 2021 was developed in C++. This section outlines the effects of this difference in implementation language.

Feature	C++	Python
Execution Speed	Native machine code (very fast)	Interpreted (slow)
Memory Efficiency	Manual, optimized	Automatic, less efficient
Library Support	Full access to fast crypto libs	Limited by wrappers
Parallelism	Multi-threaded, SIMD	Limited (GIL-bound)
Real-Time Predictability	High	Low

Figure 6.1: Benefits of using C++ instead of Python

1. Compiled Execution
C++ is compiled ahead of time into machine code by tools like GCC or Clang. This results in significantly faster execution compared to Python, which is interpreted line-by-line at runtime. For compute-heavy workloads like hashing, modular arithmetic, or encryption used in PSI, this difference can heavily impact performance.

2. Low-Level Memory Control
C++ gives developers fine-grained control over memory allocation, pointer arithmetic, and data layout. This allows for optimizations that reduce memory usage and improve cache locality, which is important when working with large sets that may not fit entirely in fast-access CPU cache.

3. Optimized Cryptographic Libraries
High-performance cryptographic toolkits (like OpenSSL, RELIC, libsodium, EMP-toolkit) are primarily written in C or C++. C++ implementations can call these libraries directly, without the performance penalty of crossing into Python via wrappers or bindings.

4. Parallelism and SIMD
C++ supports parallel computation through native threads, OpenMP, or libraries like Intel TBB. It also allows for explicit use of SIMD instructions (AVX, SSE, AVX-512), which can process multiple data elements in parallel. In contrast, Python's Global Interpreter Lock (GIL) limits its ability to fully utilize multi-core CPUs for CPU-bound tasks.

5. Lower Runtime Overhead
Python is easier to use but introduces significant runtime overhead from features like dynamic typing, interpreted execution, and object abstraction. This makes it unsuitable for high-performance tasks where even milliseconds matter.

So why did we write in python instead of C++?
The main reason we chose to write our implementation in Python rather than C++, as done in the original protocol, is that we had no real prior experience with C++. We felt more comfortable

using a language we were familiar with, which allowed us to focus on the logic and design of the protocol without being slowed down by language-specific hurdles.

In hindsight, we now recognize that using C++ could have provided significant performance benefits due to its lower-level control and efficiency. Interestingly, one of the biggest performance improvements we achieved was by switching to the gmpy2 package (a C-coded Python extension) for modular exponentiation, instead of using Python's built-in pow() function. This change alone made our implementation approximately 5–6× faster.

This also highlights that there are likely other areas in our code where similar optimizations could be made, and that the choice of language and libraries can have a major impact on runtime efficiency.

6.1 FUTURE WORK IDEAS OR OPTIMIZATION

Implementing using elleptic curves

As discussed earlier, implementing the PSI protocol using ECDH could be a promising avenue for optimization, especially since our approach using classic DHKA seems to be causing a bottleneck for our protocol. Elliptic curves offer significantly smaller key sizes while providing the same level of security as larger integer-based systems, such as classic DHKA, which leads to noticeable performance improvements. This results in faster computations, reduced memory usage, and lower bandwidth requirements, all which are critical factors when handling large-scale datasets efficiently. By switching to elliptic curves, we could not only improve the performance of the protocol but also make it more scalable for real-time applications.

Moreover, adopting elliptic curves would allow us to test and observe the changes in performance and behavior that different languages and libraries bring. By utilizing the same algorithms as those in [Rosulek and Trieu, 2021], we could make a direct comparison and gain insight into how much of a difference the implementation choices make. This would also provide valuable perspective on the ease of implementing the protocol directly using elliptic curve cryptography.

Implementing our protocol in C++

Choosing a language like C++ to implement a PSI protocol is obviously in hindsight a smarter choice due to its combination of performance and control over system resources. C++ has the ability to handle complex algorithms efficiently, especially when dealing with large integers and cryptographic operations. This would help in the instances of memory management and parallelization, which would be crucial for speeding up the computation-heavy steps of our PSI. Moreover, C++ has optimized cryptographic libraries like OpenSSL or GMP to ensure the performance bottlenecks are minimized. Already from a top-down perspective, this should lead to a lot of enhancements to our protocol, which would be very interesting to delve into.

Additionally, implementing the protocol in C++ would allow us to more precisely compare the performance and behavior of large integer-based cryptography against elliptic curve cryptography, offering a clearer picture of the trade-offs between these two approaches in terms of speed, efficiency, and scalability.

7 CONCLUSION

In this project, we implemented and evaluated a Private Set Intersection (PSI). Our goal was to gain practical insight into the cryptographic principles behind PSI protocols and use this knowledge to make our own implementation inspired by the work of [Rosulek and Trieu, 2021]. Unlike their protocol, which relies on elliptic curve cryptography (ECC) and Elligator encodings, our implementation uses classic Diffie-Hellman key agreement (DHKA) and custom encodings.

Although our protocol deviates from the one presented in [Rosulek and Trieu, 2021], by omitting ECC and Elligator, it still satisfies the security definitions required, making our protocol maliciously secure.

Our protocol performed correctly, but showed clear performance limitations when applied to large datasets. This was largely due to the computational overhead of large-integer arithmetic and the interpreted nature of Python. While we did not implement the protocol in a low-level language like C++, our results suggest that the poor performance we observed is unlikely to be fully mitigated by a change in language alone. The overhead from using large integer arithmetic in our finite field implementation proved to be a significant bottleneck, particularly with larger datasets. As such, our implementation is only useful when handling small set sizes, and would benefit from more optimizations.

BIBLIOGRAPHY

Bernstein, Daniel J, Mike Hamburg, Anna Krasnova, and Tanja Lange (2013). "Elligator: Elliptic-curve points indistinguishable from uniform random strings." In: *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security*. ACM, pp. 967–980.

Mezzour, Ghita, Adrian Perrig, Virgil Gligor, and Panos Papadimitratos (2009). "Privacy-Preserving Relationship Path Discovery in Social Networks." In: *Cryptology and Network Security*. Ed. by Juan A. Garay, Atsuko Miyaji, and Akira Otsuka. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 189–208. ISBN: 978-3-642-10433-6.

Pinkas, Benny, Thomas Schneider, and Michael Zohner (2014). *Faster Private Set Intersection based on OT Extension*. Cryptology ePrint Archive, Paper 2014/447. URL: https://eprint.iacr.org/2014/447.

Raghuraman, Srinivasan and Peter Rindal (2022). *Blazing Fast PSI from Improved OKVS and Subfield VOLE*. Cryptology ePrint Archive, Paper 2022/320. URL: https://eprint.iacr.org/2022/320.

Rosulek, Mike and Ni Trieu (2021). *Compact and Malicious Private Set Intersection for Small Sets*. Cryptology ePrint Archive, Paper 2021/1159. DOI: 10.1145/3460120.3484778. URL: https://eprint.iacr.org/2021/1159.

Declaration of use of GAI tools

☐ **I/we used generative artificial intelligence (GAI) to complete this project** *(tick the box).* List the GAI tool(s) you used (remember to specify version):

- *ChatGPT(GPT-4, GPT-3.5)*
- *Claude sonnet*
- *Co-pilot version 1.330.0*

I/we used GAI tools in the following way *(See accompanying list of possible uses for inspiration)*

EXAMPLES OF LEGITIMATE USE CASES FOR GAI
For alternative ways of formulating text
To understand a topic better
For programming tasks

For each relevant use case, explain how you used GAI (see an explanation of the various possibilities on Studypedia). For example, if you used GAI to generate information, explain how you did this and how you used the output in your paper.

For alternative ways of formulating text:
We have used chatGPT to come up with better ways to formulate text. It could be the case that we have written a block of text where it might be hard to see what we are trying to say, then we would give the text as a promt and ask for a modified version with more clarity. We would then review the response and use it/parts of if we feel it improves the quality.

To understand a topic better:
When encountering topics outside our scope we used the chatGPT to expand our knowlegde upon that topic.

For programming tasks:
When implementing the code for our protocol we used Claude, chatGPT and co-pilot. Examples of use could be when encountering errors. Co-pilot was used for autospelling.