AN ABSTRACT OF THE THESIS OF

Ni Trieu for the degree of Doctor of Philosophy in Computer Science presented on March 10, 2020.

Title: Efficient Private Matching for Private Databases

Abstract approved: _____

Mike Rosulek

Private matching (PM) is a key cryptographic primitive in secure computation that allows several parties to jointly compute some functions depending on their private inputs. Indeed, this primitive has many practical applications. For instance, in online advertising, two companies may wish to find their common customers for a joint marketing campaign. In this scenario, privacy is of utmost importance and it is imperative to ensure that neither company can learn more than their own data and the results of the match.

In recent years, secure computation in general and PM in particular has attracted considerable attention from the research community, partly due to the rise of Big Data along with the ever-increasing privacy concerns. This thesis describes three secure private set intersection (PSI) protocols, one private set union (PSU) construction, and one pattern matching scheme. PM can be considered as a main building block in all these protocols.

To securely compute the intersection of two sets of size 2^{20} our proposed protocols require only 3 seconds which is $4\times$ faster than the previous best protocol. In the multi-party setting, we provide the first implementation that takes only 72 seconds to compute PSI for 5 parties with data-sets of 2^{20} items each. For private set union (PSU), our protocol improves prior work by a factor up to $7,600\times$ for large instances. In addition, our wild-card pattern matching (WPM) protocol shows over two orders of magnitude faster than the state-of-the-art scheme.

Efficient Private Matching for Private Databases

by

Ni Trieu

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented March 10, 2020
Commencement June 2020

Doctor of Philosophy thesis of Ni Trieu presented on March 10, 2020.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Enginering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Ni Trieu, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation to my PhD advisor, Professor Mike Rosulek, for his superb guidance, enthusiastic encouragement, and unconditional support during the past five years, and for bringing me to the field of Cryptography. I consider myself fortunate enough to have studied under his supervision. He is my primary resource for getting answers to my crypto questions, even sometimes "*stupid*" and naive, for instance, questions relating to OT when I was a first-year PhD student. Mike has been always very supportive, provided insightful discussions about the research, and has given me the freedom to work on many exciting projects without any pressure.

I am also grateful to my awesome mentors, Vladimir Kolesnikov, Payman Mohassel, Mariana Raykova, for giving me the opportunities to enjoy wonderful summer internships at Bell Labs, Visa Research, and Google. I would like to thank them for their brilliant supervision and mentoring, and also for providing excellent environments for doing world-class research.

Additionally, I want to give special thanks to all my coauthors, Daniel Demmler, Vladimir Kolesnikov, Ranjit Kumaresan, Naor Matania, Benny Pinkas, Peter Rindal, Avishay Yanai, and Xiao Wang. This dissertation would not have been possible without their expertise and contributions.

Furthermore, I wish to take this opportunity to acknowledge and extend my thanks to my committee members, Glencora Borradaile, Rakesh Bobba, Vladimir

Kolesnikov, and Oksana Ostroverkhova, for their thoughtful suggestions and constructive feedback on my research topics. Their input has been extremely helpful to improve this dissertation.

Finally, I especially thank my family, friends for their endless support and encouragement. Without them, this journey would have been much more difficult and a lot less enjoyable.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF TABLES (Continued)

LIST OF TABLES (Continued)

LIST OF ALGORITHMS

Algorithm | Page

Chapter 1: Introduction

With the rise of big data and machine learning, computing on joint databases begins to play a central role in both the scientific and business environments. Along with these developments, the privacy and security concerns have become pervasive. For example, how can we enable two untrusted parties to jointly evaluate a function over their private inputs without revealing their data? How can we ensure that computations are not compromised, passwords are not broken, and your encrypted data is not eavesdropped?

My research focuses on solving the above problems by designing cryptographic protocols and systems that can be evaluated in terms of security assumptions, computational running time, and required number of bits of communication. The main objectives of my research are to minimize the necessary assumptions and to devise more efficient solutions.

To this end, I have proposed several protocols for private matching and private set intersection/union, which allow many parties, each holding a set of items, to learn their common/different items without revealing anything else about their sets. Indeed, these problems are special cases of secure computation, which allows many parties jointly compute a function on their private inputs in a way that no party learns anything other than the output itself. This thesis summarizes my research so far, which has made significant contributions to the field of secure

computation in general and private matching in particular.

1.1 Problem Statement

Private Matching (PM) is a 2-party protocol in which a receiver who has an input string x_0 interacts with a sender holding an input string x_1. The result is that the receiver learns an output of bit/function depending on whether x_0 and x_1 are matched according to a defined measurement and nothing else, whereas the sender learns nothing. The defined measurement can be hamming distance, edit distance, Euclidean distance. Instead of the single input string, each party can have a set of items. In addition, when we define that x_0 and x_1 are matched if they are strictly equal, PM can be extended to a private set operation where parties want to learn only the intersection (so-called PSI) or union (so-called PSU) of two sets. Moreover, revealing the intersection/union sets to the party might be violation of the privacy objective in some cases and it would be more appropriate to only output a function on the set of common/different items. For example, parties would like to compute the cardinality of the intersection, or some aggregation (e.g., sum, average) of items in the intersection without revealing any additional information, including the intersections set.

Secure multi-party computation enables distrustful users to jointly evaluate any function on their private inputs without requiring a trusted third party and without revealing anything except the result itself. However, generic cryptographic and secure computation tools may not be suitable and efficient to tackle some

variants of PM since every problem may have distinct features and come with specific requirements. Thus, it is highly desirable to seek new theoretical and technical refinements to improve their performance, as well as propose concrete optimizations tailored for the real-world and important applications of PM.

1.2 Applications

To motivate my research, I first present some relevant and promising application scenarios of PM and explain why a privacy-preserving solution is desirable in these settings. Indeed, PM can be applied in various contexts, such as: (1) *Private Database Queries* where a client wants to retrieve some records based on her/his keyword search (e.g., contact discovery for mobile services in which the client learns the intersection of its own contact list and a server's user database); (2) *Voter Registration* where two states would like to identify people registered to vote in both states; and (3) *Threat Log Comparison* which attempts to identify common attacks across several networks. These applications exemplify the wide variety of research interests and practical areas where private matching can be a cryptographic primitive tool for secure computation.

The most common solution to the above applications is to utilize a trusted third-party (organization) to whom users could send their data. The trusted third-party would then compute the desired functions (e.g., matching) and send the final results to the users. For instance, consider an example related to DNA analysis, where a researcher holds a specific cancer marker sequence with some errors and

wants to discover the frequency and positions of the gene occurrences in a patient genomic database possessed by a hospital. Since genomic data is highly sensitive in nature, the hospital is required keep the database private, while the researcher needs to protect the specific genome sequence that he is working on. To tackle this problem, both the researcher and the hospital send their data to a trusted cloud service provider (e.g., AWS) for further computation, and obtain the output from the service. As a result, the service provider has access to a large amount of highly sensitive information that is supposed to be kept strictly private.

1.3 Contributions

This dissertation makes significant improvements to the state-of-the-art PM literature. In particular, Chapter 3 and 4 present three of my fastest PSI protocols [KKRT16, PRTY19, KMP+17]. In Chapter 5, I describe my PSU scheme [KRTW19] which shows $7,600\times$ faster than the state-of-the-art work. The richer problem of PM, which is secure pattern matching [KRT18], is shown in the last chapter.

I have also summarized my recent PSI works [PRTY20, RT20, DRRT18] in this section. However, due to the space limitation, this dissertation will not discuss them in details.

1.3.1 Two-Party PM/PSI.

One of my first works [KKRT16] significantly improves the state-of-the-art PSI protocol of [PSSZ15] by replacing one of its key components with a new cryptographic primitive "batched, related-key Oblivious Pseudorandom Function" (BaRK-OPRF). Indeed, many recent works have employed our new primitive as a building block for their applications. We implemented our BaRK-OPRF-based PSI protocol and found ours to be $2.3-3.6\times$ faster than the protocol in [PSSZ15]. In particular, our protocol requires only 3.8 seconds to securely compute the intersection of 2^{20}-size sets. Moreover, our protocol is only $4.3\times$ slower than a plaintext intersection with no security.

Later, I introduced a new PSI technique [PRTY19, RT20] that has the lowest communication complexity to date. The technical core of the approach [PRTY19] is a new cryptographic tool (called sparse OT extension). Conceptually, it can be thought of as a communication-efficient OPRF evaluation. Compared to my first work [KKRT16], this new scheme requires $1.8-2.1\times$ less communication.

Very recently, I proposed a new oblivious data structure as a key-value store which called a probe-and-XOR of strings (PaXoS), and use it as the building block for our efficient PSI protocol [PRTY20] against malicious adversary who o may arbitrarily deviate from the protocol. Our new PSI protocol achieves the first linear-communication protocol in malicious setting.

1.3.2 Multi-party PM/PSI.

We extended the traditional two-party protocol setting to a multi-party scenario that enables more than two parties to compute the private intersection/matching of their datasets. Our protocols [KMP+17] avoid computationally expensive public-key operations and are secure in the presence of any subset of semi-honest participants who assumed to follow the protocol, but attempts to obtain extra information from the execution transcript.

Noticeably, to the best of our knowledge, ours is the first protocol that has implementation (the code is publicly available), and we also provide experimental results. For 5 parties with data-sets of 2^{20} items each, our protocol requires only 72 seconds. Our novel underlying technique is oblivious evaluation of a programmable pseudorandom function (OPPRF), which we believe is of independent interest and may find applications in other applications as well.

1.3.3 PM/PSI Real-world Applications.

We consider the "*Private Contact Discovery*" problem. In this scenario, when a new user signs up for a social networking service, she is required to identify which of her existing contacts also use the service. Most of the current services (e.g. WhatsApp) ask the new user to send her/his contacts to their servers. Consequently, the service providers hold significant information about the user such as the user's name, phone numbers, emails, and friends that have not registered for the service, and possibly much more. Recently, a German court has ruled that a

user is not allowed to upload her contact list to the WhatsApp servers without written permission of all of her contacts.

To address the privacy concerns mentioned above, we present a novel system [DRRT18] in which the user learns only the common list of its own contacts and the server's user database of the service providers, while the server can learn only the (approximate) size of the user's list. We combine several existing cryptographic tools and propose concrete optimizations tailored for this application. Our contact discovery scheme between a client with 1024 contacts and a server with 67 million user entries requires only 1.36 sec and costs only 4.28 MiB of communication, which is truly practical via mobile networks.

1.3.4 PM/PSI Advanced Variant Functionality.

We present richer functionalities to capture more accurately the requirements of real-world applications. Suppose a server holds a long text string and a client holds a short pattern string. Secure pattern matching allows the client to learn the locations in the long text where the pattern appears, while leaking nothing else to either party besides the length of their inputs. In [KRT18], we investigate the secure wildcard pattern matching problem, where the client's pattern can include wildcard characters that can match any character in the database. Indeed, DNA analysis can be a promising application of our proposed wildcard pattern matching scheme.

Additionally, we proposed a new cryptographic primitive [KRTW19] (called

reverse private membership test) and considered its application for computing private set union (i.e., compute the union of parties' private sets without revealing anything else to any other party). Private set union has several real-world applications, especially in network security, such as cyber risk assessment and management via joint IP blacklists and joint vulnerability data.

Chapter 2: Background Theory

2.1 Notation

Throughout this thesis we use the following notation. We denote vectors in bold $\boldsymbol{a}$, and matrices in capitals A. For the vector, we let $\boldsymbol{a}_{[i,j]}$ denote the sub-vector of $\boldsymbol{a}$ from i-th bit to j-th bit, and a_i denote the i-th bit of vector $\boldsymbol{a}$. Given vectors $\boldsymbol{a} = a_1 \| \cdots \| a_n$ and $\boldsymbol{b} = b_1 \| \cdots \| b_n$, we define $\oplus$ and $\cdot$ operations as follows. We use the notation $\boldsymbol{a} \oplus \boldsymbol{b}$ to denote the vector $(a_1 \oplus b_1) \| \cdots \| (a_n \oplus b_n)$. Similarly, the notation $\boldsymbol{a} \cdot \boldsymbol{b}$ denotes the vector $(a_1 \cdot b_1) \| \cdots \| (a_n \cdot b_n)$. Let $c \in \{0, 1\}$, then $c \cdot \boldsymbol{a}$ denotes the vector $(c \cdot a_1) \| \cdots \| (c \cdot a_n)$. For a matrix A, we let $\mathbf{a}_i$ denote the i-th row of A, $\mathbf{a}^j$ denote the j-th column of A; A_i^j denote the entry of A at the i-th row and the j-th column.

The computational and statistical security parameters are denoted by κ, λ, respectively. $[m]$ to denote a set $\{1, \ldots, m\}$.

Consider an alphabet Σ. Wildcard is denoted by $\star$. We define a **pattern matching relation** $\preceq$ via the following rules: (1) $a \preceq a$ for $a \in \Sigma$; (2) $\star \preceq a$ for $a \in \Sigma$. We extend the notation to vectors as $\boldsymbol{x} \preceq \boldsymbol{y} \Leftrightarrow (\forall i) x_i \preceq y_i$. If $\boldsymbol{p} \preceq \boldsymbol{x}$ we say that $\boldsymbol{x}$ **matches the pattern** $\boldsymbol{p}$**.**

We write $d_H(x,)$ to denote the hamming weight of a binary string x. Our computational security parameter is κ and statistical security parameter is σ.

2.2 Secure Computation

The security of a secure multi-party protocol is formally defined by comparing the distribution of the outputs of all parties in the execution of the protocol π to an *ideal* model where a trusted third party is given the inputs from the parties, computes f and returns the outputs. The idea is that if it is possible to simulate the view of the adversary in the real execution of the protocol, given only its view in the ideal model (when it only sees its input and output), then the adversary cannot do in the real execution anything that is impossible in the ideal model, and hence the protocol is said to be secure.

In the real-world execution, the parties often execute the protocol in the presence of an adversary $\mathcal{A}$ who corrupts a subset of the parties. In the ideal execution, the parties interact with a trusted party that evaluates the function f in the presence of a simulator Sim that corrupts the same subset of parties. There are two classical security models.

- Colluding model: This is modeled by considering a single monolithic adversary that captures the possibility of collusion between the dishonest parties. The protocol is secure if the joint distribution of those views can be simulated.

- Non-colluding model: This is modeled by considering independent adversaries, each captures the view of each independent dishonest party. The protocol is secure if the individual distribution of each view can be simulated. In this work, we study a model that we assume to know at least two

parties that do not collude.

There are also two adversarial models. In the semi-honest (or honest-but-curious) model, the adversary is assumed to follow the protocol, but attempts to obtain extra information from the execution transcript. In the malicious model, the adversary may follow any arbitrary strategy.

The following definitions present a simplified form of the non-collusion/collusion semi-honest security [Ode09, KMR11] in multi-party setting.

Real-world execution. The real-world execution of protocol Π takes place between parties $(P_1, \ldots, P_n)$ and adversaries $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$, where $m < n$. Let $H \in [n]$ denote the honest parties, $I \in [n]$ denote the set of corrupted and non-colluding parties and $C \in [n]$ denote the set of corrupted and colluding parties.

At the beginning of the execution, each party $P_{i\in[n]}$ receives its input x_i and an auxiliary input z_i while each adversary $\mathcal{A}_{i\in[m-1]}$ receives an index $i \in I$ that indicates the party it corrupts, while adversary $\mathcal{A}_m$ receives C indicating the set of parties it corrupts.

For all $i \in H$, let OUT_i denote the output of P_i and for $i \in I \cup C$, let OUT'_i denote the view of party P_i during the execution of Π. The i^{th} partial output of a real-world execution of Π between parties $(P_1, \ldots, P_n)$ in the presence of adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_m)$ is defined as

$$\mathsf{REAL}^{i}_{\Pi,\mathcal{A},I,C,z_i}(k, x_i) \stackrel{\mathsf{def}}{=} \{\mathsf{OUT}_j \mid j \in H\} \cup \mathsf{OUT}'_i$$

Ideal-world execution. In the ideal-world execution, all the parties interact

with a trusted party that evaluates a function f. As in the real-world execution, the ideal execution begins with $P_{i\in[n]}$ receives its input x_i and an auxiliary input z_i. Since we consider a semi-honest setting, each party $P_{i\in[n]}$ sends x_i to the trusted party. The trusted party then returns $f(x_1, \ldots, x_n)$ to all the parties.

For all $i \in H$, let OUT_i denote the output returned to P_i by the third party, and for $i \in I \cup C$, let OUT'_i denote some value output by party P_i. The i^{th} partial output of a ideal-world execution of Π between parties $(P_1, \ldots, P_n)$ in the presence of independent simulators $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_m)$ is defined as

$$\mathsf{IDEAL}^{i}_{\Pi,\mathcal{S},I,C,z_i}(k, x_i) \stackrel{\mathsf{def}}{=} \{\mathsf{OUT}_j \mid j \in H\} \cup \mathsf{OUT}'_i$$

Definition 2.2.1. (Semi-Honest Security) Suppose f is a deterministic-time n-party functionality (deterministic in all cases considered in this paper), and Π is the protocol. Let x_i be the parties' respective private inputs to the protocol. Let $I \in [n]$ denote the set of corrupted and non-colluding parties and $C \in [n]$ denote the set of corrupted and colluding parties. We say that protocol $\Pi(I, C)$ securely computes deterministic functionality f if there exist probabilistic polynomial-time simulators $\mathsf{Sim}_{i\in m}$ for $m < n$ such that all adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_m)$, for all $\bar{x} \leftarrow \{0,1\}^\star$ and $\bar{z} \leftarrow \{0,1\}^\star$, and for all $i \in [m]$,

$$\{\mathsf{REAL}^{i}_{\Pi,\mathcal{A},I,C,\bar{z}}(k, \bar{x}) \stackrel{c}{=} \{\mathsf{IDEAL}^{i}_{\Pi,\mathsf{Sim},I,C,\bar{z}}(k, \bar{x})\}$$

Where $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_m)$ and $\mathcal{S} = \mathsf{Sim}_i(\mathcal{A}_i)$

Definition 2.2.2. (Malicious Security) Suppose f is a deterministic-time n-party

functionality (deterministic in all cases considered in this paper), and Π is the protocol. Let x_i be the parties' respective private inputs to the protocol. Let $I \in [n]$ denote the set of corrupted and non-colluding parties and $C \in [n]$ denote the set of corrupted and colluding parties. We say that protocol $\Pi(I, C)$ securely computes deterministic functionality f if there exist probabilistic polynomial-time simulators $\mathsf{Sim}_{i \in m}$ for $m < n$ such that all *all probabilistic polynomial time* adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_m)$, for all $\bar{x} \leftarrow \{0,1\}^\star$ and $\bar{z} \leftarrow \{0,1\}^\star$, and for all $i \in [m]$,

$$\{\mathsf{REAL}^i_{\Pi,\mathcal{A},I,C,\bar{z}}(k,\bar{x}) \widetilde{=} \{\mathsf{IDEAL}^i_{\Pi,\mathsf{Sim},I,C,\bar{z}}(k,\bar{x})\}$$

Where $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_m)$ and $\mathcal{S} = \mathsf{Sim}_i(\mathcal{A}_i)$

2.3 Oblivious transfer

Oblivious Transfer (OT) has been a central primitive in the area of secure computation. Indeed, the original protocols of Yao [Yao86] and GMW [Gol04, GMW87] both use OT in a critical manner. In fact, OT is both necessary and sufficient for secure computation [Kil88]. Until early 2000's, the area of generic secure computation was often seen mainly as a feasibility exercise, and improving OT performance was not a priority research direction. This changed when Yao's Garbled Circuit (GC) was first implemented [MNPS04] and a surprisingly fast OT protocol (which we will call IKNP) was devised by Ishai et al. [IKNP03].

The IKNP OT extension protocol [IKNP03] is truly a gem; it allows 1-out-of-2 OT execution at the cost of computing and sending only a few hash values

(but a security parameter of public key primitives evaluations were needed to bootstrap the system). IKNP was immediately noticed and since then universally used in implementations of the Yao and GMW protocols. It took a few years to realize that OT extension's use goes far beyond these fundamental applications. Many aspects of secure computation were strengthened and sped up by using OT extension. For example, Nielsen et al. [NNOB12] propose an approach to malicious two-party secure computation, which relates outputs and inputs of OTs in a larger construction. They critically rely on the low cost of batched OTs. Another example is the application of information-theoretic Gate Evaluation Secret Sharing (GESS) [Kol05] to the computational setting [KK12]. The idea of [KK12] is to stem the high cost in secret sizes of the GESS scheme by evaluating the circuit by shallow slices, and using OT extension to efficiently "glue" them together. Particularly relevant for our work, efficient OTs were recognized by Pinkas et al. [PSZ14] as an effective building block for private set intersection, which we discuss in more detail later.

The IKNP OT extension, despite its wide and heavy use, received very few updates. In the semi-honest model it is still state-of-the-art. Robustness was added by Nielsen [Nie07], and in the malicious setting it was improved only very recently [ALSZ15, KOS15]. Improvement for short secret sizes, motivated by the GMW use case, was proposed by Kolesnikov and Kumaresan [KK13]. We use ideas from their protocol, and refer to it as the KK protocol. Under the hood, KK [KK13] noticed that one core aspect of IKNP data representation can be abstractly seen as a repetition error-correcting code, and their improvement stems from using

PARAMETERS: Sender $\mathcal{S}$, receiver $\mathcal{R}$, length κ
FUNCTIONALITY:

- Wait for an input $b \leftarrow \{0,1\}$ from the receiver $\mathcal{R}$.
- Choose $\boldsymbol{m_0}, \boldsymbol{m_1} \leftarrow \{0,1\}^\kappa$, and give both to sender $\mathcal{S}$.
- Give $\boldsymbol{m_b}$ to receiver $\mathcal{R}$.

Figure 2.1: The $\mathcal{F}_{\mathsf{ROT}}^{\kappa}$ ideal functionality for Random Oblivious Transfer.

a better code. As a result, instead of 1-out of-2 OT, a 1-out of-n OT became possible at nearly the same cost, for n up to approximately 256. In the same year, [ALSZ13] presented several IKNP optimizations and several weaker variants of OT. In Random OT (ROT), the sender's OT inputs $(\boldsymbol{m_0}, \boldsymbol{m_1})$ are chosen at random, therefore, it allows the protocol itself to give him the values $(\boldsymbol{m_0}, \boldsymbol{m_1})$ randomly. With ROT, the bandwidth requirement is significantly reduced since the sender sends nothing to receiver. In our construction, we require this weaker variant, random OT, whose functionality is described in Figure 2.1.

2.4 Oblivious PRFs

An oblivious pseudorandom function (OPRF) [FIPR05] is a protocol in which a sender learns (or chooses) a random PRF seed s while the receiver learns $F(s, r)$, the result of the PRF on a single input r chosen by the receiver. While the general definition of an OPRF allows the receiver to evaluate the PRF on several inputs, in this paper we consider only the case where the receiver can evaluate the PRF *on a single input.*

PARAMETERS: A PRF F and bound t.

BEHAVIOR: Wait for input $(q_1, \ldots, q_t)$ from the receiver $\mathcal{R}$. Sample a random PRF seed k and give it to the sender $\mathcal{S}$. Give $(F(k, q_1), \ldots, F(k, q_t))$ to the receiver.

Figure 2.2: The OPRF ideal functionality $\mathcal{F}_{\mathsf{oprf}}^{F,t}$

The central primitive of this work, an efficient OPRF protocol, can be viewed as a variant of Oblivious Transfer (OT) of random values. We build it by modifying the core of OT extension protocols [IKNP03, KK13], and its internals are much closer to OT than to prior works on OPRF. Therefore, our presentation is OT-centric, with the results stated in OPRF terminology.

OT of random messages shares many properties with OPRF. In OT of random messages, the sender learns random m_0, m_1 while the receiver learns m_r for a choice bit $r \in \{0, 1\}$. One can think of the function $F((m_0, m_1), r) = m_r$ as a pseudorandom function with input domain $\{0, 1\}$. Similarly, one can interpret 1-out-of-n OT of random messages as an OPRF with input domain $\{1, \ldots, n\}$. The OPRF functionality is described in Figure 2.2.

2.5 Cuckoo Hashing

We review the basics of Cuckoo hashing [PR01], specifically the variant of Cuckoo hashing that involves a stash [KMW08]. In basic Cuckoo hashing, there are m bins, a *stash*, and several random hash functions $h_1, \ldots, h_k$ (often $k = 2$), each with range $[m]$. The invariant is that any item x stored in the Cuckoo hash table is stored either in the stash or (preferably) in one of the bins $\{h_1(x), \ldots, h_k(x)\}$.

Each non-stash bin holds at most one item. To insert and element x into a Cuckoo hash table, we place it in bin $h_i(x)$, if this bin is empty for any i. Otherwise, choose a random $i \in_R [k]$, place x in bin $h_i(x)$, evict the item currently in $h_i(x)$, and recursively insert the evicted item. After a fixed number of evictions, give up and place the current item in the stash.

2.6 Private Matching and Its Extension

2.6.1 Private Matching (PM)

We present a special case of PM in which the sender with input string $\boldsymbol{x}_0$ interacts with a receiver with input string $\boldsymbol{x}_1$ in the following way. The receiver learns a bit indicating whether $\boldsymbol{x}_0 = \boldsymbol{x}_1$ and nothing else, while the sender learns nothing about $\boldsymbol{x}_1$. We describe the ideal functionality for this special PM in Figure 2.3.

To our knowledge, PM was first introduced in 1996 by Fagin, Naor, and Winkler [FNW96]. Follow-up works[NP99, BST01, Lip03] improved the efficiency of PM, while still relying on expensive public-key operations. PM is heavily used in two-party private set intersection (PSI) protocols [FNP04].

2.6.2 Private Set Intersection (PSI)

PSI is a special case of secure PM computation. The guarantees of PSI are captured in the ideal functionality $\mathcal{F}_{\mathsf{PSI}}$ defined in Figure 2.4. For security against malicious parties, we use the framework of universal composability (UC) [Can01].

PARAMETERS: Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$

FUNCTIONALITY:

- Wait for input $\boldsymbol{x}_0 \in \{0,1\}^*$ from the sender $\mathcal{S}$.
- Wait for input $\boldsymbol{x}_1 \in \{0,1\}^*$ from the receiver $\mathcal{R}$.
- Give the receiver $\mathcal{R}$ output 1 if $\boldsymbol{x}_0 = \boldsymbol{x}_1$ and 0 otherwise.

Figure 2.3: The Private Matching ideal functionality $\mathcal{F}_{\mathsf{pm}}$

PARAMETERS: The number of parties n, and the size of the parties' sets m.

FUNCTIONALITY:

- Wait for an input $X_i = \{x_i^1, \ldots, x_i^m\} \subseteq \{0,1\}^*$ from each party P_i.
- Give output $\bigcap_{i=1}^{n} X_i$ to all parties.

Figure 2.4: PSI ideal functionality.

2.6.3 Private Set Union (PSU)

Private set union (PSU) is a special case of secure two-party computation. PSU allows two parties holding sets X and Y respectively, to compute the union $X \cup Y$, without revealing anything else, namely what are the items in the intersection of X and Y. The guarantees of PSU are captured in the ideal functionality $\mathcal{F}_{\mathsf{PSI}}$ defined in Figure 2.5.

PARAMETERS: Set sizes m and n; two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$

FUNCTIONALITY:

- Wait for an input $X = \{x_1, x_2, \ldots, x_n\} \subseteq \{0,1\}^*$ from sender $\mathcal{S}$, and an input $Y = \{y_1, y_2, \ldots, y_m\} \subseteq \{0,1\}^*$ from receiver $\mathcal{R}$
- Give output $X \cup Y$ to the receiver $\mathcal{R}$.

Figure 2.5: Private Set Union Functionality $\mathcal{F}_{\mathsf{psu}}^{m,n}$.

PARAMETERS: A text length n, a pattern length m, and two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$

FUNCTIONALITY:

- Wait for text $\boldsymbol{x} \in \{0,1\}^n$ from the sender $\mathcal{S}$
- Wait for pattern $\boldsymbol{p} \in \{0,1,\star\}^m$ from the receiver $\mathcal{R}$
- Give the receiver $\mathcal{R}$ output $\{i \in [n-m+1] \mid \boldsymbol{p} \preceq \boldsymbol{x}_{[i,i+m-1]}\}$ (see Section 2.1 for notation)

Figure 2.6: Wildcard Pattern Matching functionality $\mathcal{F}_{\mathsf{wpm}}^{n,m}$.

2.6.4 Wildcard Pattern Matching (WPM)

In secure pattern matching with *wildcards*, which we will call WPM, the receiver's pattern can include wildcard characters that can match any character in the data, hence $p \in (\Sigma \cup \{\star\})^m$. With wildcards, the security requirements are more demanding: the server should not learn which positions of p contain wildcards, and in the case of a match the receiver should not learn the text character that matches a wildcard character in the pattern (unless this could be inferred from the presence or absence of an overlapping match). The ideal functionality WPM is defined in Figure 2.6.

2.7 Correlation Robustness

The OT extension protocol of IKNP [IKNP03] is proven secure under a so-called *correlation robustness* assumption on the underlying hash function. Our protocol makes use of the following generalization of this notion:

Definition 2.7.1. Let H be a hash function with input length n. Then H is d**-Hamming correlation robust** if, for any strings $z_1, \ldots, z_m \in \{0,1\}^*$, $a_1, \ldots, a_m, b_1, \ldots, b_m \in \{0,1\}^n$ with $d_H(b_i, \geq)d$, the following distribution, induced by random sampling of $s \leftarrow \{0,1\}^n$, is pseudorandom:

$$H(z_1 \| a_1 \oplus [b_1 \cdot s]), \ldots, H(z_m \| a_m \oplus [b_m \cdot s])$$

As in the overview, "$\cdot$" denotes bitwise-AND.

The definition generalizes previous ones in the following way:

- If $d = n$, then the only legal choice of b_i is 1^n, and $H(z_i \| a_i \oplus [b_i \cdot s])$ simplifies to $H(z_i \| a_i \oplus s)$. Restricting the definition in this way, and taking $z_i = i$ corresponds to the IKNP notion of correlation robustness.

- If the b_i values are required to be elements of a linear error correcting code $\mathcal{C}$, then the resulting definition is one under which the construction of [KK13] is secure (for simplicity they prove security in the random oracle model).

Chapter 3: Two-party PSI

Efficient Batched Oblivious PRF with Applications to Private Set Intersection by Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, Ni Trieu, in CCS [KKRT16].

SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension by Benny Pinkas, Mike Rosulek, Ni Trieu, Avishay Yanai, in CRYPTO [PRTY19]

3.1 Introduction

Private set intersection (PSI) allows two parties, who each hold a set of items, to learn the intersection of their sets without revealing anything else about the items. PSI has many privacy-preserving applications: e.g., private contact discovery [CLR17, RA17, DRRT18][1], DNA testing and pattern matching [TKC07], remote diagnostics [BPSW07], record linkage [HMFS17], and measuring the effectiveness of online advertising [IKN+17]. Over the last several years PSI has become truly practical with extremely fast implementations [CKT10, CGT12, DCW13, PSZ14, KKRT16, KMP+17, RR17b, CLR17, HV17, RA17, GNN17, CCS18] that can process millions of items in seconds.

[1] See also https://whispersystems.org/blog/contact-discovery/

The standard ways to measure the cost of a protocol are running time and communication. Depending on which of these metrics is prioritized, a different protocol will be preferred.

Minimizing time. The fastest known PSI protocols are all based on efficient oblivious transfers (OT). The idea is to reduce the PSI computation to many instances of oblivious transfer. This approach is the fastest because modern OT extension protocols [Bea96, IKNP03, ALSZ13] use only a small (fixed) number of public-key operations (e.g., elliptic curve multiplications) but otherwise use only *cheap symmetric-key operations.* The approach to PSI was introduced by Pinkas et al. [PSZ14] and refined in a sequence of works [PSSZ15].

Minimizing communication. To the best of our knowledge, the PSI protocol with lowest communication in this setting is due to Ateniese et al. [ACT11]. This protocol requires communication that is only marginally more than a naïve and *insecure* protocol (in which one party sends just a short hash of each item), and also has the nice property of hiding the size of the input set. However, the protocol requires at least $n \log n$ RSA exponentiations (for PSI of n items). These requirements make the protocol prohibitively expensive in practice.[2]

A more popular (as well as the earliest) approach to low-communication PSI is based on the commutative property of Diffie-Hellman key agreement (DH-PSI), and appears in several works [Sha80, Mea86, HFH99]. The idea is for the parties

[2]We are not aware of any prior implementation of this protocol, but estimated the running time through benchmark RSA exponentiations. For the set sizes we consider in this work, the protocol would require many hours or even a day.

to compute the intersection of $\{(H(x)^\alpha)^\beta) \mid x \in X\}$ and $\{(H(y)^\beta)^\alpha) \mid y \in Y\}$ in the clear, where α and β are secrets known by Alice and Bob, respectively. The DH-PSI protocol strikes a more favorable balance between communication and computation than the RSA-based protocol. It requires n exponentiations in a Diffie-Hellman group, which are considerably cheaper than RSA exponentiations but considerably more expensive than the symmetric-key operations used in OT extension. In terms of communication, it requires less than 3 group elements per item. When instantiated with compact elliptic curve groups (ECDH-PSI), the communication complexity is very small. For example, Curve25519 [Ber06] provides 128-bit security with only 256-bit group elements (around 600 bits of communication per item).

An ideal balance. Communication cost and overall running time are clearly both important, but which metric best reflects the balance between the two costs, and the true suitability of a protocol for practice? We argue that the most appropriate metric which balances the two costs is the monetary cost to run the protocol on a cloud computing service. First, a typical real-world application of PSI is likely to use such a service rather than in-house computing. Second, the pricing model of such services already takes into account the difference in cost to send a bit vs. perform a CPU clock cycle.

3.2 Related Work and Comparison

DH-PSI Our protocol uses less communication than DH-PSI, even when the latter is instantiated with the most compact elliptic curve known. In terms of computation, our protocol uses only symmetric-key operations (apart from a fixed number of base OTs). Its main computational bottleneck is computing polynomial interpolation, requiring either $O(n \log^2 \lambda)$ or $O(n \log^2 n)$ finite field operations (i.e., multiplications), depending on the variant, where n is the set size and λ is the statistical security parameter. The DH-PSI protocol computes $O(n)$ exponentiations (or elliptic curve multiplications, which are each computed using $\log |\mathbb{G}|$ multiplication operations in the underlying cyclic group $\mathbb{G}$). If we consider the basic unit of computation to be a multiplication in the underlying field/group, then our protocol uses at most $O(n \log^2 n)$ multiplications whereas DH-PSI uses $O(n \log |\mathbb{G}|)$ multiplications. The experiments that we describe in Section 3.4.5 demonstrate that our protocol is substantially faster than DH-PSI for all realistic set sizes and on all network configurations.

Our communication-optimized protocol variant has security against one malicious party. In contrast, DH-PSI is not easily adapted to malicious security, even against just one party.[3] In order to harden DH-PSI against malicious parties, the leading protocol of De Cristofaro et al. [CKT10] requires *both parties* to run zero-knowledge proofs involving all of their input items. Thus, even one-sided malicious security requires significant overhead to the semi-honest protocol.

[3]The main challenge is that a simulator would have to extract effective inputs $\{x_1, \ldots, x_n\}$ from a corrupt party, seeing only $\{H(x_1)^\alpha, \ldots, H(x_n)^\alpha\}$.

While we do not formally consider security against quantum adversaries, we do point out that our protocol exclusively uses primitives that can be instantiated with post-quantum security (OT, PRFs, and hash functions). DH-PSI on the other hand is trivially broken against quantum adversaries.

Protocols based on an RSA accumulator. The protocol of [ACT11] has a very low communication overhead of roughly $\lambda + \log^2 n$ bits per item, which may even be optimal (even for an insecure protocol). On the other hand, it computes $O(n \log n)$ RSA exponentiations, and as such is slower than DH-PSI by at least an order of magnitude (due to the $\log n$ factor, and to RSA exponentiations being slower than elliptic curve multiplications). Our protocols are substantially faster than both of these protocols (see Section 3.4.5). This protocol also requires a random oracle, whereas for semi-honest security ours is in the standard model.

OT-based protocols In our CCS paper [KKRT16], the proposed protocol computes an intersection of million-item sets in about 4 seconds. Our CRYPTO paper [PRTY19] requires 40-50% less communication compared to [KKRT16] and is the fastest over low-bandwidth networks (30 Mbps and lower). Over high-bandwidth networks, even though ours [PRTY19] is slower than [KKRT16], this protocol still requires less monetary cost (see Section 3.4.5).

Independently, Lambæk [Lam16] and Patra et al. [PSS17] showed how to enhance the protocols of [PSZ14, KKRT16] with a security against a *malicious receiver* with almost no additional overhead. Interestingly, our protocol naturally provides security against a *malicious sender*. In both of these protocols, if the

parties have sets of very different sizes then the party with more items should play the role of sender. Providing a different flavor of one-sided malicious security is therefore potentially valuable.

Ghosh and Nilges [GN19] proposed a PSI protocol based on oblivious linear function evaluation (OLE). This protocol requires $2n$ passive OLE invocations, polynomial interpolations at 3 times (one of degree n, and two of degree $2n$), and polynomial evaluation on $2n + 1$ points at 4 times. In terms of communication, the required passive OLE instances [IPS09, GNN17] require $8(n+1)$ elements sent from the receiver to the sender to create a noisy encoding, and the cost of doing $4n$-out-of-$8(n+1)$ OT which incurs an overhead of at least $8(n+1)$ on the number of Correlated OT [ALSZ13]. Hence, this OLE-based PSI protocol requires at least $8(n + 1)(\kappa + 2\ell)$ bits communication, where ℓ is bit-length of item. For example, when $\ell = 128$, our protocols show a factor of $4.8 - 6.3\times$ improvement in terms of communication.

Recently, Falk, Noble and Ostrovsky [FNO18] presented a protocol for PSI that achieves linear communication complexity relying on standard assumption (i.e. in the OT-hybrid model, assuming the existence of correlation robust hash and one-way functions) and in the standard model (i.e. without a random oracle). This is in contrast to previous protocols that achieve linear communication but rely on stronger assumptions (like [CKT10, CT12] that are based on the one-more RSA assumption and a random oracle); and to previous OT-based protocols that achive only super-linear communication complexity due to the stash handling. In the protocol of [FNO18], just like previous OT-based protocols, Bob maps his n

items to $O(n)$ bins using a Cuckoo hashing, hence, it has at most one item in each bin. Bob also maintains a special bin for items that could not be mapped to the 'regular' bins, this special bin is called the *stash* and it contains $\omega(1)$ items. Alice maps her items to $O(n)$ bins using simple hashing, hence, she has at most $O(\log n/\log\log n)$ items in each bin with high probability. Then, Bob can obtain the intersection between items in its 'regular' bins and Alice's set using the BaRK-OPRF technique of [KKRT16] with communication complexity $O(n \cdot \kappa)$ (where κ is the computational security parameter). It remains to compare the items in Bob's stash to all Alice's items; since the stash is of size $\omega(1)$ this comparison would naively require $\omega(n \cdot \kappa)$ communication overall. However, the observation in [FNO18] is that this comparison can be performed using a separate PSI protocol that is specialized for *unbalanced* set sizes in which Alice has much more items than Bob; such a protocol can achieve communication complexity that depends only on the larger set size, therefore, the overall communication complexity of [FNO18] is $O(n \cdot \kappa)$ rather than $\omega(n \cdot \kappa)$. We note that in concurrent to their work, the protocol [PRTY19] achieves the same (linear) communication complexity, under the same standard assumptions and without a random oracle, using a new primitive, namely the *Sparse OT Extension*.

Other paradigms. Other approaches for PSI have been proposed, including ones based on Bloom filters [DCW13] and generic MPC [HEK12]. Pinkas et al. [PSZ14, PSSZ15] performed a comprehensive comparison of semi-honest PSI techniques and found the OT-extension paradigm to strictly dominate others in terms of performance. They found that the best Bloom-filter approach is 2x worse

in runtime, 4x worse in communication; best generic-MPC-based approach is 100x worse in runtime and 10x worse in communication. For this reason, we do not include these protocol paradigms in further comparisons.

Other related work. One way of viewing our new technique is that we covertly embed some protocol messages into a polynomial. Similar ideas appear in [MPP10, CDJ16]. In particular, [CDJ16] explicitly propose to embed private equality-test protocol messages into a polynomial, to yield a PSI protocol. Their protocol is based on the DH paradigm, and therefore requires a linear number of exponentiations. They also achieve a stronger *covertness* property (participants cannot distinguish other participants from random noise, until the protocol terminates). In our case, we look inside IKNP OT extension and identify the minimal part of the protocol that needs to be covertly embedded into a polynomial, in order to achieve *standard* (semi-honest or malicious) security.

3.3 Efficient Batched Oblivious PRF with Applications to Private Set Intersection

We describe a lightweight protocol for oblivious evaluation of a pseudorandom function (OPRF) in the presence of semi-honest adversaries. In an OPRF protocol a receiver has an input r; the sender gets output s and the receiver gets output $F(s, r)$, where F is a pseudorandom function and s is a random seed. Our protocol uses a novel adaptation of 1-out-of-2 OT-extension protocols, and is particularly efficient when used to generate a large batch of OPRF instances. The cost to

realize m OPRF instances is roughly the cost to realize $3.5m$ instances of standard 1-out-of-2 OTs (using state-of-the-art OT extension).

We explore in detail our protocol's application to semi-honest secure private set intersection (PSI). The fastest state-of-the-art PSI protocol (Pinkas et al., Usenix 2015) is based on efficient OT extension. We observe that our OPRF can be used to remove their PSI protocol's dependence on the bit-length of the parties' items. We implemented both PSI protocol variants and found ours to be 3.1–3.6× faster than Pinkas et al. for PSI of 128-bit strings and sufficiently large sets. Concretely, ours requires only 3.8 seconds to securely compute the intersection of 2^{20}-size sets, regardless of the bit length of the items. For very large sets, our protocol is only 4.3× slower than the *insecure* naïve hashing approach for PSI.

3.3.1 Technical Overview of Our Results

We start with the OT-extension paradigm of Ishai, Kilian, Nissim & Petrank (IKNP) [IKNP03]. The goal of OT extension is to use a small number k of "base-OTs," plus only symmetric-key operations, to achieve $m \gg k$ "effective OTs." Here, k is chosen depending on the computational security parameter κ; in the following we show to what value k should be set. Below we describe an OT extension that achieves m 1-out-of-2 OTs of random strings, in the presence of semi-honest adversaries.

We follow the notation of [KK13] as it explicates the coding-theoretic framework for OT extension. Suppose the receiver has choice bits $r \in \{0,1\}^m$. He

chooses two $m \times k$ matrices (m rows, k columns), T and U. Let $\boldsymbol{t}_j, \boldsymbol{u}_j \in \{0,1\}^k$ denote the j-th row of T and U, respectively. The matrices are chosen at random, so that:

$$\boldsymbol{t}_j \oplus \boldsymbol{u}_j = r_j \cdot 1^k \stackrel{\text{def}}{=} \begin{cases} 1^k & \text{if } r_j = 1 \\ 0^k & \text{if } r_j = 0 \end{cases}$$

The sender chooses a random string $s \in \{0,1\}^k$. The parties engage in k instances of 1-out-of-2 string-OT, *with their roles reversed*, to transfer to sender $\mathcal{S}$ the columns of either T or U, depending on the sender's bit s_i in the string s it chose. In the i-th OT, the receiver gives inputs $\boldsymbol{t}^i$ and $\boldsymbol{u}^i$, where these refer to the i-th *column* of T and U, respectively. The sender uses s_i as its choice bit and receives output $\boldsymbol{q}^i \in \{\boldsymbol{t}^i, \boldsymbol{u}^i\}$. Note that these are OTs of strings of length $m \gg k$ — the *length* of OT messages is easily extended. This can be done, e.g., by encrypting and sending the two m-bit long strings, and using OT on short strings to send the right decryption key.

Now let Q denote the matrix obtained by the sender, whose columns are $\boldsymbol{q}^i$. Let $\boldsymbol{q}_j$ denote the jth row. The key observation is that

$$\boldsymbol{q}_j = \boldsymbol{t}_j \oplus [r_j \cdot s] = \begin{cases} \boldsymbol{t}_j & \text{if } r_j = 0 \\ \boldsymbol{t}_j \oplus s & \text{if } r_j = 1 \end{cases} \tag{3.1}$$

Let H be a random oracle (RO). We have that the sender can compute two random strings $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$, of which the receiver can compute only one, via $H(\boldsymbol{t}_j)$. Note that $\boldsymbol{t}_j$ equals either $\boldsymbol{q}_j$ or $\boldsymbol{q}_j \oplus s$, depending on the receiver's choice bit r_j.

Note that the receiver has no information about s, so intuitively he can learn only one of the two random strings $H(\boldsymbol{q}_j), H(\boldsymbol{q}_j \oplus s)$. Hence, each of the m rows of the matrix can be used to produce a single 1-out-of-2 OT.

As pointed out by [IKNP03], it is sufficient to assume that H is a correlation-robust hash function, a weaker assumption than RO. A special assumption is required because the same s is used for every resulting OT instance. See Section 3.3.2 for definition of correlation-robustness.

Coding interpretation In IKNP, the receiver prepares secret shares of T and U such that each row of $T \oplus U$ is either all zeros or all ones. Kolesnikov & Kumaresan [KK13] interpret this aspect of IKNP as a *repetition code* and suggest to use other codes instead.

Consider how we might use the IKNP OT extension protocol to realize 1-out-of-2^ℓ OT. Well, instead of a choice bit r_i for the receiver, r_i will now be an ℓ-bit string. Let C be a linear error correcting code of dimension ℓ and codeword length k. The receiver will prepare matrices T and U so that $\boldsymbol{t}_j \oplus \boldsymbol{u}_j = C(r_j)$.

Now, generalizing Equation 3.1 the sender receives

$$\boldsymbol{q}_j = \boldsymbol{t}_j \oplus [C(r_j) \cdot s] \tag{3.2}$$

where "$\cdot$" now denotes bitwise-AND of two strings of length k. (Note that when C is a repetition code, this is exactly Equation 3.1.)

For each value $r' \in \{0,1\}^\ell$, the sender associates the secret value $H(\boldsymbol{q}_j \oplus [C(r') \cdot s])$, which it can compute for all $r' \in \{0,1\}^\ell$. At the same time, the receiver can

compute one of these values, namely, $H(\boldsymbol{t}_j)$. Rearranging Equation 3.2, we have:

$$H(\boldsymbol{t}_j) = H(\boldsymbol{q}_j \oplus [C(r_j) \cdot s])$$

Hence, the value that the receiver can learn is the secret value that the sender associates with the receiver's choice string $r' = r_j$.

At this point, OT of random strings is completed. For OT of chosen strings, the sender will use each $H(\boldsymbol{q}_i \oplus [C(r) \cdot s])$ as a key to encrypt the r'th OT message. The receiver will be able to decrypt only one of these encryptions, namely one corresponding to its choice string r_j.

To argue that the receiver learns *only one* string, suppose the receiver has choice bits r_j but tries to learn also the secret $H(\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s])$ corresponding to a different choice $\tilde{r}$. We observe:

$$\begin{aligned} \boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s] &= \boldsymbol{t}_j \oplus [C(r_j) \cdot s] \oplus [C(\tilde{r}) \cdot s] \\ &= \boldsymbol{t}_j \oplus [(C(r_j) \oplus C(\tilde{r})) \cdot s] \end{aligned} \tag{3.3}$$

Importantly, everything in this expression is known to the receiver except for s. Now suppose the minimum distance of C is κ (the security parameter). Then $C(r_j) \oplus C(\tilde{r})$ has Hamming weight at least κ. Intuitively, the adversary would have to guess at least κ bits of the secret s in order to violate security. The protocol is secure in the RO model, and can also be proven under the weaker assumption of correlation robustness, following [IKNP03, KK13].

Finally, we remark that the width k of the OT extension matrix is equal to the

length of codewords in C. The parameter k determines the number of base OTs and the overall cost of the protocol.

Pseudorandom codes The main technical observation we make in this work is pointing out that the code C need not have many of the properties of error-correcting codes. In particular,

- We make no use of decoding, thus our code does not need to be efficiently decodable.
- We require only that for all possibilities r, r', the value $C(r) \oplus C(r')$ has Hamming weight at least equal to the computational security parameter κ. In fact, it is sufficient even if the Hamming distance guarantee is only probabilistic — i.e., it holds with overwhelming probability over choice of C (we discuss subtleties below).

For ease of exposition, imagine letting C be a random oracle with suitably long output. (Later we will show that C can be instantiated from a pseudorandom function in a straight-forward way.) Intuitively, when C is sufficiently long, it should be hard to find a "near-collision." That is, it should be hard to find values r and r' such that $C(r) \oplus C(r')$ has low (less than a computational security parameter κ) Hamming weight. Later in Table 5.2 we compute the parameters more precisely, but for now we simply point out that a random function with output length $k = 4\kappa$ suffices to make near-collisions negligible in our applications.

We refer to such a function C (or family of functions, in our standard-model

instantiation) as a **pseudorandom code (PRC)**, since its coding-theoretic properties — namely, minimum distance — hold in a cryptographic sense.

By relaxing the requirement on C from an error-correcting code to a pseudorandom code, we *remove the a-priori bound* on the size of the receiver's choice string! In essence, the receiver can use *any string* as its choice string; the sender can associate a secret value $H(\boldsymbol{q}_j \oplus [C(r') \cdot s])$ for any string r'. As discussed above, the receiver is only able to compute $H(\boldsymbol{t}_j) = H(\boldsymbol{q}_j \oplus [C(r) \cdot s])$ — the secret corresponding to its choice string r. The property of the PRC is that, with overwhelming probability, all other values of $\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s]$ (that a polytime player may ever ask) differ from $\boldsymbol{t}_j$ in a way that would require the receiver to guess at least κ bits of s.

Interpretation as an oblivious PRF variant We can view the functionality achieved by this protocol as a kind of oblivious PRF. Intuitively, $r \mapsto H(\boldsymbol{q} \oplus [C(r) \cdot s])$ is a function that the sender can evaluate on any input, whose outputs are pseudorandom, and which the receiver can evaluate only on its chosen input r.

In Section 3.3.2 we give a formal definition of the functionality that we achieve. The main subtleties of the definition are:

1. the fact that the receiver learns slightly more than the output of this "PRF" — in particular, the receiver learns $\boldsymbol{t} = \boldsymbol{q} \oplus [C(r) \cdot s]$ rather than $H(\boldsymbol{t})$;

2. the fact that the protocol realizes many instances of this "PRF" but with related keys — s and C are shared among all instances.

We prove our construction secure assuming C is a pseudorandom code and that H satisfies a natural generalization of the "correlation robust" assumption from [IKNP03].

Summary & cost With our new variant of the IKNP protocol, we can obtain m OPRF instances efficiently, using only k base OTs plus symmetric-key operations. Compared to IKNP-paradigm OT extension for 1-out-of-2 OTs, the main differences in cost are:

- Cost associated with the increased width of the OT extension matrices. In our case, the matrix has width k rather than κ — concretely $3\kappa < k < 4\kappa$ in our applications. Note that the parameter k controls the number of base OTs required.[4]

- Computational costs associated with the pseudorandom code C. While in IKNP C is a repetition code, and in [KK13] C is a short Walsh-Hadamard code, in our protocol C is cryptographic. However, we are able to instantiate C using a PRF. In practice, we use AES as the PRF, and the associated hardware acceleration for AES in modern processors makes the cost of computing C minimal.

Application to private set intersection Private set intersection (PSI) refers to a computation in which Alice has a set A of items, Bob has a set B of items,

[4] In our instantiation, we actually use IKNP to extend κ base OTs to k OTs, and then use those k OTs as base OTs for BaRK-OPRF instances. Hence, the number of public-key OT operations is unchanged. Still, the total communication cost remains proportional to km in our protocol rather than κm.

and the two learn only $A \cap B$ and nothing more.

We show how BaRK-OPRF can be used to significantly reduce the cost of semi-honest-secure PSI. The current fastest protocol for the task is that of Pinkas et al. [PSSZ15]. The protocol relies heavily on efficient OT extension (for standard 1-out-of-2 OTs).

Looking closely at the PSI protocol of [PSSZ15], we see that they use a number of OTs that is proportional to $N\ell$, where N is the number of items in the parties' sets and ℓ is the length (in bits) of those items. We can replace their use of 1-out-of-2 OTs with a suitable use of BaRK-OPRF and remove the dependence on ℓ. Our protocol uses a number of BaRK-OPRF instances that is proportional only to N.

We implemented our BaRK-OPRF-based PSI protocol and compared its performance to that of [PSSZ15]. For PSI on strings of length $\ell \in \{64, 128\}$ and sufficiently large sets, our protocol is 2.3–3.6 times faster. This is a significant achievement in the already very polished PSI state of the art.

3.3.2 Our Oblivious PRF Variant

3.3.2.1 Pseudorandom Codes

We now formalize the notion of a pseudorandom code, motivated in Section 3.3.1.

Definition 3.3.1. Let $\mathcal{C}$ be a family of functions. We say that $\mathcal{C}$ is a (d, ϵ) **pseu-**

dorandom code (PRC) if for all strings $x \neq x'$,

$$\Pr_{C \leftarrow \mathcal{C}} \Big[d_H(C(x) \oplus C(x'), <) d \Big] \leq 2^{-\epsilon}$$

That is, a (d, ϵ)-PRC guarantees that the hamming distance of two codewords is less or equal to d with probability at most $2^{-\epsilon}$.

The reader may find it convenient to think of C as a random oracle. However, it suffices for C to be a pseudorandom function instantiated with random seed:

Lemma 1. Suppose $F : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^n$ is a pseudorandom function. Define $\mathcal{C} = \{F(s, \cdot) \mid s \in \{0,1\}^\kappa\}$. Then $\mathcal{C}$ is a (d, ϵ)-pseudorandom-code where:

$$2^{-\epsilon} = 2^{-n} \sum_{i=0}^{d-1} \binom{n}{i} + \nu(\kappa).$$

and ν is a negligible function.

Proof. Consider the following game. An adversary has strings x and x' hard-coded. It queries its oracle $\mathcal{O}$ on x and x' and outputs 1 if $\mathcal{O}(x)$ and $\mathcal{O}(x')$ are within Hamming distance d.

When $\mathcal{O}$ is instantiated as a random function, a simple counting argument shows that the adversary outputs 1 with probability $2^{-n} \sum_{i=0}^{d-1} \binom{n}{i}$.

When $\mathcal{O}$ is instantiated as a PRF F with random seed, the probability must be within $\nu(\kappa)$ of the above probability, where ν is negligible. The adversary's output probability in this instantiation is exactly the probability specified in the PRC security definition, so the lemma follows.

Note that in our typical usage of PRCs, the choice of C (in this case, the seed to the PRF) is a public value. But in both the security definition for PRC and in this analysis, the values x and x' are fixed before the PRF key is chosen. Whether or not $F(s, x)$ and $F(s, x')$ are within Hamming distance d is not affected by making the PRF seed public. □

3.3.2.2 Our Oblivious PRF Variant

As outlined in Section 3.3.1, our main construction is a variant of OT-extension which associates a pseudorandom output $R(x)$ for every possible input $r \in \{0,1\}^*$. The sender can compute $R(r)$ for any r, while the receiver learns $R(x)$ for only a single value r. This functionality is reminiscent of an **oblivious PRF (OPRF)** [FIPR05]. In this section we describe how our construction can be interpreted as a variant OPRF functionality.

In an OPRF functionality for a PRF F, the receiver provides an input[5] r; the functionality chooses a random seed s, gives s to the sender and $F(s, r)$ to the receiver.

In our protocol, the sender knows $\boldsymbol{q}_j$ and s. We can consider these values as keys to a PRF:

$$F((\boldsymbol{q}_j, s), r) = H(j \| \boldsymbol{q}_j \oplus [C(r) \cdot s])$$

Intuitively, the sender can evaluate this PRF at any point, while the receiver can

[5]More general OPRF variants allow the receiver to learn the PRF output on many inputs — here it suffices to limit the receiver to one input.

evaluate it on only one. However, we point out some subtleties:

- In our protocol, the receiver learns $\boldsymbol{t}_j = \boldsymbol{q}_j \oplus [C(r^*) \cdot s]$ for his chosen input r^*, which is more information than the "PRF output" $H(j \| \boldsymbol{t}_j)$. However, even knowing $\boldsymbol{t}_j$, the other outputs of the "PRF" still look random. This common feature of an OPRF protocol leaking slightly more than the PRF output is called *relaxed OPRF* in [FIPR05].

- In our protocol, we realize many "OPRF" instances *with related keys.* In particular, all instances have the same component s (and C).

We encapsulate these properties in the following definitions.

3.3.2.3 Our PRF variant

We refer to F as a **relaxed PRF** if there is another function $\widetilde{F}$, such that $F(k, r)$ can be efficiently computed given just $\widetilde{F}(k, r)$. We then define the relevant notion of security with respect to an adversary who can query the relaxed function $\widetilde{F}$ rather than just F.

Definition 3.3.2. Let F be a relaxed PRF with output length v, for which we can write the seed as a pair (k^*, k). Then F has **m-related-key-PRF (m-RK-PRF) security** if the advantage of any PPT adversary in the following game is negligible:

1. The adversary chooses strings $x_1, \ldots, x_n$ and m pairs $(j_1, y_1), \ldots, (j_m, y_m)$, where $y_i \neq x_{j_i}$.

2. Challenger chooses random values appropriate for PRF seeds $k^*, k_1, \ldots, k_n$ and tosses a coin $b \leftarrow \{0,1\}$.

   (a) If $b = 0$, the challenger outputs $\{\widetilde{F}((k^*, k_j), x_j)\}_j$ and $\{F((k^*, k_{j_i}), y_i)\}_i$.

   (b) If $b = 1$ the challenger chooses $z_1, \ldots, z_m \leftarrow \{0,1\}^v$ and outputs $\{\widetilde{F}((k^*, k_j), x_j)\}_j$ and $\{z_i\}_i$,

3. The adversary outputs a bit b'. The advantage of the adversary is $\Pr[b = b'] - 1/2$. ♢

Intuitively, the PRF is instantiated with n related keys (sharing the same k^* value). The adversary learns the *relaxed* output of the PRF on one chosen input for each key. Then any m additional PRF outputs (corresponding to any seed) are indistinguishable from random by the adversary.

Lemma 2. Let $\mathcal{C}$ be a $(d, \epsilon + \log_2 m)$-PRC, where $1/2^\epsilon$ is a negligible function, Let H be a d-Hamming correlation robust hash function. Define the following relaxed PRF, for $C \in \mathcal{C}$:

$$F\Big(((C,s),(\boldsymbol{q},j)),r\Big) = H(j \| \boldsymbol{q} \oplus [C(r) \cdot s])$$
$$\widetilde{F}\Big(((C,s),(\boldsymbol{q},j)),r\Big) = (j, C, \boldsymbol{q} \oplus [C(r) \cdot s])$$

Then F has m-RK-PRF security.

Proof. In the m-RK-PRF game with this PRF, we can rewrite the adversary's

view as in Section 3.3.1 as:

$$(C, \{\boldsymbol{t}_j\}_j, \{H(j_i \| \boldsymbol{t}_{j_i} \oplus [(C(x_{j_i}) \oplus C(y_i)) \cdot s])\}_i)$$

There are m terms of the form $C(x_{j_i}) \oplus C(y_i)$ for $x_{j_i} \neq y_i$. Each of these terms has Hamming weight at least d with probability at least $1 - 2^{-\epsilon - \log_2 m}$ over the choice of C. By a union bound, *all* m terms have Hamming weight at least d with probability $1 - 2^{-\epsilon}$. Conditioning on this (overwhelmingly likely) event, we can apply the d-Hamming correlation robust property of H to see that the H-outputs are indistinguishable from random. □

3.3.2.4 Our BaRK-OPRF functionality

In Figure 3.1 we formally describe the variant OPRF functionality we achieve. It generates m instances of the PRF with related keys, and allows the receiver to learn the (relaxed) output on one input per key.

The functionality is parameterized by a relaxed PRF F, a number m of instances, and two parties: a **sender** and **receiver**.

On input $(r_1, \ldots, r_m)$ from the receiver,

- Choose random components for seeds to the PRF: $k^*, k_1, \ldots, k_m$ and give these to the sender.
- Give $\widetilde{F}((k^*, k_1), r_1), \ldots, \widetilde{F}((k^*, k_m), r_m)$ to the receiver.

Figure 3.1: Batched, related-key OPRF (BaRK-OPRF) ideal functionality.

3.3.3 Main construction

We present our main construction, which is a semi-honest secure protocol for the functionality in Figure 3.1, instantiated with the relaxed PRF defined in Lemma 2.

3.3.3.1 Notation

We use the notation OT_m^k to denote k instances of 1-out-of-2 string-OT where the strings are m bits long. Let $\mathcal{S}$ denote the sender, and let $\mathcal{R}$ denote the receiver. In OT_m^k, the sender's input is $\{(x_{j,0}, x_{j,1})\}_{j\in[k]}$, i.e., m pairs of strings, each of length m, and the receiver holds input $\{r_j\}_{j\in[k]}$, where each r_j is a choice bit. Note that if $\mathcal{S}$ provides input $\{(x_{j,0}, x_{j,1})\}_{j\in[k]}$ to OT_m^k, and if $\mathcal{R}$ provides input $\{r_j\}_{j\in[k]}$ to OT_m^k, then $\mathcal{R}$ receives back $\{x_{j,r_j}\}_{j\in[k]}$, while $\mathcal{S}$ receives nothing.

We note that to simplify notation via indexing, in the following we will refer to the OT matrices as T_0 and T_1, rather than as T and U, as we did when presenting high-level overview of our work.

3.3.3.2 The BaRK-OPRF construction

Our BaRK-OPRF protocol is presented in Figure 3.2. It closely follows the high-level overview. Recall that we are considering a PRF whose seed is of the form $((C, \text{sec}), (j, \boldsymbol{q}_j))$ and whose relaxed output is of the form $\boldsymbol{t}_{0,j} = \boldsymbol{q}_j \oplus (C(r_j) \cdot \text{sec})$.

Theorem 3. The BaRK-OPRF protocol in Figure 3.2 securely realizes the functionality of Figure 3.1, instantiated with the relaxed PRF defined in Lemma 2,

INPUT OF $\mathcal{R}$: m selection strings $\boldsymbol{r} = (r_1, \ldots, r_m)$, $r_i \in \{0,1\}^*$.

PARAMETERS:

- A (κ, ϵ)-PRC family $\mathcal{C}$ with output length $k = k(\kappa)$.
- A κ-Hamming correlation-robust $H : [m] \times \{0,1\}^k \to \{0,1\}^v$.
- An ideal OT_m^k primitive.

PROTOCOL:

0. $\mathcal{S}$ chooses a random $C \leftarrow \mathcal{C}$ and sends it to $\mathcal{R}$.
1. $\mathcal{S}$ chooses $\mathrm{sec} \leftarrow \{0,1\}^k$ at random. Let s_i denote the i-th bit of sec.
2. $\mathcal{R}$ forms $m \times k$ matrices T_0, T_1 in the following way:
   - For $j \in [m]$, choose $\boldsymbol{t}_{0,j} \leftarrow \{0,1\}^k$ and set $\boldsymbol{t}_{1,j} = C(r_j) \oplus \boldsymbol{t}_{0,j}$.

   Let $\boldsymbol{t}_0^i, \boldsymbol{t}_1^i$ denote the i-th column of matrices T_0, T_1 respectively.
3. $\mathcal{S}$ and $\mathcal{R}$ interact with OT_m^k in the following way:
   - $\mathcal{S}$ acts as *receiver* with input $\{s_i\}_{i\in[k]}$.
   - $\mathcal{R}$ acts as *sender* with input $\{\boldsymbol{t}_0^i, \boldsymbol{t}_1^i\}_{i\in[k]}$.
   - $\mathcal{S}$ receives output $\{\boldsymbol{q}^i\}_{i\in[k]}$.

   $\mathcal{S}$ forms $m \times k$ matrix Q such that the i-th column of Q is the vector $\boldsymbol{q}^i$. (Note $\boldsymbol{q}^i = \boldsymbol{t}_{s_i}^i$.) Let $\boldsymbol{q}_j$ denote the j-th row of Q. Note, $\boldsymbol{q}_j = ((\boldsymbol{t}_{0,j} \oplus \boldsymbol{t}_{1,j}) \cdot \mathrm{sec}) \oplus \boldsymbol{t}_{0,j}$. Simplifying, $\boldsymbol{q}_j = \boldsymbol{t}_{0,j} \oplus (C(r_j) \cdot \mathrm{sec})$.
4. For $j \in [m]$, $\mathcal{S}$ outputs the PRF seed $((C, \mathrm{sec}), (j, \boldsymbol{q}_j))$.
5. For $j \in [m]$, $\mathcal{R}$ outputs relaxed PRF output $(C, j, \boldsymbol{t}_{0,j})$.

Figure 3.2: The BaRK-OPRF protocol

in the presence of semi-honest adversaries, where κ is the computational security parameter.

Proof. When using the abstraction of our OPRF functionality, the proof is elementary.

Simulating $\mathcal{S}$. The simulator receives output from the OPRF ideal functionality consisting of related PRF seeds: a common (C, sec) and a $\boldsymbol{q}_j$ for each $j \in [m]$. Let Q be a matrix whose rows are the $\boldsymbol{q}_j$. Let $\boldsymbol{q}^i$ denote the ith column of Q.

The simulator simulates an execution of the protocol in which $\mathcal{S}$ chooses C in step 0, chooses sec in step 1, and receives output $\{\boldsymbol{q}^i\}_{i\in[k]}$ as OT output in step 3.

Simulating $\mathcal{R}$. The simulator has input $(r_1, \ldots, r_m)$ and receives output from the OPRF ideal functionality consisting of a relaxed PRF output $(j, C, \boldsymbol{t}_j)$ for each $j \in [m]$.

The simulator simulates an execution of the protocol in which $\mathcal{R}$ receives C in step 0 and samples $\boldsymbol{t}_{0,j} = \boldsymbol{t}_j$ in step 2.

In both cases it is straightforward to check that the simulation is perfect. □

3.3.4 Improving Private Set Intersection

The main application of BaRK-OPRF is to improve the performance of semi-honest-secure **private set intersection (PSI).** Pinkas et al. [PSZ14] give a thorough summary of many different paradigms for PSI in this model.

For our purposes, we summarize only the most efficient PSI protocol, which is the OT-based paradigm of [PSZ14] including the optimizations suggested in follow up work [PSSZ15]. Hereafter we refer to their protocol as the "PSSZ" protocol.

3.3.4.1 The OPRF Implicit in PSSZ

The main building block of PSSZ, private equality test, can be viewed as a relaxed OPRF based on random OTs (i.e., oblivious transfers of random messages), which can be obtained efficiently from OT extension. The protocol is as follows, where Bob has input r, with $\ell = |r|$.

- The parties perform ℓ 1-out-of-2 OTs of random messages, with Alice as receiver. Bob acts as receiver and uses the bits of r as his choice bits. In the ith OT, Alice learns random strings $m_{i,0}$ and $m_{i,1}$, while Bob learns $m_{i,r[i]}$.

- Define the mapping $F(x) = H\left(\bigoplus_i m_{i,x[i]}\right)$, where H is a random oracle. One can then view F as a PRF whose keys are the $m_{i,b}$ values (known to Alice). Bob learns the output of F on r only. More precisely, he learns *relaxed* output $\{m_{i,r[i]}\}_i$, for which all other outputs of F are pseudorandom.

In this description, we have treated r as a string of bits, and therefore use 1-out-of-2 (random) OTs. However, when using the OT extension protocol of [KK13], the cost of a 1-out-of-2 random OT is essentially the same as a 1-out-of-256 random OT. Hence, PSSZ interpret r as strings of characters over $\{0,1\}^8$. The protocol uses one instance of 1-out-of-256 ROT for each *byte* (not bit) of r.

Regardless of whether one uses 1-out-of-2 or 1-out-of-256 OT, this OPRF protocol has cost that scales with length of the input r, whereas ours has cost independent of the input length. Our main improvement to PSSZ consists of replacing their OPRF with ours. The rest of the protocol is largely unchanged.

3.3.4.2 PSI from OPRF

We now describe how the PSSZ paradigm achieves PSI using an OPRF. This part of the overall PSI protocol is nearly identical between our implementation and that of [PSSZ15] (we include an additional small optimization). For concreteness, we describe the parameters used in PSSZ when the parties have roughly the same number n of items.

The protocol relies on **Cuckoo hashing** [PR04] with 3 hash functions, which we briefly review now. To assign n items into b bins using Cuckoo hashing, first choose random functions $h_1, h_2, h_3 : \{0,1\}^* \to [b]$ and initialize empty bins $\mathcal{B}[1, \ldots, b]$. To hash an item x, first check to see whether any of the bins $\mathcal{B}[h_1(x)]$, $\mathcal{B}[h_2(x)]$, $\mathcal{B}[h_3(x)]$ are empty. If so, then place x in one of the empty bins and terminate. Otherwise, choose a random $i \in \{1, 2, 3\}$, evict the item currently in $\mathcal{B}[h_i(x)]$, replacing it with x, and then recursively try to insert the evicted item. If this process does not terminate after a certain number of iterations, then the final evicted element is placed in a special bin called the *stash*.

PSSZ use Cuckoo hashing for PSI in the following way. First, the parties choose 3 random hash functions h_1, h_2, h_3 suitable for 3-way Cuckoo hashing. Suppose Alice has a set X of inputs and Bob has a set Y, where $|X| = |Y| = n$. Bob maps his items into $1.2n$ bins using Cuckoo hashing and a stash of size s. At this point Bob has at most one item per bin and at most s items in his stash — he pads his input with dummy items so that each bin contains exactly 1 item and the stash contains exactly s items.

The parties then run $1.2n + s$ instances of an OPRF, where Bob plays the role of receiver and uses each of his $1.2n + s$ items as OPRF input. Let $F(k_i, \cdot)$ denote the PRF evaluated in the ith OPRF instance. If Bob has mapped item y to bin i via cuckoo hashing, then Bob learns $F(k_i, y)$; if Bob has mapped y to position j in the stash, then Bob learns $F(k_{1.2n+j}, y)$.

On the other hand, Alice can compute $F(k_i, \cdot)$ for any i. So she computes sets of candidate PRF outputs:

$$H = \{F(k_{h_i(x)}, x) \mid x \in X \text{ and } i \in \{1, 2, 3\}\}$$
$$S = \{F(k_{1.2n+j}, x) \mid x \in X \text{ and } j \in \{1, \ldots, s\}\}$$

She randomly permutes elements of H and elements of S and sends them to Bob. Bob can identify the intersection of X and Y as follows. If Bob has an item y mapped to the stash, he checks whether the associated OPRF output is present in S. If Bob has an item y *not* mapped to the stash, he checks whether its associated OPRF output is in H.

Intuitively, the protocol is secure against a semi-honest Bob by the PRF property. For an item $x \in X \setminus Y$, the corresponding PRF outputs $F(k_i, y)$ are pseudorandom. It is easy to see that security holds even if Bob learns *relaxed* PRF outputs and the PRF achieves RK-PRF security, Definition 3.3.2 (i.e., Alice's PRF outputs are pseudorandom to an adversary who learns *relaxed* PRF outputs). Similarly, if the PRF outputs are pseudorandom even under related keys, then it is safe for the OPRF protocol to instantiate the PRF instances with related keys.

The protocol is correct as long as the PRF does not introduce any further collisions (i.e., $F(k_i, x) = F(k_{i'}, x')$ for $x \neq x'$). Below we discuss the parameters required to prevent such collisions.

An Optimization In the protocol summary above, Bob must search for each of his OPRF outputs, either in the set H or the set S. Furthermore, $|H| = 3n$ and $|S| = sn$. Even when using a reasonable data structure for these comparisons, they have a non-trivial effect on the protocol's running time. We now describe an optimization that reduces this cost (by approximately 10% in our implementation). The full protocol is described in Figure 3.5.

Our modification works as follows. First, Bob keeps track of a hash index $z \in \{1, 2, 3\}$ for each item $y \in Y$ that is not mapped to the stash. For example, if Bob's Cuckoo hashing maps y to bin $\#h_2(y)$, then Bob associates $z = 2$ with y. If for example y is mapped to bin by two hash functions $\#h_1(y) = \#h_2(y)$ then Bob may choose either $z = 1$ or $z = 2$ arbitrarily.

Then in the first $1.2n$ OPRF instances, Bob uses input $y\|z$. For the OPRF instances associated with the stash, he does not need to append the index z. Summarizing, if Bob has mapped item to position j in the stash, then Bob learns $F(k_{1.2n+j}, y)$. If he has not mapped y to the stash, then he learns $F(k_{h_z(x)}, y\|z)$ for *exactly one* z.

Then Alice computes the following sets:

$$H_i = \{F(k_{h_i(x)}, x\|i) \mid x \in X\}, \text{ for } i \in \{1, 2, 3\}$$
$$S_j = \{F(k_{1.2n+j}, x) \mid x \in X\}, \text{ for } j \in \{1, \ldots, s\}$$

She randomly permutes the contents of each H_i and each S_j and sends them to Bob. For each item y of Bob, if y is not mapped to the stash then Bob can whether $F(k_{h_z(y)}, y\|z) \in H_z$, for the associated hash-index z. If his Cuckoo hashing maps item y to position j in the stash, he can check whether $F(k_{1.2n+j}, y) \in S_j$.

The reason for appending the hash-index z to the PRF input is as follows. Suppose $h_1(x) = h_2(x) = i$, which is indeed can happen with noticeable probability, since the output range of h_1, h_2 is small ($[1.2n]$). Without appending z, both H_1 and H_2 would contain the identical value $F(k_i, x)$. This would leak the fact that a collision $h_1(x) = h_2(x)$ occurred. Such an event is input-dependent so cannot be simulated.[6]

With our optimization: (1) All of the calls to the PRF made by Alice (to compute the H_i's and S_j's) invoke the PRF on *distinct* key-input pairs. This ensures that the contents of these sets can be easily simulated; (2) Bob searches for each of his PRF outputs within only one set (either an H_i or an S_j) of n items. Contrast this with the approach described previously, where Bob must find each OPRF output in either a set of size $3n$ or sn (depending on whether the item is in the stash or not).

[6]The protocol and implementation of PSSZ do not account for such collisions among the Cuckoo hash functions. Duplicate values will appear in H in such an event.

Recall that the protocol is correct as long as there are no spurious collisions among PRF outputs. Since there are at most n^2 opportunities for a spurious collision (Bob searches each time for a PRF output in a set of n items), we can limit the overall probability of a spurious collision to $2^{-\sigma}$ by using PRF outputs of length $\sigma + \log_2(n^2)$.

3.3.4.3 Comparing OPRF Subprotocols

When comparing our protocol to that of PSSZ, the major difference is the choice of OPRF subprotocols. Later in Section 4.7 we give an empirical comparison of the protocols. For now, we derive an analytical comparison of the costs of the two OPRF subprotocols, to give a better sense of our improvement.

We focus on the *communication cost* associated with the OT primitives. Communication cost is an objective metric, and it often reflects the bottleneck in practice (especially in these protocols where essentially all of the cryptographic computations are precomputed). Although the computation costs of our protocols are different (e.g., ours requires computing the pseudorandom code, which is a cryptographic operation), communication cost is nonetheless a good proxy for computation costs in OT extension protocols. The data that is communicated in these protocols is a large matrix that must be transposed, and this transposition is the primary contributor to the computational cost.

The main benefit of our protocol is that its cost *does not scale with the size of the items being compared.* Each instance of OPRF consumes just one row of

Parameters: Alice has input X; Bob has input Y, with $|X| = |Y| = n$. s is an upper bound on the stash size for Cuckoo hashing.

1. Bob specifies random hash functions $h_1, h_2, h_3 : \{0,1\}^* \to [1.2n]$ and tells them to Alice.

2. Bob assigns his items Y into $1.2n$ bins using Cuckoo hashing. Let Bob keep track of $z(y)$ for each y so that: if $z(y) = \perp$ then y is in the stash; otherwise y is in bin $h_{z(y)}(y)$. Arrange the items in the stash in an arbitrary order.

   Bob selects OPRF inputs as follows: for $i \in [1.2n]$, if bin $\#i$ is empty, then set r_i to a dummy value; otherwise if y is in bin $\#i$ then set $r_i = y \| z(y)$. For $i \in [s]$, if position i in the stash is y, then set $r_i = y$; otherwise set r_i to a dummy value.

3. The parties invoke $1.2n + s$ OPRF instances, with Bob the receiver with inputs $(r_1, \ldots, r_{1.2n+s})$. Alice receives $(k_1, \ldots, k_{1.2n+s})$ and Bob receives $F(k_i, r_i)$ for all i.

4. Alice computes:

$$H_i = \{F(k_{h_i(x)}, x \| i) \mid x \in X\}, \text{ for } i \in \{1,2,3\}$$
$$S_j = \{F(k_{1.2n+j}, x) \mid x \in X\}, \text{ for } j \in \{1, \ldots, s\}$$

   and sends a permutation of each set to Bob.

5. Bob initializes an empty set $\mathcal{O}$ and does the following for $y \in Y$: If $z(y) = \perp$ and y is at position j in the stash and $F(k_{1.2n+j}, y) \in S_j$, then Bob adds y to $\mathcal{O}$. If $z(y) \neq \perp$ and $F(k_{h_{z(y)}(y)}, y \| z(y)) \in H_{z(y)}$ then Bob adds y to $\mathcal{O}$.

6. Bob sends $\mathcal{O}$ to Alice and both parties output $\mathcal{O}$.

Figure 3.3: Our optimization to the PSSZ PSI protocol, written in terms of an OPRF functionality.

the OT extension matrix. The width of this OT extension matrix is exactly the length of the pseudorandom code (PRC). In Section 3.3.5.1 we describe how to

compute an appropriate length of PRC. For the range of parameters we consider, this parameter is 424–448 bits. Hence the OT-cost of one instance of our OPRF protocol is 424–448 bits. The specific numbers are in Table 3.1.

The PSSZ OPRF protocol uses several instances of 1-out-of-256 ROT. With security parameter 128, the cost of such a random OT is 256 bits using the OT extension of [KK13].

The main optimization of [PSSZ15] allows for the OPRF subprotocols to be performed on items of length $\ell^* = \ell - \log n$ (n is the number of items in the overall PSI protocol) rather than length ℓ. Let ℓ^* denote this *effective item length.* Then $\ell^*/8$ instances of 1-out-of-256 ROT are needed for one OPRF instance. The total OT-cost of their OPRF protocol is therefore $256\ell^*/8 = 32\ell^*$ bits.

Hence, we see that our protocol has lower communication cost whenever $\ell^* > 448/32 = 14$. Among the different parameter settings reported in [PSSZ15], the only configuration with $\ell^* < 14$ is for PSI of $n \geq 2^{20}$ items of length 32 bits. For all other configurations, our PSI protocol has lower communication cost, with the savings increasing as the items become longer. See Table 3.1.

Remark on pre-hashing long PSI inputs Our improvements to PSI are most significant for PSI of long items. Yet, if the parties have items which are *very* long strings (say, thousands of bits), they can agree on a random hash function, hash their items locally, and perform PSI on the shorter hashes instead. The reader may rightfully wonder whether this idea make our improvements irrelevant!

For this approach (hash-then-PSI) to work, we must ensure that the hashing

n	ℓ	ℓ^*	OT cost PSSZ	OT cost our BaRK-OPRF	ratio
2^8	32	24	768	**424**	0.54
2^8	64	56	1792	**424**	0.24
2^8	128	120	3840	**424**	0.11
2^{12}	32	20	640	**432**	0.68
2^{12}	64	52	1664	**432**	0.26
2^{12}	128	116	3712	**432**	0.12
2^{16}	32	16	576	**440**	0.76
2^{16}	64	48	1536	**440**	0.29
2^{16}	128	112	3584	**440**	0.12
2^{20}	32	12	**384**	448	1.17
2^{20}	64	44	1408	**448**	0.32
2^{20}	128	108	3456	**448**	0.13
2^{24}	32	8	**256**	448	1.75
2^{24}	64	40	1280	**448**	0.35
2^{24}	128	104	3328	**448**	0.13

Table 3.1: Comparing the OT-cost of PSSZ-paradigm OPRF subprotocol and ours, for various parameters. The entries in the table refer to the contribution (in bits) to the size of the OT-extension matrices. ℓ is the item length (in bits), n is the total number of items in the parties' sets, and ℓ^* is the *effective* item length when using the optimizations of [PSSZ15].

introduces no collisions among the parties' items. If the parties have n items each, and we wish to limit the probability of a collision to $2^{-\sigma}$, then we must choose a hash function whose length is $\sigma + 2\log n$. When using the optimizations of [PSSZ15], the *effective* item length can be reduced from $\sigma + 2\log n$ to $\sigma + \log n$ bits.

We see that pre-hashing the items cannot reduce their *effective* length below σ bits, where σ is a statistical security parameter. Standard practice suggests $\sigma \geq 40$, and yet our protocol outperforms [PSSZ15] whenever the *effective* item

length is at least 14 bits. Hence hash-then-PSI does not allow one to bypass our improvement to [PSSZ15].

On a similar note, in our experimental results we report performance of the protocols only for PSI inputs up to 128 bits long. For statistical security parameter $\sigma = 40$, as long as the parties have at most 2^{34} (17 billion) items, they can use hash-then-PSI with a 128-bit hash.

3.3.5 Implementation & Performance

We implemented our PSI protocol and report on its performance in comparison with the state-of-the-art PSI protocol of [PSSZ15]. Our complete implementation is available on GitHub: `https://github.com/osu-crypto/BaRK-OPRF`.

In our implementation we used parameter settings consistent with PSSZ or stricter, and ran their and our code on our system so as to obtain meaningful comparisons. As do PSSZ, we use matrix transposition code from [ALSZ13] and several other optimizations.

3.3.5.1 Choosing Suitable Parameters

In this section we discuss concrete parameters used in our implementation. We use a computational security parameter of $\kappa = 128$ and a statistical security parameter of $\sigma = 40$.

The other parameters are:

n	s	k	v
2^8	12	424	56
2^{12}	6	432	64
2^{16}	4	440	72
2^{20}	3	448	80
2^{24}	2	448	88

Table 3.2: Parameters used in our implementation. n is the size of the parties' input sets; s is the maximum stash size for Cuckoo hashing; k is the width of the pseudorandom code (in bits); v is the length of OPRF output (in bits).

s: the maximum size of the stash for Cuckoo hashing, when hashing n items into $1.2n$ using 3 hash functions.

k: length of the pseudorandom code (and hence the width of the OT extension matrix) in the BaRK-OPRF protocol.

v: output length of the PRF realized by the BaRK-OPRF protocol.

A summary of our concrete parameter choices is given in Table 5.2. Below we describe how these parameters were derived.

Hashing parameters Bob uses Cuckoo hashing with 3 hash functions to assign his n items into $1.2n$ bins (and a stash). For the appropriate choice of the stash size s, we use the numbers given in [PSSZ15], which limit the probability of hashing failure to 2^{-40}.

Size of pseudorandom code Our BaRK-OPRF protocol requires a pseudorandom code achieving minimum distance $\kappa = 128$. In our protocol, Alice evaluates the PRF on $(3+s)n$ values. In order to argue that these values can be collectively

pseudorandom, so we require the underlying PRF to have m-RK-PRF security (Definition 3.3.2) for $m = (3 + s)n$.

From Lemma 2, this means we must choose a pseudorandom code with parameters $(d = \kappa, \epsilon = \sigma + \log m)$. Using Lemma 1, we calculate the minimum length of such a pseudorandom code; the results are column k in Table 5.2. We round up to the nearest multiple of 8 so that protocol messages will always be whole bytes.

Length of OPRF outputs The length of OPRF output controls the probability of a spurious collision in the PSI protocol. In Section 3.3.4.2 we argued that output length of $\sigma + \log_2(n^2)$ is sufficient to bound the probability of any spurious collision to $2^{-\sigma}$.

Using $\sigma = 40$, we compute the appropriate length in column v of Table 5.2. We round up to the nearest multiple of 8 so that protocol messages will always be whole bytes.

3.3.5.2 Environment settings

All of our experiments were implemented on a server with Intel(R) Xeon(R) CPU E5-2699 v3 2.30GHz CPU and 256 GB RAM. We run both clients on the same machine, but simulate a LAN and WAN connection using the Linux `tc` command. In the WAN setting, the average network bandwidth and the average (round-trip) latency are set to be 50 MB/s and 96 ms, respectively. In the LAN setting, the network has 0.2ms latency. All of our experiments use a single thread for each

party.

3.3.5.3 Implementation Details

In our BaRK-OPRF protocol, the offline phase is conducted to obtain an OT extension matrix of size $(1.2n+s)\times k$ by using the IKNP OT extension. Specifically, first we use the Naor-Pinkas construction [NP01] to get 128 base-OTs, which are then extended to a $k \times 128$ matrix by utilizing the pseudorandom generator. The transpose of this matrix yields the k base OTs for the BaRK-OPRF extension protocol. We extend to $1.2n + s$ OPRF instances.

We hash all inputs of both client and server at the beginning of the online phase. Following Lemma 1, we use a PRF with suitably long output as our pseudorandom code. More concretely, the parties agree on an AES-128 key sk, which is independent of their inputs, and then extend the output of AES via:

$$C(x) = AES_{sk}(1\|x)\|AES_{sk}(2\|x)\|AES_{sk}(3\|x)\|AES_{sk}(4\|x)$$

to obtain the desired k random output bits. Furthermore, to reduce the waiting time at the server side, the client will constantly send a new packet encompassing multiple code words to the server. Based on trail-and-error approach, the packet size of $2^{12} \times k$ bits is selected to minimize the waiting time. In Table 3.4, we report the running time of our protocol for both offline and online phases in different settings. For instance, in LAN environment, the online phase of our BaRK-OPRF

protocol takes about 3.2s for $n = 2^{20}$.

To illustrate the efficacy of the BaRK-OPRF-PSI approach, we compared it with a **naïve hashing** protocol and the **PSSZ** protocol. The naïve hashing protocol is a widely-used insecure protocol [PSSZ15] where both parties use the same cryptographic hash function to hash their elements, then one of the parties permutes their hash value and sends the result to the other party, who will compute the intersection by computing the match of the hash values. In the following, we conducted several performance tests with the input sets of equal size n and for inputs of length 32, 64, and 128 bits.

Note that the running time of our PSI protocol does not depend on the bit length of the input. It can be explained as follows. First, the upper bound of the length of the input is 128 bits. Second, the hash function will call a block of 128 bits to encrypt the input data, thus our protocol has the same computation cost for all bit length of the input. In addition, the communication cost of our BaRK-OPRF protocol depends only on the length of the pseudorandom code k and the length v of the OPRF outputs, which are independent of the bit length ℓ. Similarly, the naïve hashing protocol does not depend on ℓ. This was confirmed by our simulation results for different bit lengths (e.g. 32 bits, 64 bits, and 128 bits). Table 6.3 presents the running time of the naïve hashing protocol, PSSZ, and our PSI protocol in both LAN and WAN environment.

As we can see in the tables, our protocol outperforms PSSZ in almost all the case studies, especially for the long bit length of input ℓ and large values of the input size n. For example, we consider the results in the LAN setting. For the

input size of 2^{20}, our approach can improve 2.8 times and 3.6 times the performance of PSSZ for the bit lengths of 64 bits and 128 bits, respectively. For the input size of 2^{24}, the corresponding improvements are 2.3 times and 3.6 times. It is worth mentioning that it takes about 1 minute to compute the intersection for the sets of size $n = 2^{24}$. Similar observations can be inferred from Table 6.3 for the WAN setting.

At the same time, for smaller bit lengths, the PSSZ protocol can be faster than our PSI protocol. This is the case, for example, when the bit length is 32 bits and $n = 2^{24}$ in LAN setting. Since the two protocols are very similar, differing only in the choice of OPRF subprotocol, it would be relatively straightforward to implement a hybrid that always chooses the best OPRF subprotocol based on n and ℓ according to Table 3.1. However, in order to clarify the strengths/weaknesses of the two protocols, we report the performance for our approach even when it is worse.

Similar to the running time result, our communication cost is 2.9–3.3× faster than Pinkas et al. for PSI of 128-bit strings and sufficiently large sets. Concretely, for the input size of 2^{20}, our protocol can improve 3.2 times the performance of PSSZ for the bit lengths 128 bits. Table 4.5 presents the communication (in MB) of the naïve hashing protocol, PSSZ, and our BaRK-OPRF-PSI protocol.

3.4 SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension

We describe a novel approach for two-party private set intersection (PSI) with semi-honest security. Compared to existing PSI protocols, ours has a more favorable balance between communication and computation. Specifically, our protocol has the *lowest monetary cost* of any known PSI protocol, when run over the Internet using cloud-based computing services (taking into account current rates for CPU + data). On slow networks (e.g., 10Mbps) our protocol is actually the *fastest*.

Our novel underlying technique is a variant of oblivious transfer (OT) extension that we call *sparse OT extension.* Conceptually it can be thought of as a communication-efficient multipoint oblivious PRF evaluation. Our sparse OT technique relies heavily on manipulating high-degree polynomials over large finite fields (i.e. elements whose representation requires hundreds of bits). We introduce extensive algorithmic and engineering improvements for interpolation and multi-point evaluation of such polynomials, which we believe will be of independent interest.

Finally, we present an extensive empirical comparison of state-of-the-art PSI protocols in several application scenarios and along several dimensions of measurement: running time, communication, peak memory consumption, and — arguably the most relevant metric for practice — monetary cost.

3.4.1 Technical Overview of Our Results

We present a new PSI protocol paradigm that is secure against semi-honest adversaries under standard-model assumptions. We offer two variants of our protocol: one is optimized for low communication and the other for fast computation. The variant that is optimized for low communication is also secure against a malicious sender in the (non-programmable) random oracle model.

Better Balance of Computation and Communication. Compared to DH-PSI and RSA-based PSI [ACT11], both of our protocol variants have much faster running time, since ours are based on OT extension (i.e., dominated by cheap symmetric-key operations). The low-communication variant has smaller communication overhead than DH-PSI (even on a 256-bit elliptic curve) while the fast-computation variant has about the same communication cost as DH-PSI.

Compared to [KKRT16], both of our protocol variants require much less communication. Our protocols perform more computation in the form of finite field operations, making our protocols slower over high-bandwidth networks. However, the variant optimized for fast computation has a competitive running time and is the fastest over low-bandwidth networks (e.g., 30Mbps and less).

Extensive Cost Comparison. In Section 3.4.5 we perform an extensive benchmark of state-of-the-art PSI protocols for various set sizes and bandwidth configurations. To the best of our knowledge, our analysis is the first to assess PSI protocols in terms of their monetary costs. Our experiments show that in *all* settings we considered, the fast variant of our protocol has the **least monetary**

cost of all protocols — up to 40% less in some cases. A summary of the state of the art (including this work) is depicted in Figure 3.4.

Sparse OT extension technique. Our main technique, which we call *sparse OT extension*, is a novel twist on oblivious transfer (OT) extension. Roughly speaking, the idea allows the receiver to obliviously pick up a chosen subset of k out of N random secrets (where N may be exponential), with communication cost proportional only to k.

The concept is similar to an oblivious PRF [FIPR05] on which the receiver can evaluate k chosen points. Other PSI protocols like [PSSZ15, KKRT16] can also be expressed as a construction of OPRF from OT extension. However, these involve an OPRF that the receiver can evaluate *on only a single value*, resulting in significantly more effort to build PSI. This qualitative difference in OPRF flavor is the main source of our performance improvements.

New hashing techniques. It is common in PSI literature to assign items randomly to bins, and then perform a PSI within each bin. For security reasons, it is necessary to add dummy items to each bin. With existing techniques, dummy items account for 20-80% of the protocol cost! Our speed-optimized protocol variant is the first to use a kind of 2-choice hashing [SEK03] that requires almost no dummy items (e.g., 2.5%). This 2-choice hashing technique requires placing many items per bin, while previous PSI techniques are only efficient with 1 item per bin (due to their qualitatively different OPRF flavor). Hence, this hashing technique does not immediately benefit existing PSI protocols.

New polynomial interpolation techniques. Our communication-optimized protocol variant requires interpolation and multi-point evaluation of a polynomial, which turns out to be the main bottleneck for the following reasons: (1) The polynomial is over a large field of $\gg 2^{400}$ elements, since the polynomial encodes values related to an underlying OT-extension protocol. (2) The number of interpolation points depend on the parties' set size, which could be in the millions. (3) The best algorithms, which incur $O(n \log n)$ field operations, require a special set of interpolation points, namely, the x-values should be the roots of unity of the field or have a special algebraic structure. In contrast, in the context of our protocol the interpolation points (the x-values) are the parties PSI input items, which are arbitrary. The best algorithms with an arbitrary set of interpolation points incur $O(n \log^2 n)$ field operations [MB72].

We develop and demonstrate new techniques, called *Slice & Stream* and *Subproduct-Tree Reuse*, to speed up the concrete efficiency of these tasks by up to $2\times$ for the special case in which the x and y-coordinates of the points are drawn from the domains $\mathcal{D}_x$ and $\mathcal{D}_y$ where $|\mathcal{D}_x| \ll |\mathcal{D}_y|$. We believe those techniques could have a general interest (even outside of the field of cryptography).

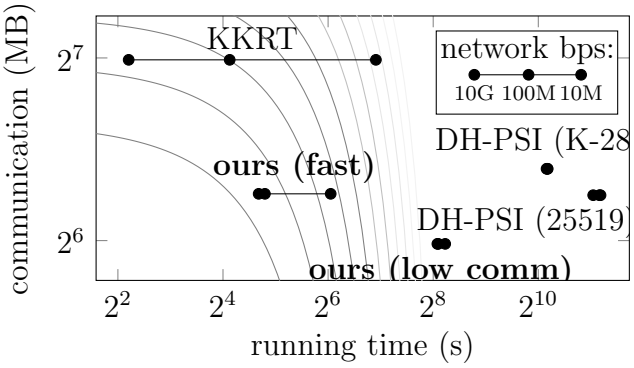


Figure 3.4: Communication and running time for different PSI protocols, with $n = 2^{20}$ items, on 3 network configurations. Curved lines are lines of equal monetary cost on a representative AWS configuration (see Section 3.4.5).

3.4.2 Main Construction

3.4.2.1 A Conceptual Overview: PSI From a Multi-Point OPRF

A conceptually simple way to realize PSI is with an **oblivious PRF** (OPRF) [FIPR05, HL08], which allows a sender Alice to learn a [pseudo]random function F, and allows the receiver Bob to learn $F(y_i)$ for each chosen item in his set $\{y_1, \ldots, y_n\}$. If Alice has items $\{x_1, \ldots, x_n\}$, she can send $F(x_1), \ldots, F(x_n)$ to Bob. If the output of F is sufficiently long, then except with negligible probability we have $F(x_i) = F(y_j)$ if and only if $x_i = y_j$. Hence, Bob can deduce the intersection of the two sets. The fact that F is pseudorandom ensures that for any item $x_i \notin \{y_1, \ldots, y_n\}$, the corresponding $F(x_i)$ looks random to Bob. Hence, no information about such items is leaked to Bob.

Sparse OT Extension: Key Idea. We can interpret IKNP OT extension as an OPRF as follows: Define the function $F(i) = \boldsymbol{m_{i,0}}$. Clearly the sender who knows the key of F can compute $F(i)$ for any i. The receiver can set his i'th choice

bit in the OT to be $r_i = 0$ if he chooses to learn $F(i)$ (in this case he learns $\boldsymbol{m_{i,0}}$), and use $r_i = 1$ if he chooses not to learn $F(i)$ (now he learns $\boldsymbol{m_{i,1}}$). To learn k OPRF outputs, the receiver includes k 0s among his choice bits. The security of OT extension implies that the receiver learns nothing about $F(i)$ whenever $r_i = 1$, and the sender learns nothing about the r_i bits.

This yields an OPRF of the form $F : [N] \to \{0,1\}^\kappa$, where N is the number of rows in the OT extension matrix. To be useful for PSI, N should be exponentially large, making this simple approach extremly inefficient. The following two key observations allow us to make the above approach efficient:

1. The parties require only *random access* to the large OT extension matrices. In the PSI application, they only read the $n \ll N$ rows indexed by their PSI inputs. While IKNP defines the matrices T, U, Q by expanding base OT values via a *PRG*, we instead expand with a PRF[7].

2. Besides the base OTs, the only communication in IKNP is when Bob sends the $N \times \kappa$ matrix P. In PSI, Bob only has knowledge of the $n \ll N$ rows of P indexed by his PSI input. Yet he must not let Alice identify the indices of these rows. Our idea is to have Bob interpolate a degree-n polynomial P where $P(y)$ is the correct "target row" of the IKNP OT extension matrix, for each y in his PSI input set. He then **sends this polynomial P instead of a huge matrix**. This change reduces Bob's communication from $O(N\kappa)$

[7] In [HS13, Sec 3.2] they also use a PRF rather than PRG, but for a completely different purpose: random access to the OT extension matrix was used to parallelize OT extension and reduce memory footprint.

to $O(n\kappa)$, allowing N to be exponential.

The polynomial P is distributed as a random polynomial (hiding Bob's inputs) since all rows of the IKNP matrix are pseudorandom from Alice's point of view. The more important concern is whether Bob learns too much. For example, suppose Bob interpolates P on points $\{y_1, \ldots, y_n\}$, but P happens to match the correct "IKNP target value" on some other $y^* \notin \{y_1, \ldots, y_n\}$ as well. This would allow Bob to learn whether Alice holds y^*, violating privacy. We argue that: (1) When the OT extension matrix is sufficiently wide, all relevant values $P(y^*)$ are sufficiently far in Hamming distance from their "target value". (2) When this is true, then Bob gets no information about Alice's items not in the intersection.

Comparison to other PSI paradigms. Other state-of-the-art PSI protocols (*e.g.*, [KKRT16, PSZ14]) can also be interpreted as constructing an OPRF from OT extension ([KKRT16] is explicitly described this way). These works construct an OPRF that the receiver can evaluate on *only one point,* and use various hashing tricks to reduce PSI to many independent instances of such an OPRF. In contrast, we construct a single instance of an OPRF where the receiver can evaluate *many points.* With such a multi-point OPRF it is trivial to achieve PSI, as illustrated above.

3.4.2.2 Protocol Details, Correctness, Performance

The formal details of our protocol are given in Figure 3.5. We use n_1 for the size of Alice's set and n_2 for the size of Bob's. We write $\mathsf{Interp}_{\mathbb{F}}(\{(x_1, y_1), \ldots, (x_d, y_d)\})$ to

denote the unique polynomial P over field $\mathbb{F}$ of degree less than d where $P(x_i) = y_i$. In IKNP, the width of the matrices (and number of base OTs) is κ whereas the width in our instantiation is $\ell > \kappa$, where ℓ is determined by the security analysis.

Costs. The main computational cost is evaluating the degree-n_2 polynomial for Alice and interpolating the polynomial for Bob. In the case of $n_1 = n_2 = n$ this can be done with $O(n \log^2 n)$ field operations (details in 3.4.4.1).

In the communication costs of the protocol, we exclude the cost of the base OTs. These are fixed and don't depend on the parties' set sizes. Bob sends $n_2\ell$ bits, while Alice sends $n_1(\lambda + \log(n_1 n_2))$ bits. Generally speaking, ℓ is much larger than $\lambda + \log(n_1 n_2)$, which suggests that the party with more items should play the role of Alice. Concrete values are discussed later in Section 3.4.5.

Correctness. The idea behind the protocol is that for every row which Bob uses to interpolate the polynomial P (namely, a row corresponding to an input of Bob), Alice sends a value which is equal to the corresponding hash value that Bob computes in the last step of the protocol.

Namely, following the discussion of IKNP, we can see that

$$Q(x) = T(x) \oplus \boldsymbol{s} \cdot \big(T(x) \oplus U(x)\big) = T(x) \oplus \boldsymbol{s} \cdot R(x)$$

and therefore in Step 5 Alice computes:

$$Q(x) \oplus \boldsymbol{s} \cdot P(x) = T(x) \oplus \boldsymbol{s} \cdot \big(P(x) \oplus R(x)\big) \tag{3.4}$$

INPUT OF SENDER ALICE: $X = \{x_1, \ldots, x_{n_1}\} \subseteq [N]$
INPUT OF RECEIVER BOB: $Y = \{y_1, \ldots, y_{n_2}\} \subseteq [N]$
PARAMETERS:

- The size $\ell := \log_2 |\mathbb{F}|$ as defined in Table 3.6.
- A κ-Hamming CRF $H : \{0,1\}^{\ell} \to \{0,1\}^{\lambda + \log(n_1 n_2)}$
- A PRF $F : \{0,1\}^{\kappa} \times [N] \to \{0,1\}$

PROTOCOL:

1. Alice chooses $\boldsymbol{s} \leftarrow \{0,1\}^{\ell}$ uniformly at random.
2. Alice and Bob invoke ℓ instances of Random OT $\mathcal{F}_{\mathsf{ROT}}^{\kappa}$. In the i-th instance:
   - Alice acts as *receiver* with input s_i.
   - Bob acts as *sender*, and receives outputs $\boldsymbol{t_i}, \boldsymbol{u_i} \in \{0,1\}^{\kappa}$.
   - Alice receives output $\boldsymbol{q_i}$.
3. For $y \in Y$, Bob computes $R(y) = T(y) \oplus U(y)$, where:
$$T(y) := F(\boldsymbol{t_1}, y) \| F(\boldsymbol{t_2}, y) \| \cdots \| F(\boldsymbol{t_\ell}, y)$$
$$U(y) := F(\boldsymbol{u_1}, y) \| F(\boldsymbol{u_2}, y) \| \cdots \| F(\boldsymbol{u_\ell}, y)$$
4. Bob computes a polynomial $P := \mathsf{Interp}_{\mathbb{F}}(\{y, R(y)\}_{y \in Y})$, and sends its coefficients to Alice
5. Alice defines Q as follows:
$$Q(x) := F(\boldsymbol{q_1}, x) \| F(\boldsymbol{q_2}, x) \| \cdots \| F(\boldsymbol{q_\ell}, x)$$
and sends $\mathcal{O} = \big\{ H\big(Q(x) \oplus \boldsymbol{s} \cdot P(x)\big) \mid x \in X \big\}$ randomly permuted to Bob
6. Bob outputs $\{y \in Y \mid H(T(y)) \in \mathcal{O}\}$

Figure 3.5: Our SpOT-PSI protocol

Now, consider the case that both parties have a common item x^*. Bob constructs P so that $P(x^*) = R(x^*)$. Alice computes $H(Q(x^*) \oplus \boldsymbol{s} \cdot P(x^*))$ which from Equation 3.4 gives Alice $H(T(x^*))$. Hence, Bob will include x^* in his output.

In case that $x \notin Y$, $P(x)$ and $R(x)$ will be different in at least κ bits with overwhelming probability (see the analysis below). Therefore, $H\big(Q(x) \oplus \boldsymbol{s} \cdot P(x)\big)$ is pseudorandom from Bob's view, under the Hamming correlation robust assumption. If σ is the output length of H, then the probability that this random value equals $H(T(y))$ for some $y \in Y$ is $n_2 2^{-\sigma}$. By a union bound over the items of $X \setminus Y$, the overall probability of Bob including an incorrect value in the output is at most $n_1 n_2 2^{-\sigma}$. Hence, choosing $\sigma = \lambda + \log_2(n_1 n_2)$ ensures that this error probability is negligible ($2^{-\lambda}$).

3.4.2.3 Properties of Polynomials

We first prove some simple lemmas about polynomials that are used in the security proof of our PSI protocol.

Hiding Bob's input. For security against a corrupt sender Alice, we simply need Bob's polynomial to hide his input:

Proposition 4. If $z_1, \ldots, z_d$ are uniformly distributed over $\mathbb{F}$, then for all distinct $x_1, \ldots, x_d$, the output of $\mathsf{Interp}_{\mathbb{F}}(\{(x_1, z_1), \ldots, (x_d, z_d)\})$ is uniformly distributed. In particular, the distribution does not depend on the x_i's.

Proof. Viewing polynomial interpolation as a linear operation, we have the follow-

ing, where $p_0, \ldots, p_{d-1}$ are the coefficients of the polynomial.

$$\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{d-1} \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{d-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \cdots & x_d^{d-1} \end{bmatrix}^{-1} \times \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{bmatrix}$$

Since the polynomial is computed as a nonsingular matrix times a uniform vector, the polynomial's distribution is also uniform. ■ □

Security for Alice. In our protocol, Bob generates a polynomial P such that $P(y) = R(y)$ for his input points $y \in Y$. The security of the protocol relies on the property that for *all other* points $x \notin Y$, $P(x)$ is far from $R(x)$ in Hamming distance (with very high probability).

Definition 3.4.1 (Bad polynomial)**.** Let $\mathsf{BadPoly}_{\mathbb{F}}^{R}(X, Y)$ be the procedure defined as follows:

1. $P := \mathsf{Interp}_{\mathbb{F}}(\{(y, R(y)) \mid y \in Y\})$
2. Output 1 iff $\exists x \in X \setminus Y$ s.t. $d_H(P(x), R(x)) < \kappa$

Proposition 5. The probability that a polynomial interpolated over points in Y also passes "too close" to another point in X is bounded by $\frac{n_1}{|\mathbb{F}|} \sum_{i<\kappa} \binom{\log_2 |\mathbb{F}|}{i}$. Formally, for all X, Y with $|X| = n_1$,

$$\Pr[\mathsf{BadPoly}_{\mathbb{F}}^{R}(X, Y) = 1] \leq \frac{n_1}{|\mathbb{F}|} \sum_{i<\kappa} \binom{\log_2 |\mathbb{F}|}{i},$$

	n_1:							
$\Pr[\mathsf{BadPoly}]$	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
2^{-40}	416	420	424	428	432	436	440	444
2^{-80}	491	495	498	502	505	509	512	515

Figure 3.6: Field size $\log_2 |\mathbb{F}|$ for our protocol, with $\kappa = 128$.

where the probability is over choice of random function $R : \mathbb{F} \to \mathbb{F}$.

Proof. For a fixed element $v \in \mathbb{F}$, the probability of a uniformly chosen element $u \leftarrow \mathbb{F}$ being closer than Hamming distance κ to v is $\sum_{i<\kappa} \binom{\log_2 |\mathbb{F}|}{i} / |\mathbb{F}|$. This is the case when entering to the second step of the procedure in Definition 3.4.1, where each $P(x)$ is already fixed and $R(x)$ is uniform in $\mathbb{F}$. The claim follows by a union bound over the (at most n_1) items in $X \setminus Y$. ■ □

On the communication complexity of the protocol. Let $\ell = \log_2 |\mathbb{F}|$. In our protocol a small ℓ leads to a bad event where two terms are close in Hamming distance. Since this bad event is a *one-time* event, it suffices to bound its probability by the statistical security parameter λ. Since the bad event involves a union bound over n, the concrete analysis involves both λ and n.

However, we could also just compute ℓ assuming the worst case $n = 2^\kappa$ (where κ is the computational security parameter), and we would get $\ell = \mathsf{poly}(\kappa)$ and a bad-event probability of $\mathsf{poly}(\mathsf{n})/2^\kappa$. For our specific protocol/analysis, $\ell = 4.3 \cdot \kappa$ appears sufficient to achieve bad event probability $n/2^\kappa$ (robust to a wide range of κ). As an analogy: in *any* OPRF-based PSI protocol, receiver learns $F(y_1), F(y_2), \ldots$ and sender sends $F(x_1), F(x_2), \ldots$. For correctness it suffices to

truncate F to $\lambda + 2log(n)$ bits, but of course it is quite enough to let F have $O(\kappa)$ bits.

In summary, asymptotically $O(n \cdot \kappa)$ bits do suffice for correctness/security, but so do $O(n \cdot \ell)$ bits, where ℓ is some function of λ, κ, n. The more fine-grained analysis of ℓ leads to less concrete communication, and that is why our concrete analysis displays a dependency of ℓ on n.

Hence, given a desired κ, n_1, and $\Pr[\mathsf{BadPoly}]$ one can solve for the smallest compatible field size. A table of such field sizes is provided in Figure 3.6.

3.4.2.4 Semi-Honest Security

Theorem 6. The protocol in Figure 3.5 securely realizes the PSI functionality of Figure 2.4 in a semi-honest setting, when F is a pseudo-random function, H is a κ-Hamming correlation robust (Definition 2.7.1), and the parameter ℓ is chosen according to the table in Figure 3.6.

Proof. Due to space limitation we only sketch here the simulators for the two cases of corrupt Alice and corrupt Bob. The full security proof including (via hybrid arguments) is defered to the the full version of our paper.

Corrupt Alice. The simulator observes Alice's inputs to the $\mathcal{F}_{\mathsf{ROT}}$ primitive and gives random $\boldsymbol{q_i}$ as OT outputs in Step 2. The only other message Alice receives is the polynomial P in Step 4. Instead of $P := \mathsf{Interp}_{\mathbb{F}}(\{y, R(y)\}_{y \in Y})$, the simulator sends a uniformly random polynomial to Alice.

Briefly, this simulation is indistinguishable for the following reasons: $R(y)$ is pseudorandom from Alice's view (by the security of the PRF which defines the conceptual OT-extension matrices). Hence, the polynomial P is distributed uniformly (from Proposition 4).

Corrupt Bob. The simulator for a corrupt Bob first obtains $X \cap Y$ from the ideal PSI functionality. It simulates random outputs $\boldsymbol{t_i}, \boldsymbol{q_i}$ from $\mathcal{F}_{\mathsf{ROT}}$. The only other message received by Bob is the set $\mathcal{O}$ in Step 5. To simulate this message, the simulator computes $n' = n_1 - |X \cap Y|$ and uniformly samples values $z_1, \ldots, z_{n'}$. It then simulates $\mathcal{O} = \{H(T(x)) \mid x \in X \cap Y\} \cup \{z_1, \ldots, z_{n'}\}$.

This simulation is indistinguishable because $P(x)$ and $R(x)$ will differ in at least κ bits for every $x \in X \setminus Y$ (Proposition 5), and as long as that is true, the corresponding outputs of H will be pseudorandom. □

3.4.2.5 Optimizations: Reducing Alice's Communication

Recall that Alice's communication consists of n_1 OPRF outputs, each of length $\lambda + \log(n_1 n_2)$. Using a trick of Tamrakar *et al.* [TLP+17], this can be reduced to roughly $\lambda + \log n_1$ bits per item. For $\lambda = 40$ and $n_1 = n_2 = 2^{20}$ this reduces communication by 25%. This improvement is even more beneficial when $n_1 \gg n_2$, since Alice's communication (despite being less *per item*) dominates the protocol overall.

For completeness, we describe the trick in the full version of our paper. It can be viewed as a public lossless compression of Alice's protocol message, and there-

fore does not affect security. We also show another approach to reduce Alice's communication to *exactly* $\lambda + \log n_1$ bits per item, inspired by polynomial encodings. However, that optimization is much slower in practice for only a marginal reduction in communication.

3.4.2.6 Security against Malicious Sender

Our protocol is secure against a malicious sender if F is modeled as a non-programmable random oracle. (In the full version of our paper, we show that our protocol is *insecure* against a malicious receiver.)

Theorem 7. The protocol in Figure 3.5 securely realizes the PSI functionality of Figure 2.4 against a malicious sender Alice, when F is modeled as a (non-programmable) random oracle.

Proof Sketch. The simulator plays the role of honest receiver Bob and the ideal $\mathcal{F}_{\mathsf{ROT}}$ functionalities in steps 1 and 2, observing Alice's $\mathcal{F}_{\mathsf{ROT}}$-input $\boldsymbol{s}$ and generating random outputs $\{\boldsymbol{q_i}\}_{i\in[\ell]}$. Throughout the protocol, the simulator also observes all of Alice's queries to the random oracle F. Without loss of generality, we can assume that whenever Alice makes a query of the form $F(\boldsymbol{q_i}, x)$ to the random oracle, where $\boldsymbol{q_i}$ is one of the $\mathcal{F}_{\mathsf{ROT}}$-outputs, it also queries $F(\boldsymbol{q_j}, x)$ for *all* $j \in [\ell]$. The simulator observes Alice's oracle queries and maintains a list

$$C = \{x \mid \text{Alice queried } F \text{ on some } F(\boldsymbol{q_i}, x)\}.$$

In step 4, the simulator sends a random polynomial P. In step 5, the simulator receives a set $\mathcal{O}$ from the corrupt Alice and computes

$$\widetilde{X} = \{x \in C \mid H(Q(x) + \boldsymbol{s} \cdot P(x)) \in \mathcal{O}\},$$

and finally sends $\widetilde{X}$ to the PSI ideal functionality.

In the full version of our paper, we use a hybrid argument to formally prove the indistinguishability of this simulator. □

3.4.3 Fast Protocol Variant

The biggest performance bottleneck in our protocol is interpolating and evaluating extremely high-degree (e.g., $d = 2^{20}$) polynomials over large (e.g., $|\mathbb{F}| > 2^{64}$) finite fields. To reduce this computational cost, we employ a technique of hashing the items into bins, and performing PSI (involving lower-degree polynomials) within each bin. This general technique is quite common in the PSI literature, and two different types of hashing have been suggested in previous work. However, we introduce a new hashing technique that (to the best of our knowledge) has not been suggested previously for PSI. As we illustrate, previous protocols are not able to immediately benefit from this new hashing technique — only our approach enjoys the advantages of this new approach.

3.4.3.1 Previous Hashing Techniques

In **simple hashing**, parties choose a random hash function $h : \{0,1\}^* \to [m]$ and assign each item x to bin with index $h(x)$. Since if Alice and Bob have the same item they both map it to the same bin, then they can perform a separate PSI within each bin. The load of each bin leaks information (i.e., it cannot be simulated just given the intersection), and therefore the parties must pad each bin up to a maximum size with dummy items. For example, with n items and $m = O(n/\log n)$ bins, the expected load of each bin is $n/m = O(\log n)$ and the maximum load B is $O(\log n)$ with high probability. In practice, B may be 4 to 5 times higher than n/m, meaning that **about 80% of the items are dummies.**

In **Cuckoo hashing** (used in [PSZ14, KKRT16]), the parties choose two hash function $h_1, h_2 : \{0,1\}^* \to [m]$. The receiver Bob places his items into m bins so that x is placed in either $h_1(x)$ or $h_2(x)$, and each bin contains at most one item. Alice places each of her items x in *both* locations $h_1(x)$ and $h_2(x)$. As above, Bob must pad each bin with dummy items to contain *exactly* one item (we can avoid dummy items for Alice). The parties perform a PSI in each bin. Cuckoo hashing leads to roughly **20% dummy items** (this is for Cuckoo hashing with three hash functions; Cuckoo hashing with two hash functions has even more dummy items), not to mention extra protocol costs associated with the stash (a special bin for items that cannot find a home in the Cuckoo hashing).

3.4.3.2 Our High-Level Approach

An important feature of Cuckoo hashing is that it results in at most one item per bin for Bob. This situation is the ideal fit for the underlying OPRF primitive of [PSZ14, KKRT16], which allows the receiver (Bob in this case) to evaluate the OPRF on *a single value.* With Cuckoo hashing, the PSI performed in each bin can be achieved with such an OPRF.

But our sparse OT extension technique results in a multi-point OPRF primitive that allows the receiver to evaluate on many values. Hence we have no need to constrain the receiver Bob to have only one item per bin. We propose to use a generalization of Cuckoo hashing called **2-choice hashing**. Similar to Cuckoo hashing, there are two hash functions h_1 and h_2, and item x can be placed in either $h_1(x)$ or $h_2(x)$. Unlike Cuckoo hashing, there is no restriction on the number of items per bin.

Cuckoo hashing is also often synonymous with an *online* hashing procedure, where all the items are processed in a single pass. For the application to PSI, though, all items are known upfront. We are free to make the best assignment of items to bins, taking into account global information about all items. [8]

These facts about 2-choice hashing indeed lead to much better performance (in terms of dummy items). The following theorem of Czumaj, Riley, and Scheideler [CRS03] shows that when the bins are allowed to contain significantly many items,

[8] This observation was concurrently and independently noted in [FNO18]; however, their focus is exclusively on Cuckoo hashing, with at most one item per bin. They do not consider our generalized 2-choice hashing.

no dummy items are needed at all!

Theorem 8 ([CRS03])**.** Let $h_1, h_2 : \{0,1\}^* \to [m]$ be two random functions. Suppose there are n items and m bins, where each item x can be placed in either $h_1(x)$ or $h_2(x)$. Let $L = \lceil n/m \rceil$. If $n = \Omega(m \log m)$ then with high probability there exists an **optimal assignment**, where each bin contains no more than L items.

The proof uses an explicit randomized algorithm to generate an optimal assignment. However, we found that the algorithm takes prohibitively long to converge. Also, its analysis of error probability is not concrete. However, if we are willing to settle for merely an "almost optimal" assignment of items to bins, the following theorem of Sanders, Egner, and Korst [SEK03] suggests that one can be found quite efficiently:

Theorem 9 ([SEK03])**.** Let n, m, h_1, h_2 be as above, with $L = \lceil n/m \rceil$. There is a deterministic algorithm running in time $O(n \log n)$ that assigns at most $L + 1$ items to each bin, with probability $1 - O(1/m)^L$ over the choice of h_1, h_2.

We propose the two-pass heuristic in Algorithm 1 for assigning items to bins. This very simple, **linear time** algorithm seems to perform well. In our experience, it never fails to find a near-optimal assignment with maximum load $L + 1 = \lceil n/m \rceil + 1$, for the parameters we use. In the rare event that it *does* fail, more iterations of the final loop are likely to succeed.

With such a near-optimal assignment, we can see that for each of the n/m bins there is only one dummy item. In practice, we set n/m to be the statistical

security parameter λ so that an assignment exists with overwhelming probability. Setting $n/m = \lambda = 40$ leads to the most dummy items one would ever consider for our protocol, but still there are **only 2.5% (= 1/40) dummy items.**

In the overall PSI protocol, Bob will send a polynomial of degree $\lceil n/m \rceil + 1$ for each bin. For each item of Alice $x \in X$, she considers both locations $h_1(x)$ and $h_2(x)$ and derives an OT-extension / OPRF output for both possibilities. She then sends these 2 outputs for each item.

Algorithm 1
FindAssignment(X, m, h_1, h_2)

1: **for** $x \in X$ **do**
2: Assign item x to bin $h_1(x)$
3: **for** $x \in X$ **do**
4: Assign item x to whichever of $h_1(x), h_2(x)$ currently has fewest items

3.4.3.3 Protocol Details

The details of the protocol are given in Figure 3.7. It mostly follows the outline given above, with one important exception. Most of the time, Alice computes two distinct mask values for each $x \in X$: one for $h_1(x)$ and one for $h_2(x)$. But $h_1(x) = h_2(x)$ is possible with probability $1/m$. In that case, depending on how one specifies this edge case, Alice will either send a repeated mask or send less masks overall. Either way, this event leaks to Bob that Alice holds such an item satisfying $h_1(x) = h_2(x)$. This issue is common to all PSI protocols that use Cuckoo hashing as well.

To address this issue, we let Bob append to each item y a bit $b \in \{1, 2\}$

INPUT OF SENDER ALICE: $X = \{x_1, \ldots, x_{n_1}\} \subseteq [N]$
INPUT OF RECEIVER BOB: $Y = \{y_1, \ldots, y_{n_2}\} \subseteq [N]$
PARAMETERS: (same as Figure 3.5, except ℓ is chosen to be compatible with $2n_1$ rather than n_1 — see text for discussion)
PROTOCOL: (steps 1–3 are the same as Figure 3.5)

4. Bob sets $m = n_2/\lambda$, chooses random functions $h_1, h_2 : [N] \to [m]$, and sends them to Alice. Then Bob assigns its items using $\mathsf{FindAssignment}(Y, m, h_1, h_2)$ (from Alg. 1) and adds dummy items so that each bin has exactly $\lceil n_2/m \rceil + 1$ items. Write $y\|b \in \mathcal{B}_i$ to mean that y was assigned to bin i by hash h_b. For each bin i, Alice computes a polynomial $P_i := \mathsf{Interp}_{\mathbb{F}}(\{y\|b, R(y\|b)\}_{y\|b \in \mathcal{B}_i})$, and sends its coefficients to Alice.

5. Alice defines Q as in Figure 3.5 and defines the sets:

$$\mathcal{O}_1 = \left\{ H\big(Q(x\|1) \oplus s \cdot P_{h_1(x)}(x\|1)\big) \;\middle|\; x \in X \right\}$$
$$\mathcal{O}_2 = \left\{ H\big(Q(x\|2) \oplus s \cdot P_{h_2(x)}(x\|2)\big) \;\middle|\; x \in X \right\}$$

   She permutes each one randomly and sends them to Bob.

6. Bob outputs $\{y \mid y\|b \in \bigcup_i \mathcal{B}_i \text{ and } H(T(y\|b)) \in \mathcal{O}_b\}$

Figure 3.7: PSI protocol using 2-choice hashing optimization.

indicating which hash function h_b was used to assign it to this bin. If $h_1(y) = h_2(y)$ we just choose b arbitrarily. Then the OT extension & polynomials are done with respect to these "extended" values. Now in the case of $h_1(y) = h_2(y)$, Bob will only learn the OT-extension output for one variant $y\|b$, but Alice (if she has such an item) will still be able to compute two distinct OT-extension outputs for the two variants.

Theorem 10. The protocol in Figure 3.7 securely realizes the PSI functionality

of Figure 2.4 in a semi-honest setting, with F, H as in Theorem 6 and ℓ according to the column indexed by $2n_1$ in Table Figure 3.6.

The semi-honest security of the modified protocol follows with a very similar proof as the original protocol, therefore we omit it for the sake of space. Unlike the original protocol, this new one is *not secure* against malicious adversaries (details are given in the full version of our paper).

3.4.3.4 Efficiency.

Theorem 9 suggests that a near-optimal assignment of items to bins exists with probability at least $1 - 2^{-n_2/m}$.

Hence, we must have $n_2/m \geq \lambda$, the statistical security parameter, to ensure that Bob's hashing step succeeds with overwhelming probability. Setting $m = n_2/\lambda$, the cost of all interpolations is now $m \cdot O(\lambda \log^2 \lambda) = O(n_2 \log^2 \lambda)$ field operations if using the asymptotically efficient algorithm, or $m \cdot O(\lambda^2) = O(n_2\lambda)$ using the simpler quadratic interpolation algorithm (which is indeed faster in practice for such small polynomials). In either case, this is a **significant** improvement over $O(n_2 \log^2 n_2)$ of the basic protocol (not to mention that distinct bins allow for easy parallelization). The cost of Alice's polynomial evaluation is similarly improved.

No matter what m we choose (assuming n_2/m is an integer), there will always be exactly m dummy items for Bob. The percentage of dummy items is m/n_2, so Alice's communication will increase by a multiplicative factor of $(1 + m/n_2)$. We

suggest $m = n_2/\lambda$, so Alice's communication increases by a $(1 + 1/\lambda)$ factor. As mentioned above, for $\lambda = 40$, this increase is only 2.5%.

Recall from Section 3.4.2.3 that the parameter ℓ (width of OT extension matrix) depends on the number of rows of the OT extension matrix that Alice accesses. With this new optimization, she accesses twice as many rows (rows $x\|1$ and $x\|2$ for every $x \in X$). This leads to a slight increase in ℓ. For the concrete parameters we consider (see Figure 3.6), ℓ must increase by only 2 bits.

3.4.4 Optimizations for High-Degree Polynomials

Despite using fast polynomial algorithms, having one party (the *interpolating party*) interpolating the huge-degree polynomial leads to a long idle time by the other party (*evaluating party*), which implies a serious computational bottleneck. In this section we show that in case that the x and y coordinates of the interpolation points are drawn from the domains $\mathcal{D}_x$ and $\mathcal{D}_y$, respectively, such that $\mathcal{D}_x \ll \mathcal{D}_y$,the idle time can be significantly shrinked. To this end, we developed new techniques, namely, *slice & stream* and *sub-product tree reuse* that allow a significant reduction of the overall time of the protocol. The former technique means that we "slice" the interpolation points into several parts, then we can interpolated each part over a smaller field and hence faster; when a slice is ready it is sent immediately to the other party for evaluation (i.e. streaming of polynomials). The latter technique is based on our observation that one sub-algorithm that constructs a sub-product tree (which is used both in interpolation and evaluation) depends

only on the x-values of the interpolation points. Since all polynomial slices use the same x-values and differ only on their y-values we can reuse the same sub-product tree for all slices! We believe our techniques are valuable for other applications that require an implementation of high-degree polynomial algorithms over large fields. As demonstrated in Section 3.4.4.2, our techniques reduce the overall interpolation and evaluation time by up to 60%.

In Section 3.4.4.1 we give an overview on known polynomial algorithms and in Section 3.4.4.2 we introduce our techniques in detail.

3.4.4.1 Background: Interpolation and Multi-Point Evaluation

Trivial implementations of polynomial interpolation and multi-point evaluation of *arbitrary* points adopt the $O(n^2)$ algorithms as they are sufficient for the typical use cases of low-degree polynomials. However, in our case the degree is in the millions, so the $O(n^2)$ algorithms are completely impractical. Faster algorithms, by Moenck and Borodin from 1972 [MB72], achieve computational complexity of $O(n \log^2 n)$. In the following we present a high level overview on the algorithms, while a detailed description is given in the full version of our paper.

Let $X = \{x_1, \ldots, x_n\} \subset \{0,1\}^\alpha$ and $Y = \{y_1, \ldots, y_n\} \subset \{0,1\}^\beta$.

- Given X and Y, the problem of *polynomial interpolation* is to find the unique $(n-1)$-degree polynomial P that passes through the points $\{(x_i, y_i)\}_{i \in [n]}$.
- Given X and an $(n-1)$-degree polynomial Q, the problem of *multi-point evaluation* is to compute $Q(X) = \{Q(x_i)\}_{i \in [n]}$.

Algorithms for both problems follow the divide-and-conquer approach such that in every iteration the problem is reduced to two half-size problems. Combining the solutions of the half-size problems to a solution of the full-size problem has a computational complexity of $O(n \log n)$. Formally, let $T(n)$ be the time to solve the interpolation and multi-point evaluation problems for $|X| = |Y| = n$, then the recurrence relation is: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n \log n) = O(n \log^2 n)$ where the second equality follows from the Master theorem [CLRS09, Ch. 4].

The evaluation and interpolation algorithms are separated to two and four sub-procedures, respectively, as follows.

Evaluation. Algorithm MultipointEvaluate(Q, X) invokes $M \leftarrow$ BuildTree(X) and outputs $Y \leftarrow$ Evaluate(Q, M).

- BuildTree(X) constructs and outputs a binary tree of polynomials, denoted M. Its leaves are the 1-degree polynomials $\{(x - a)\}_{a \in X}$ and each node is the multiplication of its two children. Thus, if the degrees of the childs are d_1 and d_2 then the node's degree is $d_1 \cdot d_2$. If n is a power of 2 then the degree of M's root is n.

- Evaluate(Q, M) evaluates the polynomial Q on X, note that X is implicitly "encoded" within M. The idea is that for every node $m \in M$ (recall that m is a polynomial), if $(x - a)$ divides m then $Q(a) = R(a)$ where $R = Q \mod m$ (i.e. it is the remainder of the division of Q by m). To obtain $Q(a)$ we replace each node m with (Parent$(m) \mod m$) and finally output the result on that

leaf. The remainder is computed in $O(n \log n)$ arithmetic operations in the underlying field.

Interpolation. Algorithm InterpolateInterpolate(X, Y) invokes $M \leftarrow$ BuildTreeBuildTree(X) as described above. Let M_0 be M's root, it computes M_0's derivative by $M_0' \leftarrow$ DerivativeDerivative(M_0) and then evaluates M_0' over X by $A \leftarrow$ MultipointEvaluateMultipointEvaluate(M_0', X). Finally it invokes $P \leftarrow$ InternalInterpolateInternalInterpolate(M, A) and outputs P. The purpose of the sub-algorithms is to enable the division of a n-size problem to two $\frac{n}{2}$-size problems. Note that within MultipointEvaluateMultipointEvaluate there is a construction of the same sub-product tree as in BuildTreeBuildTree, therefore we can skip this and construct M only once. The time of the algorithm is the sum of the times of these four sub-algorithms, $T_{\text{Interpolate}}(n) = T_{\text{BuildTree}}(n) + T_{\text{Derivative}}(n) + T_{\text{MultipointEvaluate}}(n) + T_{\text{InternalInterpolate}}(n) = O(n \log^2 n) + O(n) + O(n \log^2 n) + O(n \log^2 n) = O(n \log^2 n)$.

3.4.4.2 Polynomial Slicing and Streaming

Let $x_1, \ldots, x_n \in \{0,1\}^\alpha$ and $y_1, \ldots, y_n \in \{0,1\}^\beta$ (where $\beta > \alpha$) then we interpolate the polynomial P using points $\{(x_i, y_i)\}_{i \in [n]}$ over a field $\mathbb{F}$ where $|\mathbb{F}| = 2^\beta$. For the sake of exposition suppose that α divides β and let $\rho = \frac{\beta}{\alpha}$. For each i we define y_i^j for $j \in [\rho]$ such that $|y_i^j| = \alpha$ and $y_i = y_i^1 || \ldots || y_i^\rho$. We can "cut" P into ρ slices $P_1, \ldots, P_\rho$ such that for every x_i it holds that $P(x_i) = P_1(x_i) || \ldots || P_\rho(x_i)$. This is

done by interpolating the polynomial P_j (for $j \in [\rho]$) using the points $\{(x_i, y_i^j)\}_{i\in[n]}$. This requires a smaller field, i.e. we need that $|\mathbb{F}| = 2^\alpha$, hence P_j is produced in a shorter time.

To demonstrate the above let us fix some parameters. Assume that the parties' only task is to interpolate P using $n = 2^{20}$ points and then perform a multi-point evaluation of n points; also assume an ideal network with zero latency. Consider first performing this task directly to a "single-slice" polynomial over a field of size 2^β where $\beta = 512$. Interpolation and multi-point evaluation take $233 + 167 = 400$ seconds (detailed measurements are given in the full version.

We ignore milliseconds here and in the following). Utilizing the slicing technique with $\alpha = 128$ we have $\rho = \frac{\beta}{\alpha} = \frac{512}{128} = 4$ slices. This means that the interpolating party produces the sliced polynomials one after the other and sends them immediately (i.e. without waiting until for all polynomials to be ready) and the evaluating party evaluates them one by one upon reception. This leads to $67 \cdot 4 + 49 = 317$ seconds which is 81% of the trivial implementation.

Further utilizing the slicing technique. As shown above, the slicing and streaming technique leads to an improvement over the trivial implementation. The following observation significantly pushes forward the slicing technique: Building the polynomials tree M in the evaluation process depends only on $x_1, \ldots, x_n$, which means this can be performed *only once for all slices*. Similarly, in the interpolation algorithm the tasks of building the polynomials tree, calculating the derivative and evaluating it depends only on $x_1, \ldots, x_n$ and can be performed once

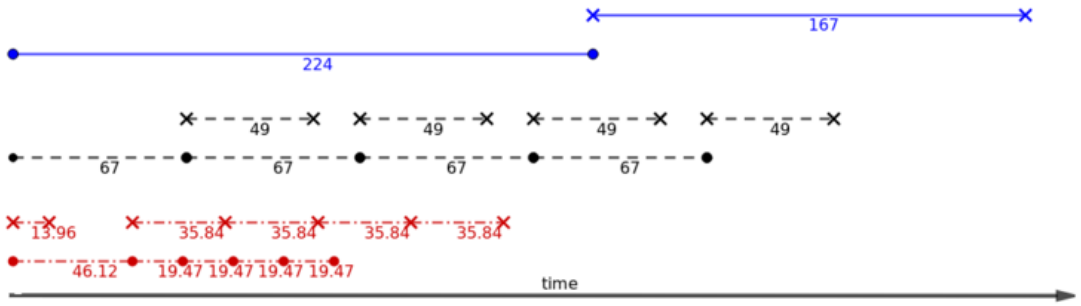


Figure 3.8: Illustrating the slicing technique. The lines between •'s represent the interpolating party and the lines between the ×'s represent the evaluating party. Solid (blue) lines illustrate the trivial implementation (overall 400 seconds), dashed (black) lines illustrate the initial slicing technique (overall 317 seconds) and dotted-dashed (red) lines illustrate the final optimization (overall 189 seconds).

and for all slices. Thus, taking $\beta = 512$, $\alpha = 128$ and $n = 2^{20}$ the one-time tasks of building the sub-product tree, calculating the derivative and evaluating it takes 12889+86+33144 = 46119 ms. The one-time task of the evaluating party (building the sub-product tree) takes 13959 ms and can surely be done simultaneously. Then the interpolating party produces 4 polynomial slices, each takes 19471 ms, and the evaluating party evaluates them upon reception. Since the evaluation task is more expensive than the interpolating task (the part being performed for each slice) the total running time is $46119 + 4 \cdot 35835 = 189459$ ms. This is less than 60% of the initial slicing technique and 48% of the trivial implementation. Both of our optimizations, together with the trivial implementation are illustrated in Figure 3.8.

Communication. Observe that this technique *does not increase* the communication complexity of the protocol. This is due to the fact that instead of sending

2^n coefficients of P, each of size β, we send 2^n coefficients of P_j, each of size α, for every j. This leads to exactly same communication size of $2^n \cdot \alpha \cdot \rho = 2^n \cdot \beta$.

3.4.5 Implementation and Performance Comparison

Recall that we have presented two variants of our protocol. In this section we will refer to them as:

spot-low: the communication-optimized variant presented in Figure 3.5, in which Bob sends one large polynomial and Alice sends one OPRF output per item.

spot-fast: the speed-optimized variant presented in Figure 3.7, in which Bob uses 2-choice hashing and Alice sends two OPRF outputs per item.

We also compare our protocols to the following:

KKRT: the leading OT-extension-based protocol from [KKRT16].

DH-PSI: Diffie-Hellman-based PSI, instantiated with either Koblitz-283 (K283) or Curve25519 (25519) elliptic curves.

Our focus in this section is on the case where $n_1 = n_2$, i.e., the parties have sets of equal size. We report some findings also for the case of unequal set sizes in the full version of our paper. Our complete implementation is available on GitHub: `https://github.com/osu-crypto/SpOT-PSI`.

3.4.5.1 Theoretical Analysis of Communication

We first compare the *theoretical* communication complexity of protocols (Table 3.6). This measures how much communication the protocols require on an idealized network where we do not care about protocol metadata, realistic encodings, byte alignment, etc. In practice, data is split up into multiples of bytes (or CPU words), and different data is encoded with headers, etc. — empirical measurements of such real-world costs are given later in Table 3.7.

For set sizes in the range 2^{16} to 2^{24}, our spot-low variant has the least communication of any of the protocols we consider: ∼15% less than DH-PSI and ∼50% less than KKRT. Our spot-fast variant uses up to ∼5% more communication than DH-PSI but 35-43% less than KKRT.

We note that KKRT uses a parameter ℓ similar to ours (corresponding to the width of the OT extension matrix), but their parameter is always slightly larger. This is because (as in our protocol) ℓ depends on how many rows of the OT matrix the sender accesses, which is more than in ours ($(3 + s)n_1$ in KKRT).

The communication optimization (described in Section 3.4.2.5) can indeed be applied to other protocols as well (DH-PSI, KKRT, and spot-fast). For example, when $n = 2^{20}$ it saves 16 bits per item (only 2.6MB in total), so the effect does not have significant impact on any comparisons. However, the optimization would be much more expensive or cumbersome to implement since it requires all OPRF outputs to be computed and sorted, but without this optimization they can be sent as they are computed.

3.4.5.2 Experimental Comparison

We now present a comparison based on implementations of all protocols.

Implementation Details. We used the implementation of KKRT provided by the authors. We implemented DH-PSI using the Miracl library implementations of Koblitz K-283 and Curve25519 elliptic curves.

For our own protocols, we implemented the polynomial interpolation and evaluation algorithms using a field of prime order p, where p is the smallest prime greater than 2^ℓ and ℓ is bit length of the output of our sparse-OT extension (the ℓ in Figure 3.6). We discuss this choice in the full version of our paper. The polynomial operations are implemented using the NTL library v10.4.0.

Note that both KKRT and our protocols require the same underlying primitives: a Hamming correlation-robust function H, a pseudorandom funtion F, and base OTs for OT extension. We instantiated these primitives exactly as KKRT: both H and F instantiated using AES, and base OTs instantiated using Naor-Pinkas [NP01]. We use the implementation of base OTs from the libOTe library[9].

All protocols use a computational security $\kappa = 128$ bits and a statistical security $\lambda = 40$ bits.

Experimental setup: AWS benchmark. We performed a series of benchmarks on the Amazon web services (AWS) EC2 cloud computing service. We use the M5.large machine class, which is classified as the current state-of-the-art

[9] https://github.com/osu-crypto/libOTe

		t					
		1	2	3	4	5	6
Virginia	1	9.6	0.17	1.08	0.063	0.068	0.084
Oregon	2			0.18	0.053	0.072	0.058
Ohio	3				0.058	0.069	0.078
Mumbai	4					0.050	0.034
Sidney	5						0.031
Sao-paolo	6						

Figure 3.9: Gbps between AWS sites.

"general purpose" instance. These machines have 2 vCPU (2.5GHz Intel Xeon) and 8 GB RAM. We considered other kinds of instances, but ultimately rejected them. The cheaper T2 class ("burstable") was found to be too unstable for our workloads, while the more expensive C5 class ("compute-optimized") resulted in more monetary cost than M5 in all cases.

Based on the geographic region of the two parties, we can realize different network speeds, as illustrated in Table 3.9. The network speeds given in the table were measured using the iperf3 command.[10] This collection of AWS sites was chosen to give a large range of bandwidth performance.

Experimental setup: local benchmark. The AWS benchmarks use a real network connection which is sometimes unpredictable. For a highly controlled experimental network, we benchmarked protocols on a single machine: Intel Xeon 2.30 GHz, 256GB RAM, 36 physical cores (note that all implementations are single-threaded unless otherwise indicated). We simulated a network connection using the Linux tc command, communicating via localhost network. We simulated a

[10] See https://iperf.fr/iperf-download.php.

LAN setting with 10 Gbps network bandwidth and 0.2ms round-trip latency, and various WAN settings with 100 Mpbs, 10 Mpbs, 1 Mpbs and 80ms round-trip latency.

AWS Pricing Scheme. Part of our motivation for evaluating protocols on AWS is to report and compare their real-world monetary costs. Hence we describe now the pricing scheme for AWS at the time of our comparison.[11] Costs are associated with both *running time* and *data transfer*, and both depend on the data center (geographic location) at which the instance runs.

The running-time cost per hour (in USD) for our instance type M5.large is 0.096 (USA), 0.101 (Mumbai), 0.12 (Sydney), 0.153 (Sao Paolo).

The data transfer cost differ depending on whether both endpoints are within AWS, and the data-center of the endpoints. We consider two network settings:

- In a **business-to-business (B2B)** setting between two fixed organizations that want to regularly perform PSI on their dynamic data, both endpoints may be within the AWS network.
- In an **internet** setting where one organization wishes to regularly perform PSI with a dynamically changing partner, only one party may be within the AWS network.

These considerations have the following effect on the cost of data transfer on AWS:

- Inbound data transfer from the Internet to EC2 is free.

[11] The pricing can be found in https://aws.amazon.com/ec2/pricing/on-demand/.

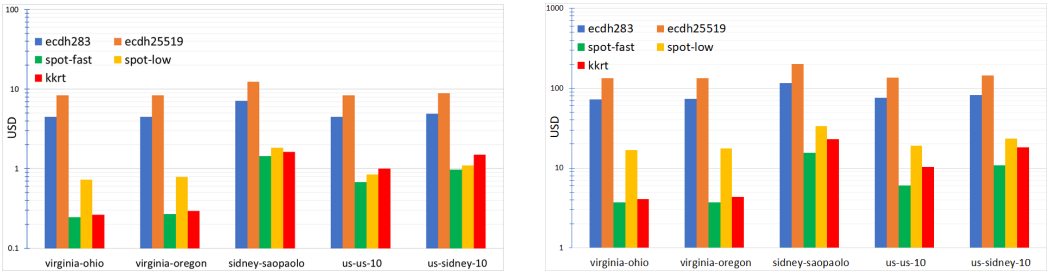


Figure 3.10: Monetary cost (in USD) per 1000 runs of PSI on 2^{16} (left) and 2^{20} (right) items, in the B2B network scenario.

- Outbound data transfer from EC2 to the Internet incurs the highest cost. Rates in USD per 1GB are 0.09 (USA), 0.1093 (Mumbai), 0.114 (Sydney), 0.25 (Sao Paolo).

- Outbound data transfer between two instance at the same site cost 0.01 USD/GB per direction.

- Outbound data transfer to another AWS site costs (in USD/GB): 0.02 (USA), 0.086 (Mumbai), 0.14 (Sydney) and 0.16 (Sao Paolo)

- Additional cost is for using a public IP address, which is indeed required for the scenarios we consider; this costs 0.01 USD/GB for all sites.

We compute the total monetary cost of a protocol execution as follows. Let T be the runtime in hours of the protocol; let X_1 and X_2 be the outbound communication of the first and second parties, resp.; let $\mathsf{C}_{\mathsf{T}1}, \mathsf{C}_{\mathsf{T}2}$ be the uptime rate of the machines run by the parties; and let $\mathsf{C}_{\mathsf{X}1}, \mathsf{C}_{\mathsf{X}2}$ be the outbound data transfer rates for the

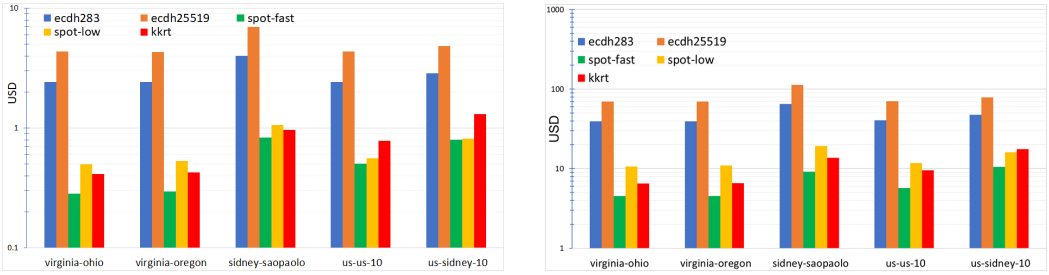
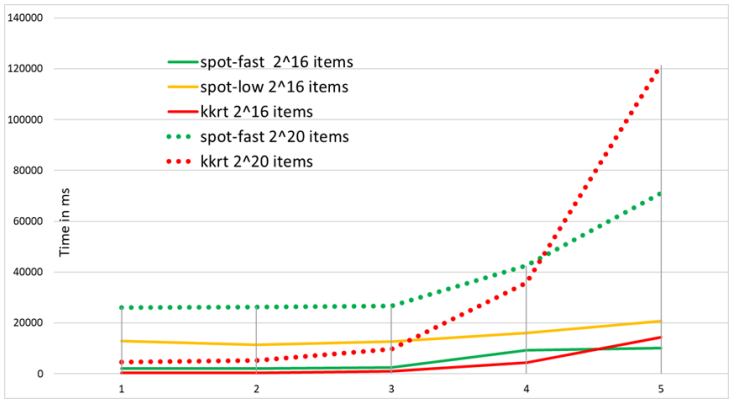


Figure 3.11: Monetary cost (in USD) per 1000 runs of PSI on 2^{16} (left) and 2^{20} (right) items, in the 'Internet' network scenario.



1	9.6 Gb/s	Virginia-Virginia
2	1.08 Gb/s	Virginia-Ohio
3	0.17 Gb/s	Virginia-Oregon
4	0.031 Gb/s	Sidney-Sao Paolo
5	0.01 Gb/s	Virginia-Virginia (controlled b/w)

Figure 3.12: Evaluated run times over AWS EC2 with descending bandwidth. Solid and dotted lines are for PSI over 2^{16} and 2^{20} items respectively. The 1-5 numbers at the x-axis of the figure represent the configurations 1-5 described in the table to the right.

machines/regions of the parties. The cost in USD is then:

$$\mathsf{TotalCost} = \mathsf{T} \cdot (\mathsf{C}_{\mathsf{T1}} + \mathsf{C}_{\mathsf{T2}}) + \mathsf{X}_1 \cdot \mathsf{C}_{\mathsf{X1}} + \mathsf{X}_2 \cdot \mathsf{C}_{\mathsf{X2}} + 0.01 \cdot (\mathsf{X}_1 + \mathsf{X}_2)$$

3.4.5.3 Experimental Results

AWS monetary cost. To limit the number of protocol executions performed on AWS, we focus on set sizes of 2^{16} and 2^{20} as they are representative of realistic set sizes for aformentioned applications of PSI.

The monetary cost of PSI protocols is presented in Figures 3.10 and 3.11. We see that our spot-fast protocol variant is the cheapest protocol in all of the settings we consider. In the B2B scenarios it is 4%-35% for PSI of 2^{16} items and 10%-40% cheaper for PSI of 2^{20} items, compared to the second cheapest protocol (KKRT). In the 'Internet' scenarios it is 13%-38% cheaper for PSI of 2^{16} items and 30%-40% cheaper for 2^{20} items. The numerical costs can be found in the full version of our paper.

Break-even point with KKRT. Our protocol has less communication than the faster KKRT protocol. As the network becomes slower, the protocol becomes more network-bound and our advantage in communication eventually leads to faster performance than KKRT. We compared the running time of the PSI protocols on networks of different speeds, in order to identify the "break-even point" where our protocol (spot-fast) becomes faster than KKRT.

From the running times in Figure 3.12, we find that the spot-fast variant overtakes KKRT as the fastest PSI protocol when network bandwidth drops below the 10–30 Mbps range. The concrete times are detailed in the full version of this paper.

Detailed, controlled local benchmarks. A more detailed benchmark for set sizes $2^{12} - 2^{24}$ and controlled network configurations is given in Table 3.7. We also considered the effect of multi-threading on protocol performance, with $T \in \{1, 4\}$ threads. The implementation of KKRT does not support multi-threading.

The communication of our protocol is approximately $2\times$ smaller than that of [KKRT16]. For example, computing the intersection of sets of size $n = 2^{20}$, spot-fast and spot-low variants require 76.43 MB and 63.18 MB respectively, whereas [KKRT16] requires 127 MB of communication, (a $1.7 - 2.0\times$ improvement).

In a single-threaded LAN setting, spot-fast variant is several times slower than KKRT, requiring 25.62 seconds with $n = 2^{20}$. Applying the same parameters to [KKRT16] results in a running time of 4.1 seconds. The running time of spot-fast variant is improved significantly by multi-threading, improving to 7.61 seconds when utilizing 4 threads.

In the WAN setting, spot-fast becomes the fastest protocol on slow (10Mbps and 1Mbps) network, due to its lower communication cost. For example, in the 10Mpbs network, for sets of size $n = 2^{20}$, spot-fast takes 66.2 seconds, while [KKRT16] requires 120.13 seconds, a $1.8\times$ improvement.

Both of our protocols outperformed DH-PSI. For example, spot-low requires 63 MB while DH-PSI (Curve25519) requires 76 MB, a $\sim 12\%$ improvement.

In terms of computation, even our slower spot-low variant is based on symmetric-key operations, and is significantly faster than DH-PSI. We also examined the effect of multi-threading. Similar to DH-PSI, spot-fast variant is extremely amenable to parallelization. Concretely, we parallelize our algorithm at the level

of bins. Both DH-PSI and spot-fast yield a similar speedup of about 3.5× by using 4 threads.

Setting	Protocol	Bit length ℓ	set size n 2^8	2^{12}	2^{16}	2^{20}	2^{24}
LAN	(insecure) naïve hashing	$\{32, 64, 128\}$	1	6	75	759	13,529
	PSSZ	32	306	380	770	4,438	**42,221**
		64	306	442	1,236	10,501	137,383
		128	307	443	1,352	13,814	213,597
	BaRK-OPRF-PSI	$\{32, 64, 128\}$	**192**	**211**	**387**	**3,780**	58,567
WAN	(insecure) naïve hashing	$\{32, 64, 128\}$	97	101	180	1,422	22,990
	PSSZ	32	609	701	1,425	8,222	**81,234**
		64	624	742	2,142	18,398	248,919
		128	624	746	2,198	23,546	381,913
	BaRK-OPRF-PSI	$\{32, 64, 128\}$	**556**	**585**	**1,259**	**7,455**	106,828

Table 3.3: Running time in ms for PSI protocols with n elements per party

Setting	Phase	set size n 2^8	2^{12}	2^{16}	2^{20}	2^{24}
LAN	Offline	171	171	216	601	7,615
	Online	21	40	171	3,179	50,952
WAN	Offline	291	313	316	758	7,482
	Online	265	272	943	6,697	99,346

Table 3.4: Running time of our BaRK-OPRF protocol in ms in offline and online phases

Protocol	Bit length ℓ	set size n 2^8	2^{12}	2^{16}	2^{20}	2^{24}	Asymptotic [bit]
naïve hashing	$\{32, 64, 128\}$	0.01	0.03	0.56	10.00	176.00	nv
PSSZ	32	0.06	0.77	9.18	142.80	1,574.40	$2\kappa(1.2n + s)\lceil \frac{\min(v', \ell) - \log(n)}{8} \rceil + (3 + s)nv'$
	64	0.09	1.37	18.78	296.40	4,032.00	
	128	0.10	1.52	23.58	411.60	6,489.60	
BaRK-OPRF-PSI	$\{32, 64, 128\}$	0.04	0.53	8.06	127.20	1,955.20	$k(1.2n + s) + (3 + s)nv$

Table 3.5: Communication in MB for PSI protocols with n elements per party. Parameters k, s, and v refer to those in Table 5.2 / Section 3.3.5.1. PSSZ requires slightly long OPRF outputs: $v' = \sigma + \log(3n^2)$. Communication costs for PSSZ and for our protocol ignore the fixed cost of base OTs for OT extension.

Protocol	**Communication**	$n = n_1 = n_2$		
		2^{16}	2^{20}	2^{24}
KKRT	$(3+s)(\lambda + \log(n_1 n_2))n_1 + 1.2\ell n_2$	$1042n$	$1018n$	$978n$
DH-PSI	$\phi n_1 + (\phi + \lambda + \log(n_1 n_2))n_2$	$584n$	$592n$	$600n$
spot-low	$1.02(\lambda + \log_2(n_2) + 2)n_1 + \ell n_2$	$488n$	$500n$	$512n$
spot-fast	$2(\lambda + \log(n_1 n_2))n_1 + \ell(1 + 1/\lambda)n_2$	$583n$	$609n$	$634n$

Table 3.6: Theoretical communication costs of PSI protocols (in bits), calculated using computational security $\kappa = 128$ and statistical security $\lambda = 40$. Ignores cost of base OTs (in our protocol and KKRT) which are independent of input size. ϕ is the size of elliptic curve group elements (256 is used here). ℓ is width of OT extension matrix (depends on n_1 and protocol).

Params.		**Protocol**	**Comm.**	**Total time (seconds)**							
n_1	n_2		(MB)	10 Gbps		100 Mbps		10 Mbps		1 Mbps	
				$T=1$	4	1	4	1	4	1	4
2^{24}	2^{24}	DH-PSI (K-283)	—	—	—	—	—	—	—	—	—
		DH-PSI (25519)	—	—	—	—	—	—	—	—	—
		KKRT	1955.2	**63.3**	—	**261.9**	—	1852.1	—	—	—
		spot-low	—	—	—	—	—	—	—	—	—
		spot-fast	1254.5	440.1	146.1	474.6	173.3	**1071.8**	**1062.8**	—	—
2^{20}	2^{20}	DH-PSI (K-283)	84.0	1141.8	338.5	1152.5	336.9	1158.2	334.2	1472.4	854.3
		DH-PSI (25519)	76.1	2110.6	632.8	2290.5	634.5	2325.7	673.0	2497.8	1014.0
		KKRT	127	**4.61**	—	**17.47**	—	120.1	—	1154.5	—
		spot-low	**63.1**	270.3	179.2	273.4	185.3	299.6	206.67	687.2	**311.16**
		spot-fast	76.4	25.6	**7.6**	27.8	**10.53**	**66.2**	**66.0**	**646.3**	645.3
2^{16}	2^{16}	DH-PSI (K-283)	5.2	69.8	20.20	70.77	21.93	71.10	22.8	80.1	44.4
		DH-PSI (25519)	4.7	136.9	39.4	140.4	40.1	142.8	40.8	151.3	48.2
		KKRT	8.06	**0.43**	—	**1.99**	—	8.4	—	74.5	—
		spot-low	**3.9**	12.8	8.8	13.7	9.8	15.1	10.9	41.1	**39.1**
		spot-fast	4.71	1.90	**0.77**	2.91	**2.02**	**5.46**	**5.36**	**40.19**	40.08
2^{12}	2^{12}	DH-PSI (K-283)	0.32	4.59	1.87	4.65	1.67	4.82	1.56	5.18	**2.75**
		DH-PSI (25519)	0.29	8.72	2.58	8.90	27.5	9.10	2.80	9.59	2.98
		KKRT	0.53	**0.22**	—	**0.87**	—	1.24	—	5.7	—
		spot-low	**0.25**	0.87	0.61	1.4	1.2	1.4	13.23	3.17	3.0
		spot-fast	0.3	0.4	**0.21**	1.14	**0.99**	**1.16**	**1.01**	**3.58**	3.51

Table 3.7: Total communication cost in MB and running time in seconds comparing our protocol to [KKRT16] and HD-PSI, with $T \in \{1, 4\}$ threads; each item has 128-bit length. 10Gbps network assumes 0.2ms RTT, and others use 80ms RTT. Cells with "—" denote setting not supported or program out of memory.

Chapter 4: Multi-party PSI

Practical Multi-party Private Set Intersection from Symmetric-Key Techniques by Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, Ni Trieu, in CCS [KMP+17].

We present a new paradigm for multi-party private set intersection (PSI) that allows n parties to compute the intersection of their datasets without revealing any additional information. We explore a variety of instantiations of this paradigm. Our protocols avoid computationally expensive public-key operations and are secure in the presence of any number of semi-honest participants (i.e., without an honest majority).

We demonstrate the practicality of our protocols with an implementation. To the best of our knowledge, this is the first implementation of a multi-party PSI protocol. For 5 parties with data-sets of 2^{20} items each, our protocol requires only 72 seconds. In an optimization achieving a slightly weaker variant of security (augmented semi-honest model), the same task requires only 22 seconds.

The technical core of our protocol is oblivious evaluation of a *programmable* pseudorandom function (OPPRF), which we instantiate in three different ways. We believe our new OPPRF abstraction and constructions may be of independent interest.

4.1 State of the Art for Multi-party PSI

A multi-party PSI protocol was first proposed by Freedman, Nissim, and Pinkas [FNP04]. The protocol of [FNP04] is based on oblivious polynomial evaluation (OPE) which is implemented using additively homomorphic encryption, such as Paillier encryption scheme. The basic idea is to represent a dataset as a polynomial whose roots are its elements, and send homomorphic encryptions of the coefficients of this protocol to obliviously evaluate it on the other party's inputs. Relying on the OPE technique, Kissner and Song [KS05] proposed a multi-party PSI protocol with quadratic computation and communication complexity in both the size of dataset and the number of parties. The computation overhead is reduced to be linear in number of participants in [SS08], which was based on bilinear groups. Furthermore, an efficient solution with quasi-linear complexity in the size of dataset is proposed in [CJS12]. In both [SS08, CJS12], the maximum number of the corrupted parties are assumed to be $n/2$. Very recent work [HV17] describes new protocols which run over a star network topology, and are secure in the standard model against either semi-honest or malicious adversaries. The basic idea is to designate one party to run a version of the protocol of [FNP04] with all other parties. The main building block in [HV17] is an additively homomorphic public-key encryption scheme, with threshold decryption, whose key is mutually generated by the parties. The protocol requires computing a linear number of encryptions and decryptions (namely, exponentiations) in the input sets. In contrast, our main building block is based on Oblivious Transfer extensions where the num-

Protocol	Communication Leader	Communication Client	Computation Leader	Computation Client	Corruption Threshold	Security Model
[KS05]	$\mathcal{O}(tnm\log(\lvert X\rvert))\lambda$		$\mathcal{O}(ntm^2)$		$n-1$	semi-honest
[CJS12]	$\mathcal{O}((n^2m+nm)\lambda)$		$\mathcal{O}(nm+m)$		$\lfloor n/2\rfloor$	semi-honest
[HV17]	$\mathcal{O}(nm\lambda)$	$\mathcal{O}(m\lambda)$	$\mathcal{O}(mn\log_2(m))$	$\mathcal{O}(m)$	$n-1$	semi-honest
Ours	$\mathcal{O}(nm\lambda)$	$\mathcal{O}(m\lambda)$	$\mathcal{O}(n\kappa)$	$\mathcal{O}(\kappa)$	$n-1$	augmented semi-honest
		$\mathcal{O}(mt\lambda)$		$\mathcal{O}(t\kappa)$		semi-honest

Table 4.1: Communication (bits) and computation (number of exponentiations) complexities of multi-party PSI protocols in the semi-honest setting, where n is number of parties, t dishonestly colluding, each with set size m; X is the domain of the element; and λ and κ are the statistical and computational security parameters, respectively. In our protocols, the computational complexities are in an offline preprocessing phase.

ber of exponentiations does not depend on the size of the dataset. [HV17] does not include implementation, but we expect that our protocols are much faster due to building from symmetric primitives. We describe the performance of representative multi-party PSI protocols in the semi-honest settings in Table 6.1.

We mention that multi-party PSI was also investigated in the server-aided model, based on the existence of a server which does not collude with clients [MN15, ATD15]. Information-theoretic PSI protocols, possible in the multi-party setting, are considered in [LW07, PCR08, BA12].

4.2 Our Contributions

We design a modular approach for multi-party PSI that is secure against an arbitrary number of colluding semi-honest parties. Our approach can be instantiated in a number of ways providing trade-offs for security guarantees and computation

and communication costs.

We implemented several instantiations of our PSI approach. To our knowledge, this is the first implementation of multi-party PSI. We find that multi-party PSI is practical, for sets with a million items held by around 15 parties, and even for larger instances. The main reason for our protocol's high performance is its reliance on fast symmetric-key primitives. This is in contrast with prior multi-party PSI protocols, which require expensive public-key operations for each item. Our implementation will be made available on GitHub.

Our PSI Approach. The main building block of our protocol, which we believe to be of independent interest, is *oblivious, programmable PRF (OPPRF).* Recall, oblivious PRF (OPRF) is a 2-party protocol in which the sender learns a PRF key k and the receiver learns $F(k, r)$, where F is a PRF and r is the receiver's input. In an OPPRF, the PRF F further allows the sender to "program" the output of F on a limited number of inputs. The receiver learns the PRF output as before, but, importantly, does not learn whether his input was one on which the PRF was programmed by the sender. We propose three OPPRF constructions, with different tradeoffs in communication, computation, and the number of points that can be programmed.

Basic idea. Our PSI protocol consists of two major phases. First, in the **conditional zero-sharing phase**, the parties collectively and securely generate additive sharings of zero, as follows. Each party P_i obtains, for each of its items x_j, a share of zero, denoted s_j^i. It holds that $\sum_{i=1}^{n} s_j^i = 0$. Namely, if all parties have

x_j in their sets then the sum of their obtained shares is zero (else, w.h.p., the sum is non-zero). In the second phase, parties perform **conditional reconstruction** of their shares. The idea is for each P_i to program an instance of OPPRF to output its share s_j^i when evaluated on input x_j. Intuitively, if all parties evaluate the corresponding OPPRFs on the same value x_j, then the sum of the OPPRF outputs is zero. This signals that x_j is in the intersection. Otherwise, the shares sum to a random value.

This brief overview ignores many important concerns — in particular, how the parties coordinate shares and items without revealing the identity of the items. We propose several ways to realize each of the two PSI phases, resulting in a suite of many possible instantiations. We then discuss the strengths and weaknesses of different instantiations.

A more detailed overview of the approach and the two phases is presented in Section 4.5, prior to the presentation of the full protocol.

4.3 Programmable OPRF

Our PSI approach builds heavily on the concept of oblivious PRFs (OPRF). We review the concepts here and also introduce our novel *programmable* variant of an OPRF.

PARAMETERS: A programmable PRF F, and upper bound n on the number of points to be programmed, and bound t on the number of queries.

BEHAVIOR: Wait for input $\P$ from the sender $\mathcal{S}$ and input $(q_1, \ldots, q_t)$ from the receiver $\mathcal{R}$, where $\P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ is a set of points. Run $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\P)$ and give (k, hint) to the sender. Give $(\mathsf{hint}, F(k, \mathsf{hint}, q_1), \ldots, F(k, \mathsf{hint}, q_t))$ to the receiver.

Figure 4.1: The OPPRF ideal functionality $\mathcal{F}_{\mathsf{opprf}}^{F,t,n}$

4.3.1 Definitions

Oblivious PRF. An **oblivious PRF (OPRF)** [FIPR05] is a 2-party protocol in which the sender learns a PRF key k and the receiver learns $F(k, q_1), \ldots, F(k, q_t)$, where F is a PRF and $(q_1, \ldots, q_t)$ are inputs chosen by the receiver. Note that we are considering a variant of OPRF where the receiver can obtain several PRF outputs on statically chosen inputs. We describe the ideal functionality for an OPRF in Figure 2.2.

Instantiation and Security Details. While many OPRF protocols exist, we focus on the protocol of Kolesnikov et al. [KKRT16]. This protocol has the advantage of being based on oblivious-transfer (OT) extension. As a result, it uses only inexpensive symmetric-key cryptographic operations (apart from a constant number of initial public-key operations for base OTs). The protocol efficiently generates a large number of OPRF instances, which makes it a particularly good fit for our eventual PSI application that uses many OPRF instances. Concretely, the amortized cost of each OPRF instance costs roughly 500 bits in communication and a few symmetric-key operations.

Technically speaking, the protocol of [KKRT16] achieves a slightly weaker variant of OPRF than what we have defined in Figure 2.2. In particular, (1) PRF instances are are generated with *related keys*, and (2) the protocol reveals slightly more than just the PRF output $F(k, q)$. We stress that in the resulting PRF of [KKRT16] the construction remains secure even under these restrictions. More formally, let $leak(k, q)$ denote the extra information that the protocol leaks to the receiver. [KKRT16] gives a security definition for PRF that captures the fact that outputs of F, under related keys $k_1, \ldots, k_n$, are pseudorandom even given $leak(k_i, q_i)$. Our OPPRF constructions are built on this OPRF, and as a result our constructions would inherit analogous properties as well.

For ease of presentation and reasoning, we work with the cleaner security definitions that capture the main spirit of programmable OPRF. We emphasize that, although cumbersome, it is possible to incorporate all of the [KKRT16] relaxations into the definitions. We stress that our eventual application of PSI is secure *in the standard sense* when built from such relaxed OP[P]RF building blocks.

Programmable PRF. We introduce a new notion of a programmable oblivious PRF. Intuitively, the functionality is similar to OPRF, with the additional feature that it allows the sender to program the output of the PRF on a set of points chosen by the sender. Before presenting the definition of this functionality, we discuss a PRF that supports being programmed in this way.

A **programmable PRF** consists of the following algorithms:

- $\mathsf{KeyGen}(1^\kappa, ¶) \rightarrow (k, \mathsf{hint})$: Given a security parameter and set of points

$\P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with distinct x_i-values, generates a PRF key k and (public) auxiliary information hint. We often omit the security parameter argument when it is clear from context.

- $F(k, \mathsf{hint}, x) \to y$: Evaluates the PRF on input x, giving output y. We let r denote the length of y.

A programmable PRF satisfies **correctness** if $(x, y) \in \P$, and $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\P)$, then $F(k, \mathsf{hint}, x) = y$. For the security guarantee, we consider the following experiment/game:

$\underline{\mathsf{Exp}^{\mathcal{A}}(X, Q, \kappa)}$:
for each $x_i \in X$, chose random $y_i \leftarrow \{0,1\}^r$
$(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(1^\kappa, \{(x_i, y_i) \mid x_i \in X\})$
return $\mathcal{A}\Big(\mathsf{hint}, \{F(k, \mathsf{hint}, q) \mid q \in Q\}\Big)$

We say that a programmable PRF is (n, t)**-secure** if for all $|X_1| = |X_2| = n$, all $|Q| = t$, and all polynomial-time $\mathcal{A}$:

$$\Big|\Pr[\mathsf{Exp}^{\mathcal{A}}(X_1, Q, \kappa) \Rightarrow 1] - \Pr[\mathsf{Exp}^{\mathcal{A}}(X_2, Q, \kappa) \Rightarrow 1]\Big| \quad \text{is negligible in } \kappa$$

Intuitively, it is hard to tell what the set of programmed points was, given the hint and t outputs of the PRF, if the points were programmed to random outputs. Note that this definition implies that unprogrammed PRF outputs (i.e., those not set by the input to KeyGen) are pseudorandom.

The reason for including a separate "hint" as part of the syntax is that our protocol constructions will naturally leak this hint to the receiver (in addition to the receiver's PRF output). We propose a definition that explicitly models this leakage and ensures that it is safe.

Oblivious Programmable PRF (OPPRF). The formal definition of an oblivious programmable PRF (OPPRF) functionality is given in Figure 4.1. It is similar to the plain OPRF functionality except that (1) it allows the sender to initially provide a set of points ¶ which will be programmed into the PRF; (2) it additionally gives the "hint" value to the receiver.

We now give several constructions of an OPPRF, with different tradeoffs in parameters.

4.3.2 A Construction Based on Polynomials

Our polynomial-based construction is presented in Figure 4.2. We first describe the underlying programmable PRF. Let F be a PRF and define our new programmable PRF $\widehat{F}$ as follows:

- $\mathsf{KeyGen}(¶ = \{(x_1, y_1), \ldots, (x_n, y_n)\})$: Choose a random key k for F. Interpolate a degree $n-1$ polynomial p over the points $\{(x_1, y_1 \oplus F(k, x_1)), \ldots, (x_n, y_n \oplus F(k, x_n))\}$. Let hint be the coefficients of p.
- $\widehat{F}(k, \mathsf{hint}, q) = F(k, q) \oplus p(q)$.

It is not hard to see that $\widehat{F}$ satisfies correctness since for $x_i \in ¶$ it holds that

$\widehat{F}(k, \mathsf{hint}, x_i) = F(k, x_i) \oplus p(x_i) = F(k, x_i) \oplus y_i \oplus F(k, x_i)$. Security follows from the fact that when the y_i values are distributed uniformly, so is the hint p. This is true regardless of the number of queries the receiver makes.

Finally, the OPPRF protocol for $\widehat{F}$ is straightforward if there is an OPRF protocol for F: the parties simply invoke $\mathcal{F}_{\mathsf{oprf}}^{F,t}$ on their inputs. The sender obtains k and uses it to generate the hint as above, and sends it to the receiver. The receiver, obtaining $F(k, q_i)$ from $\mathcal{F}_{\mathsf{oprf}}^{F,t}$, can compute its output $\widehat{F}(k, \mathsf{hint}, q_i) = F(k, q_i) \oplus p(q_i)$. The description of the OPPRF protocol is given in Figure 4.2. Simulation is trivial, as the parties' views in the protocol are exactly the OPPRF output.

Costs. The main advantage of this construction is that the only message that needs to be sent in addition to the $\mathcal{F}_{\mathsf{oprf}}$ protocol is the polynomial p whose length is exactly that of n values. This seems the minimal communication overhead that is needed to achieve OPPRF from OPRF. On the other hand, the interpolation of the polynomial takes time $\mathcal{O}(n^2)$ which can be expensive for large n.

4.3.3 A Construction Based on Bloom Filters

Garbled Bloom filters (GBF) were introduced in [DCW13] in the context of PSI protocols. A GBF is an array $GBF[1 \dots N]$ of strings, associated with a collection of hash functions $h_1, \dots, h_k : \{0,1\}^* \to [N]$. The GBF implements a key-value

INPUT OF $\mathcal{S}$: n points ¶ $= \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$; and $y_i \in \{0,1\}^r$

INPUT OF $\mathcal{R}$: $Q = (q_1, \ldots, q_t) \in (\{0,1\}^*)^t$.

PROTOCOL:

1. $\mathcal{R}$ sends Q to $\mathcal{F}_{\mathsf{oprf}}^{F,t}$. The sender receives k and receiver receives $F(k, q)$ for $q \in Q$.
2. $\mathcal{S}$ interpolates the unique polynomial p of degree $n-1$ over the points $\{(x_1, y_1 \oplus F(k, x_1)), \ldots, (x_n, y_n \oplus F(k, x_n))\}$.
3. $\mathcal{S}$ sends the coefficients of p to $\mathcal{R}$.
4. $\mathcal{R}$ outputs $(p, F(k, q_1) \oplus p(q_1), \ldots, F(k, q_t) \oplus p(q_t))$.

Figure 4.2: Polynomial-based OPPRF protocol

store, where the value associated with key x is:

$$\bigoplus_{j=1}^{k} GBF[h_j(x)]. \qquad (\star)$$

A GBF can be programmed to map specific keys to chosen values:

1. Initialize array GBF with all entries equal to $\perp$
2. For each key-value pair (x, v), let $J = \{h_j(x) \mid GBF[h_j(x)] = \perp\}$ be the relevant positions of GBF that have not yet been set. Abort if $J = \emptyset$. Otherwise, choose random values for $GBF[J]$ subject to the lookup equation ($\star$) equaling the desired value v.
3. For any remaining $GBF[j] = \perp$, replace $GBF[j]$ with a randomly chosen value.

It is clear that, unless this procedure aborts, it produces a GBF with the desired key-value mapping. In [DCW13] it was observed that the procedure aborts when processing item x if and only if x is a false positive for a *plain* Bloom filter containing the previous items (think of the plain Bloom filter obtained by interpreting a $\perp$ in GBF as 0 and anything else as 1). The false-positive probability for a plain Bloom filter has been well analyzed. In particular, to bound the probability by $2^{-\lambda}$, one can use a table with $N = n\lambda \log_2 e$ entries to store n items. In that case, the optimal number of hash functions is λ. If we set $\lambda = 40$, we get that the table size is about $60n$ and the number of hash functions is $k = 40$. In addition, by doing less hashing[KM08], each insert only requires two hash functions $h_1(x)$ and $h_2(x)$. The additional $k - 2$ hash functions $h_i(x), i \in [3, k]$, is simulated by $h_i(x) = h_1(x) + i \times h_2(x)$.

Given the GBF construction, an OPPRF construction is relatively straightforward and similar to the polynomial-based construction. Instead of the mappings $x_i \mapsto y_i \oplus F(k, x_i)$ being stored in a polynomial, they are stored in a GBF. The construction is defined in Figure 4.3. Security holds naturally, since if the y_i points are chosen randomly, all positions in the GBF are uniformly distributed.

Costs. The advantage of the Bloom filter based construction, compared to the polynomial-based construction, is that the insertion algorithm runs in time $\mathcal{O}(n)$ rather $\mathcal{O}(n^2)$, and is also very efficient in practice. The communication is still $\mathcal{O}(n)$ but the constant coefficient is high (the actual communication is $60n$ items rather than n) and therefore communication might be a bottleneck, especially on

INPUT OF $\mathcal{S}$: n points ¶ $= \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$ and $y_i \in \{0,1\}^r$

INPUT OF $\mathcal{R}$: $Q = (q_1, \ldots, q_t) \in (\{0,1\}^*)^t$.

PROTOCOL:

1. $\mathcal{R}$ sends Q to $\mathcal{F}_{\mathsf{oprf}}^{F,t}$. The sender receives k and receiver receives $F(k, q)$ for $q \in Q$.
2. $\mathcal{S}$ inserts the n pairs

$$\{(x_1, y_1 \oplus F(k, x_1)), \ldots, (x_n, y_n \oplus F(k, x_n))\}$$

   into a garbled Bloom filter denoted as G, with entries which are each r bits long. It fills the remaining empty entries with random values.
3. $\mathcal{S}$ sends G to $\mathcal{R}$ as well as the k hash functions (the functions need not be sent explicitly, and can be defined by setting some context dependent prefixes to inputs of a known hash function).
4. For $i = 1$ to t, $\mathcal{R}$ computes $z_i = F(k, q_i) \oplus \bigoplus_{j=1}^{k} G[h_j(q_i)]$. Finally $\mathcal{R}$ outputs $(G, z_1, \ldots, z_t)$.

Figure 4.3: Bloom-filter-based OPPRF protocol

slow networks.

4.3.4 Table-Based Construction

The previous OPPRF constructions can be instantiated with any underlying OPRF that allows the receiver to evaluate the PRF on any number t of points. The resulting OPPRF constructions will inherit the same t. Meanwhile, our most efficient OPRF building block from [KKRT16] only supports $t = 1$. In this section we describe a construction tailored for the case of $t = 1$, and for small values of n

(the number of programmed points).

The main idea behind this construction is as follows. For each pair (x_i, y_i) the sender $\mathcal{S}$ uses $F(k, x_i)$ as an encryption key to encrypt the corresponding value y_i. Let T be the collection of these encryptions; then T comprises the OPPRF hint. At a high level, the receiver can obtain $F(k, q)$ and use it as a key to decrypt the appropriate ciphertext from T.

The main challenges are: (1) $\mathcal{R}$ should not know whether he is getting random or programmed output values (i.e. whether $x = x_i$ for some i), and (2) $\mathcal{R}$ must learn which ciphertext from T to decrypt. We achieve both properties by using $F(k, q)$ to derive a *pointer* into the table T. In order to achieve property (1), $\mathcal{R}$ must *always* decrypt some ciphertext of T, even if $x \neq x_i$.

Concretely, suppose n is 20, so that $\mathcal{S}$ needs to program only 20 points. $\mathcal{S}$ will make a table T of size $2^5 = 32$ (next power of 2 greater than 20). $\mathcal{S}$ will choose a random nonce $v \in \{0,1\}^\kappa$ until $\{H(F(k, x_i)\|v) \mid i \leq 20\}$ are all distinct, where $H : \{0,1\}^* \mapsto \{0,1\}^5$ is a hash function modeled as a random oracle. For each $i \in [n]$, $\mathcal{S}$ computes $h_i = H(F(k, x_i)\|v)$, and sets $T_{h_i} = F(k, x_i) \oplus y_i$. The remaining entries of T ($32 - 20 = 12$ of them in this case) are chosen uniformly. $\mathcal{S}$ sends this nonce v together with the table T to the the receiver as part of the hint.

From the receiver's point of view, on input x he will use $F(k, q)$ to decrypt the ciphertext in position $H(F(k, q)\|v)$ of the table. The distinctness of the $H(F(k, x_i)\|v)$ values allows the sender to place encryptions of the y_i values at appropriate positions in T without any conflicts. The details are given in Fig-

INPUT OF $\mathcal{S}$: n points ¶ $= \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$; and $y_i \in \{0,1\}^r$

INPUT OF $\mathcal{R}$: $q \in \{0,1\}^*$.

PARAMETERS: random oracle $H : \{0,1\}^* \to \{0,1\}^m$, where $m = 2^{\lceil \log(n+1) \rceil}$.

PROTOCOL:

1. $\mathcal{R}$ sends q to $\mathcal{F}_{\mathsf{oprf}}^{F,t}$. The sender receives k and receiver receives $F(k, q)$.
2. $\mathcal{S}$ samples $v \leftarrow \{0,1\}^\kappa$ until $\{H(F(k, x_i)\|v) \mid i \in [n]\}$ are all distinct.
3. For $i \in [n]$, $\mathcal{S}$ computes $h_i = H(F(k, x_i)\|v)$, and sets $T_{h_i} = F(k, x_i) \oplus y_i$.
4. For $j \in \{0,1\}^m \setminus \{h_i \mid i \in [n]\}$, $\mathcal{S}$ sets $T_j \leftarrow \{0,1\}^r$.
5. $\mathcal{S}$ sends T and v to $\mathcal{R}$.
6. $\mathcal{R}$ computes $h = H(F(k, q)\|v)$, and outputs $(T, v, T_h \oplus F(k, q))$.

Figure 4.4: Basic table-based OPPRF protocol.

ure 4.4. Note that the OPPRF protocol is restricted to the case of $t = 1$. Because of that, it suffices to use one-time pad encryption for the table entries.

Security & parameters. The underlying programmable PRF satisfies security based on two observations: The easy observation is that table T itself is uniformly distributed when the y_i values are uniformly distributed (as in the security definition for programmable PRF).

Next, we must argue that the nonce v leaks no information about the set of programmed points. Fix a candidate v and define $z_i = H(F(k, x_i)\|v)$. The sender tests this candidate v by seeing whether there is a collision among $\{z_i\}$ values. The receiver sees at most one value of the form $F(k, x_i)$. So by the PRF security of F,

at least $n-1$ of the other F outputs are distributed randomly from the receiver's perspective. Since H is a random oracle, it follows that at least $n-1$ of the z_i values are distributed independent of the receiver's view (even when the receiver has oracle access to H). Finally, the condition of a collision among randomly chosen $\{z_i\}$ values is independent of any *single* z_i. Hence, the probability of a candidate v being chosen (and thus the overall distribution of v) is the same regardless of whether the receiver queried F on one of the sender's programming points.

It is important to discuss the parameter choice m (length of H output), as it greatly affects performance (the number of retries in step 2 of the protocol). We can calculate the probability that for a random v, the $\{H(s_i \| v) \mid i \in [n]\}$ values are distinct:

$$\mathsf{Pr}_{\mathsf{unique}} = \prod_{i=1}^{n-1} \left(1 - \tfrac{i}{2^m}\right) \tag{4.1}$$

The expected number of restarts when sampling v is $1/\mathsf{Pr}_{\mathsf{unique}}$.

Looking ahead to our PSI protocol, the OPPRF will be programmed with n items, where n is the number of items hashed into a particular bin. Different bins will have a different number of items. We must set m in terms of the *worst case* number of items per bin, so that no bin exceeds 2^m items with high probability. However, *on average,* a bin will have very few items.

Concretely, for PSI of 2^{20} items we choose hashing parameters so that no bin exceeds 30 items with high probability. Hence we set $m = 5$ (so T has 32 entries). Yet, the *expected* number of items in a bin is roughly 3. For the vast majority of bins, the sender programs the OPPRF on at most 7 points. In such a bin, only 2

trials are expected before finding a suitable v.

Costs. This OPPRF construction has favorable communication and computational cost. It requires communicating a single nonce v along with a table whose length is that of $\mathcal{O}(n)$ items. The constant in the big-O is at most 2 (the number of items is rounded up to the nearest power of 2). The computational cost of the protocol is to evaluate a random oracle H, $n\tau$ times, where τ is the number of restarts in choosing v. While these computational costs can be large in the worst case, the typical value of τ in our PSI protocol is a small constant when averaged over all of the instances of OPPRF. Our experiments confirm that this table-based OPPRF construction is indeed fast in practice.

4.4 Extending OPPRF to Many Queries

The OPPRF constructions in the previous section are efficient when n (the number of programmed points) is small. When built from the efficient OPRF protocol of [KKRT16], they allow the receiver to evaluate the programmable PRF on only $t = 1$ point. We now show how to use a hashing technique to overcome both of these limitations. We show how to extend OPPRF constructions described in the previous section to support both a large n and a large t.

At the high level, the idea is that each party hashes their items into bins. Each bin contains a small number of inputs which allows the two parties to evaluate OPPRF bin-by-bin efficiently. The particular hashing approach we have in mind

is as follows. Suppose the receiver has items $(q_1, \ldots, q_t)$ on which he wants to evaluate an OPPRF. The sender has a set $¶ = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of points to program.

Cuckoo hashing. The receiver uses Cuckoo hashing (Section 2.5) to hash his items into bins. We will use a variant of Cuckoo hashing with k hash functions $h_1, \ldots, h_k$, and m bins denoted as $B[1 \cdots m]$. Each item q is placed in exactly one of $\{B[h_1(q)], \ldots, B[h_k(q)]\}$. Based on t and k, the parameter m is chosen so that every bin can contain at most one item with probability $1 - 2^{-\lambda}$ for a security parameter λ. We note that previous applications of Cuckoo hashing to PSI [PSSZ15] have used a variant of Cuckoo hashing that involves an additional stash (a place to put items when insertion fails). However, a stash renders our scheme much less efficient (every item in one party's stash must be compared to every item of another party). Instead, we propose a variant of Cuckoo hashing that avoids a stash by using 3 "primary" Cuckoo hash functions, and then falling back to 2 "supplementary" Cuckoo hash functions when the first 3 fail. We empirically determine the parameters used in our hashing scheme to ensure that the hashing succeeds except with the probability less than $2^{-\lambda}$. The details are in the full version of our paper.

Simple hashing. Using the same set of hash functions, the sender then maps his points $\{x_1, \ldots, x_n\}$ into bins, with each item being mapped under *all* of the Cuckoo hash functions (i.e., each of the sender's items appears k times in the hash table). Using standard balls-and-bins calculations based on n, k, and m, one can

Probability	Bin scale & Max Bin Size	set size n				
		2^{12}	2^{14}	2^{16}	2^{20}	2^{24}
2^{-30}	ζ_1	1.15	1.13	1.13	1.13	1.12
	ζ_2	0.14	0.14	0.14	0.15	0.16
	β_1	28	28	29	30	31
	β_2	63	63	63	63	63
2^{-40}	ζ_1	1.17	1.15	1.14	1.13	1.12
	ζ_2	0.15	0.16	0.16	0.17	0.17
	β_1	27	28	29	30	31
	β_2	63	63	63	63	63

Table 4.2: Required number of bins $m_1 = n\zeta_1, m_2 = n\zeta_2$ to mapping n items using Cuckoo hashing, and required bin size β_1, β_2 to mapping n items into m_1 and m_2 bins using Simple hashing.

deduce an upper bound β such that no bin contains more than β items except with probability $1/2^{\lambda}$.

Denote by m_1, m_2 the number of bins used in 3-way "primary" Cuckoo hashing and 2-way "supplementary" Cuckoo hashing, respectively. Let β_1, β_2 denote the maximum bin size when using Simple hashing to map n items to m_1 and m_2 bins with no overflow, respectively. The parameters $m = m_1 + m_2$ and $\beta \in \{\beta_1, \beta_2\}$ presented in Table 5.2. The details of how we obtained these numbers are given in the full version of our paper.

Now within each bin, the receiver has at most one item q and the sender has at most β, call them $\{(x_1, y_1), \ldots, (x_\beta, y_\beta)\}$. They can therefore run the basic OPPRF protocol on these inputs. Note that each of the sender's points (x, y) is mapped to several bins. The OPPRF in each of those bins will be programmed with the same (x, y). That way, if the receiver does have some $q_i = x$, then no matter which of the possible bins it is mapped to in Cuckoo hashing, the receiver will receive the

INPUT OF $\mathcal{S}$: n points $¶ = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$ and $y_i \in \{0,1\}^r$

INPUT OF $\mathcal{R}$: $Q = (q_1, \ldots, q_t) \in (\{0,1\}^*)^t$.

PARAMETERS:

- Hash function $h_1, \ldots, h_5$, number of bins $m \in \{m_1, m_2\}$, and max bin size $\beta \in \{\beta_1, \beta_2\}$, suitable for our hashing scheme (Table 5.2)

PROTOCOL:

1. $\mathcal{R}$ hashes items Q into m bins using the Cuckoo hashing scheme defined in Section 4.4. Let $B_{\mathcal{R}}[b]$ denote the item in the receiver's bth bin (or a dummy item if this bin is empty).
2. $\mathcal{S}$ hashes items $\{x_1, \ldots, x_n\}$ into m_1 bins under 3 hash functions h_1, h_2, h_3, and hashes items $\{x_1, \ldots, x_n\}$ into m_2 bins under 2 hash functions h_4, h_5. Let $B_{\mathcal{S}}[b]$ denote the set of items in the sender's bth bin.
3. For $c \in [1,2]$, for each bin $b \in [m_c]$:
   (a) $\mathcal{S}$ computes $¶_b = \{(x_i, y_i) \mid (x_i, y_i) \in ¶ \text{ and } x_i \in B_{\mathcal{S}}[b]\}$, then pads $¶_b$ with dummy pairs to the maximum bin size β_c
   (b) Parties invoke an instance of $\mathcal{F}_{\mathsf{opprf}}^{F,1,\beta_c}$ with inputs $¶_b$ for the sender and $B_{\mathcal{R}}[b]$ for the receiver.
   (c) $\mathcal{S}$ receives output (k_b, hint_b), and $\mathcal{R}$ receives output $(\mathsf{hint}_b, F(k_b, \mathsf{hint}_b, B_{\mathcal{R}}[b]))$.
4. For each item $q_i \in Q$, let $z_i = F(k_b, \mathsf{hint}_b, q_i)$ where b is the bin to which $\mathcal{R}$ has hashed q_i. The receiver outputs $(\mathsf{hint}_1, \ldots, \mathsf{hint}_m), (z_1, \ldots, z_t)$

Figure 4.5: Hashing-based OPPRF protocol

correct output y.

The formal description of this protocol is given in Figure 4.5. The protocol requires m invocations of a single-query OPPRF, where $m = O(n)$ is the number of Cuckoo hash bins.

In sum, we are able to evaluate OPPRF for large number of programmed points n and large number of queries simply by having players hash their inputs into bins, and evaluate OPPRF per bin on small-size instances.

Caveats. One subtlety in analyzing our construction has to do with the security definition for a programmable PRF. Recall that in that definition (Section 4.3.1), the programmed output (y values) are chosen randomly. Yet in our protocol the sender programs different bins with *correlated* outputs. In particular, when an x_i is mapped to several bins, the OPPRF in each bin is programmed with the same (x_i, y_i) point. To deal with this, we must use the fact that the receiver is guaranteed to never query two bins on the same q (corresponding to the fact that his Cuckoo hashing assigns each q to a unique bin).

4.5 Multi-Party PSI

We now present our main result, an application of OPPRF to multi-party PSI. We use the following notation in this section. We denote the n parties by $P_1, \ldots, P_n$, and use subscripts i and j to refer to individual parties. Let $X_i \subseteq \{0,1\}^*$ denote the input set of party P_i. The goal is to securely compute the intersection $\bigcap_i X_i$. For sake of simplicity, we assume each set has m items and write $X_i = \{x_1^i, \ldots, x_m^i\}$. We use subscript k to refer to a particular item x_k^i.

As discussed at the Introduction (cf. Section 4.2), our PSI protocol proceeds in two consecutive phases, **conditional zero-sharing** and **conditional recon-**

struction of secrets. Importantly, OPPRF is efficient even when run on large input sets, thanks to our use of Cuckoo hash as discussed in Section 4.4.

4.5.1 Conditional Zero-Sharing

We will first describe the end goal of conditional zero-sharing and then discuss how we use multi-query OPPRF of Section 4.4 to achieve it. At the end of this phase, each party P_i will have a mapping $S_i : X_i \to \{0,1\}^*$ that associates each of its items $x_k^i \in X_i$ with an additive secret share $S_i(x_k^i)$. We require the following property: if $x \in \bigcap_i X_i$ (i.e., x is in the intersection), then the corresponding shares $\{S_i(x) \mid i \in [n]\}$ will XOR to zero.

To achieve this, first consider the case of two parties P_1 and P_2. For each item $x_k^1 \in X_1$, party P_1 will choose a random string s_k and record the mapping $S_1(x_k^1) = s_k$. Then the parties can use an instance of multi-query OPPRF as follows. P_1 programs the OPPRF using points $\{(x_k^1, s_k) \mid k \in [m]\}$, and P_2 acts as receiver with input queries X_2. As a result, P_2 will obtain for every $x_k^2 \in X_2$ a corresponding OPPRF output, which we will denote $S_2(x_k^2)$. From the properties of an OPPRF, the mappings S_1 and S_2 have the desired property. If the parties share an item x_k^1 then both will have $S_1(x_k^1) = S_2(x_k^1) = s_k$, corresponding to an XOR-additive sharing of 0. The properties of the OPPRF ensure that P_2 does not know whether he is receiving real shares or random values for any item.

The case of n parties is similar. Each party P_i will act as dealer for each of their items $x_k^i \in X_i$, generating a random additive sharing of zero: $s_k^{i,1} \oplus \cdots \oplus s_k^{i,n} = 0$.

Then each pair of parties P_i and P_j use an instance of OPPRF as follows. P_i programs the OPPRF using points $\{(x_k^i, s_k^{i,j}) \mid k \in [m]\}$, and P_j acts as receiver with input queries X_j. In other words, $s_k^{i,j}$ is the share that is conditionally sent from party P_i to P_j pertaining to item x_k^i.

Now each P_j has acted as OPPRF receiver for all other parties. For each item $x_k^j \in X_j$, the party has an OPPRF output from every sender P_i, along with their own share $s_k^{j,j}$. Denote by $S_j(x_k^j)$ the XOR of all of these values. It is easy to see that these S_j mappings satisfy the desired property. If some x is shared by all parties, then all pairs of parties will exchange shares corresponding to that item. All shares generated by a single party XOR to zero, so all of the $S_j(x)$ values XOR to zero as desired.

4.5.2 Conditional Reconstruction

The second phase of the protocol is a **conditional reconstruction** of secrets. In this phase party P_1 acts as a centralized "dealer." For each item $x \in X_1$ belonging to the dealer, he would like to determine whether x is in the intersection. It suffices for him to obtain all $S_i(x)$ values from all the parties. However, since some parties may not hold item x, they may not have a well-defined $S_i(x)$ value.

This problem can again be solved with an OPPRF. Each party P_i programs an OPPRF instance on points $\{(x, S_i(x)) \mid x \in X_i\}$, and P_1 acts as receiver with PRF queries X_1. Hence, for each item $x \in X_1$, dealer P_1 learns an associated value y^i from the OPPRF with party i. If x is indeed in the intersection, then we expect

PARAMETERS: n parties $P_1, \ldots, P_n$.

INPUT: Party P_i has input $X_i = \{x_1^i, \ldots, x_m^i\} \subseteq \{0,1\}^*$

PROTOCOL:

1. For all $i \in [n]$ and all $k \in [m]$, party P_i chooses random $\{s_k^{i,j} \mid j \in [n]\}$ values subject to $\bigoplus_j s_k^{i,j} = 0$.

2. For all $i, j \in [n]$, parties P_i and P_j invoke an instance of $\mathcal{F}_{\mathsf{opprf}}^{F,m,m}$ where:

   - P_i is sender with input $\{(x_k^i, s_k^{i,j}) \mid k \in [m]\}$.
   - P_j is receiver with input X_j.

   For $x_k^j \in X_j$, let $\widehat{s}_k^{i,j}$ denote the corresponding output of $\mathcal{F}_{\mathsf{opprf}}$ obtained by P_j.

3. For all $i \in [n]$ and $k \in [m]$, party P_i sets $S_i(x_k^i) = s_k^{i,i} \oplus \bigoplus_{j \neq i} \widehat{s}_k^{j,i}$.

4. For $i = 2$ to n, parties P_i and P_1 invoke an instance of $\mathcal{F}_{\mathsf{opprf}}^{F,m,m}$ where:

   - P_i is sender with input $\{(x_k^i, S_i(x_k^i) \mid k \in [m]\}$.
   - P_1 is receiver with input X_1.

   For $x_k^1 \in X_1$, let y_k^i denote the corresponding output for x_k^1 of $\mathcal{F}_{\mathsf{opprf}}$ involving P_i.

5. Party P_1 announces $\{x_k^1 \in X_1 \mid S_1(x_k^1) = \bigoplus_{i \neq 1} y_k^i\}$.

Figure 4.6: Multi-Party PSI Protocol

$\bigoplus_{i \neq 1} y^i = S_1(x)$. Otherwise the left-hand-side will be a random value.

4.5.3 Details and Discussion

A formal description of the protocol is in Figure 4.6.

Correctness. From the preceding high-level description, it is clear that the protocol is correct except in the event of a false positive — i.e., $S_1(x_k^1) = \bigoplus_i y_k^i$ for some $x_k^1 \in X_1$ not in the intersection. Let P_i be a party who did not have x_k^1 in their input set. That party will not program their OPPRF in Step 4 on the point x_k^1. As a result, the term y_k^i is pseudorandom. Hence the probability that of a false positive involving x_k^1 is $2^{-\ell}$. By setting $\ell = \lambda + \log_2(m)$, a union bound shows that the probability of *any* item being erroneously included in the intersection is $2^{-\lambda}$.

Theorem 11. The protocol of Figure 4.6 is secure in the semi-honest model, against any number of corrupt, colluding, semi-honest parties.

Proof. Let C and H be a coalition of corrupt and honest parties, respectively. To show how to simulate C's view in the ideal model, we consider two following cases based on whether all parties in C have item x:

- All parties in C have x and not all parties in H have x: if H contains only one honest party P_i, then P_i does not have x. From the output of set intersection, C can deduce that P_i does not have x. Thus, there is nothing to hide about whether P_i has x in this case.

  Consider the case that H has more than one honest party, say P_i and P_j. Suppose P_i has x, while party P_j does not. So, x does not appear in the intersection. We must show that the protocol must hide the identity of which honest party is missing x.

  In Step 2 of the protocol, there is an OPPRF instance with P_j as sender and P_i as receiver. P_j will not program the OPPRF at point x, so P_i will receive

a pseudorandom output for x that is independent of the corrupt coalition's view. This causes $S_i(x)$ to be independent of the coalition's view.

Later in Step 4, if the dealer is corrupt, both P_i and P_j act as OPPRF senders with the dealer. P_i programs the OPPRF at x using the pseudorandom value $S_i(x)$. P_j doesn't program the OPPRF at point x. The security of OPPRF is that programming the PRF at x with a random output is indistinguishable from not programming at x at all. In other words, parties P_i and P_j have indistinguishable effect on the conditional reconstruction phase. If dealer is honest, the corrupt coalition's view is simulated from Step 2 based on the functionality of OPPRF.

- Not all corrupt parties in C have x: we must show that C should learn nothing about whether any of the honest parties hold x.

  Any honest party P_i who holds x generates corresponding shares $s^{i,j}$, to be conditionally distributed in Step 2. But some corrupt party does not query the OPPRF on x in step 2. This makes all the $s^{i,j}$ shares corresponding to x distributed uniformly. All honest parties P_j who hold x will therefore have $S_j(x)$ uniformly distributed of the coalition's view. In Step 4, the honest parties that hold x will program the OPPRF on $(x, S_j(x))$. The honest parties that don't hold x will not program the OPPRF on point x. As above, programming the PRF with a random output is indistinguishable from not programming at that point at all. Hence all honest parties have indistinguishable effect on the reconstruction phase.

□

Cost and Optimizations. In the conditional sharing phase, each party performs a multi-query OPPRF with every other party. In the reconstruction phase, each party performs just one multi-query OPPRF with the leader P_1. Recall that the cost of each of these is one instance of single-query OPPRF per Cuckoo-hashing bin.

The multi-query OPPRF scales well when sender and receiver have different number of elements. Therefore, our multi-party PSI protocol allows each party's set to have different size. The number of OPPRF instance depends on the number of bins for Cuckoo-hashing, and the OPPRF receiver is the one using Cuckoo hashing (sender uses plain hashing). Thus, our PSI protocol is more efficient by setting the leader P_1 as the party with the *smallest input set*.

We note that all of the OPPRF instances in the conditional sharing phase can be done in parallel, and all the OPPRF instances in the reconstruction phase can as well. This leads to a constant-round protocol.

Finally, recall that the multi-query OPPRF uses Cuckoo hashing. It is safe for *all* such instances, between all pairs of parties, to use the same Cuckoo hash functions. That way, a party only needs to hash their input set twice at the beginning of the protocol (once with Cuckoo hashing for when they are OPPRF receiver, and once with simple hashing for when they are OPPRF sender).

Generalization. Suppose we wish to secure the protocol against the possibility of at most t corrupt (colluding) parties. The default case is to consider $t = n - 1$.

For smaller t, we can simplify the protocol. The idea is to modify the conditional zero-sharing protocol so that party P_i generates shares of zero only for $\{P_{i+1}, \ldots, P_{i+t+1}\}$ (where indices on parties are mod n). The security analysis applies also to this generalization, based on the fact that if P_i is honest, then at least one of $P_{i+1}, \ldots, P_{i+t+1}$ must also be honest.

4.6 Further Optimizations

4.6.1 PSI in Augmented Semi-Honest Model

In this section we show an optimization to our PSI protocol which results in a protocol secure in the augmented semi-honest model.

Unconditional zero-sharing. The previous protocol starts with a *conditional* zero-sharing phase, where parties obtain shares of zero or shares of a random value, based on whether they share an input item x. In this section we propose an *unconditional* zero-sharing technique in which the parties always receive shares of zero.

We describe a method for generating an unlimited number of zero-sharings derived from short seeds that can be shared in a one-time initialization step. The protocol is described in Figure 4.7. The protocol is based on an initialization step where each pair of parties exchange keys for a PRF F, after which each party knows $n-1$ keys. Then, whenever zero-sharing is needed, party P_i generates a share as $S_i(ind) = \bigoplus_r F(r, ind)$, where ind is an index which identifies this

INITIALIZATION: Each party P_i picks random seeds $r_{i,j}$ for $j = i+1, \ldots, n$ and sends seed $r_{i,j}$ to P_j

GENERATE ZERO-SHARING: Given an index ind, each P_i computes

$$S_i(ind) = \left(\bigoplus_{j=1}^{i-1} F(r_{j,i}, ind) \right) \oplus \left(\bigoplus_{j=i+1}^{n} F(r_{i,j}, ind) \right)$$

Figure 4.7: The zero-sharing protocol

protocol invocation, and r ranges over all the keys shared with other parties.

We first observe that the XOR of all $S_i(ind)$ shares is indeed 0, since each term $F(r_{i,j}, ind)$ appears exactly twice in the expression. As for security, consider a coalition of $t < n-1$ corrupt parties, and let P_k be the honest party with smallest index. P_k sends random seeds to all other honest parties. These seeds are independent of all other seeds, and are unknown to the corrupt coalition. They result in set of $n-t-1$ pseudorandom terms that are included in the shares of all honest parties other then P_k. Therefore the shares of the honest parties look pseudorandom to the coalition (subject to all shares XORing to zero).

Plugging into the PSI protocol. Suppose we modify our main PSI protocol (Figure 4.6) in the following ways:

- Instead of steps 1-3, the parties perform the unconditional zero-sharing phase of Figure 4.7. That is, they run the initialize phase to exchange seeds and then set their S_i mappings accordingly.

- Then they continue with Figure 4.6 starting at step 4.

The modification significantly reduces the cost of the zero-sharing phase (which was the most expensive part of Figure 4.6) with a zero-sharing phase that costs almost nothing. Our experiments confirm that this modified protocol is faster than the standard semi-honest-secure protocol, by a significant constant factor.

Correctness of the modified protocol follows from the same reasons as for the unmodified protocol. Namely, if some party P_i does not have an item x, then they will not program their OPPRF with P_1 at point x. This causes P_1 to obtain a random value in the reconstruction phase and subsequently not include x in the output.

Theorem 12. The modified protocol (with unconditional zero-sharing) is secure in the augmented semi-honest model.

Proof Sketch. Consider a coalition C of corrupt parties. We must show how to simulate C's view in the ideal model. If $P_1 \notin C$ then, assuming that the underlying OPRF protocol is secure, the view of C consists only of the output of the invocations of the OPRF protocol (acting as *sender* in each one), and is therefore random. If the leader $P_1 \in C$ then the simulator sends to the ideal PSI functionality the set X_1 as the input of *every* corrupt party (this is the advantage given to the simulator in the augmented security model). Let Z denote the output of the functionality (the intersection of all sets). P_1's view contains OPPRF outputs from all honest parties, corresponding to every $x \in X$. For $x \in Z$, simulate a random sharing of zero as the corresponding OPPRF outputs. For $x \in X_1 \setminus Z$, simulate random values for the corresponding OPPRF outputs. □

Let us give an intuition on why this protocol achieves security only in the augmented model. In this modified protocol, the zero-sharing for each candidate x is generated non-interactively by the parties. So even though a corrupt party P_i does not have an item x, he can non-interactively imagine what his correct share $S_i(x)$ would be. When colluding with P_1, this allows the adversary to learn exactly what would have happened if P_i included x in its set (but only if $x \in X_1$ as well).

In the semi-honest protocol (Section 4.5), however, a corrupt party interacts with honest parties to generate a zero-sharing corresponding to x. At the time of the interaction, the corrupt party P_i "commits" to having x in its input set or not, depending on whether it queries the OPPRF on x. If during the (conditional) zero-sharing phase P_i does not have x in its input set, then there is no way to later guess what the "correct share" would have been.

4.6.2 Reducing OPPRF Hint Size

In this section we look inside the several layers of abstraction in our PSI protocol, and use a global view of things to find room for optimization. We focus on the multi-query OPPRF construction from Section 4.4. Recall that it works in the following way:

- The OPPRF receiver hashes their queries into m bins via a Cuckoo hashing method.
- The OPPRF sender hashes their programming-points into m bins using simple hashing, *for each Cuckoo hash function* (i.e., assigning a single item to many

bins).

- In each bin, the parties perform a single-query OPPRF instance, where the receiver queries on their (unique) item in that bin.

Now look even further inside those single-query OPPRF instances. In each one, the parties invoke an OPRF instance and furthermore the sender gives a "hint" that contains the information to correct/program the OPRF outputs to the desired values.

There are two possible approaches for sending the hints that are required for these OPPRF computations. The straightforward approach sends a separate hint per OPPRF invocation, namely per bin. The other approach sends a single combined hint for all bins. Namely, this combined hint is a single polynomial or Bloom filter, which provides for each of the m possible inputs of P_i the correct "hint" for changing the output of the corresponding OPRF invocation.

The advantage of the "separate hints" approach is that in each OPPRF invocation each party P_i has only $S = \mathcal{O}(\log m / \log\log m)$ points and therefore computing the hint might be more efficient. This is relevant for the polynomial-based hint, since its computation time is quadratic in the size of the set of points. Therefore, the overhead of computing a single combined hint polynomial is $O(m^2)$ whereas the overhead of computing hints for all bins is only $\mathcal{O}(m \log^2 m / \log^2(\log m))$. On the other hand, when computing a hint per bin, the total number of points is $\mathcal{O}(m \log m / \log\log m)$, whereas if a combined hint is used, the total number of points is $\mathcal{O}(m)$. We expect (and validate in the experiments in Section 4.7), that

a combined hint works better for the Bloom filter-based OPPRF, since the cost of this method is linear in the total number of points. On the other hand, the bottleneck of the polynomial-based OPPRF is the quadratic overhead of polynomial interpolation, thus when using that OPPRF it is preferable to use separate hints per bin.

Improvements: We can add the following improvements to the basic protocol:

- In polynomial-based OPPRF with "separate hints", the OPPRF sender does not need to pad with dummy items to the maximum bin size β before interpolating a polynomial over β pairs per bin. Instead of that, he interpolates a polynomial $p_1(x)$ over $k < \beta$ real pairs (x_i, y_i) and then add it with a polynomial $p_2(x)$ of degree $(\beta - 1)$. $p_2(x)$ can be efficiently implemented as $R(x)\prod_{i=1}^{k}(x - x_i)$, where $R(x)$ is a random polynomial of degree $(\beta - 1 - k)$. Using example hashing parameters from Section 4.5, the expected value of k is only 3, while the worst-case $\beta = 30$. This optimization reduces the cost of expensive polynomial interpolation.

- In polynomial-based OPPRF with combined hints, the OPPRF sender can send a combined hint for *each* hash function h_i. That is, for each Cuckoo hash function h_i, the sender computes a hint that reflects *all* of the bin-assignments under that specific h_i. The receiver hashes its items with Cuckoo hashing, and places each item according to exactly one hash function h_i. For each item, the receiver can therefore use the combined hint for that specific h_i.

- In Bloom filter-based OPPRF invocation, each of sender's item appears 5 times in hash table, there are 5 different OPRF values $F(k_{h_i}, x)$). Instead of inserting 5 pairs of the form $(x, y \oplus F(k_{h_i}, x))$ into the GBF, the sender can instead insert the concatenated value $(x, (y \oplus F(k_{h_1}, x)) || \ldots || (y \oplus F(k_{h_5}, x)))$. This reduces the number of the GBF insertions.

4.6.3 3-party PSI in Standard Semi-Honest Model

Our idea for three-party PSI (3-PSI) is to have all 3 players perform an (encrypted) incremental computation of the intersection. Namely, P_1 and P_2 will first let P_2 obtain an encoding of partial intersection $X_{12} = X_1 \cap X_2$. Then P_2 and P_3 will allow P_3 to obtain some encoding of $X_{123} = X_{12} \cap X_3$. In the end, P_1 will decode the output $X_{123} = X_1 \cap X_2 \cap X_3$.

To do this, the leader P_1 chooses a random encoding e_k^1 for each of his inputs x_k^1. P_1 then acts as a sender in OPPRF, programming it on points $\{(x_k^1, e_k^1) \mid k \in [m]\}$. P_2 acts as a receiver in OPPRF using his input set X_2, and obliviously receives either one of these encodings (if his input was a corresponding match) or a random string. Denote by $\widehat{e}_k^2$ the value that P_2 obtains for each of his items x_k^2. The process repeats: P_2 will play the role of OPPRF sender with receiver P_3. P_2 will program the OPPRF on points $\{(x_k^2, \widehat{e}_k^2) \mid k \in [m]\}$ and P_3 will query the OPPRF on his input set X_3. Denote by $\widehat{e}_k^3$ the value that P_3 obtains for each of his items x_k^3.

Finally, P_3 acts as OPPRF sender and programs the OPPRF on points $\{(x_k^2, \widehat{e}_k^2) \mid k \in [m]\}$, while P_1 acts as receiver and queries the OPPRF on points X_1. It is clear

PARAMETERS: 3 parties P_1, P_2, P_3.

INPUT: Party P_i has input $X_i = \{x_1^i, \ldots, x_m^i\} \subseteq \{0,1\}^*$

PROTOCOL:

1. For all $k \in [m]$, party P_1 chooses random distinct $\{e_k^1 \mid k \in [m]\}$ values.
2. Party P_1 and P_2 invoke with an instance of $\mathcal{F}_{\mathsf{opprf}}^{F,m,m}$ where:
   - P_1 is sender with input $\{(x_k^1, e_k^1) \mid k \in [m]\}$.
   - P_2 is receiver with input X_2.

   For $x_k^2 \in X_2$, let $\widehat{e}_k^2$ denote the corresponding output of $\mathcal{F}_{\mathsf{opprf}}$ obtained by P_2.
3. In turn, each party P_i, $i \in \{2,3\}$, invokes with P_{i+1} an instance of $\mathcal{F}_{\mathsf{opprf}}^{F,m,m}$ where:
   - P_i is sender with input $\{(x_k^i, \widehat{e}_k^i) \mid k \in [m]\}$.
   - P_{i+1} is receiver with input X_{i+1}.

   For $x_k^{i+1} \in X_{i+1}$, let $\widehat{e}_k^{i+1}$ denote the corresponding output of $\mathcal{F}_{\mathsf{opprf}}$ obtained by P_{i+1} (indices are mod n)
4. Party P_1 announces $\{x_k^1 \in X_1 \mid e_k^1 = \widehat{e}_k^1\}$.

Figure 4.8: Optimized Three-party PSI Protocol

that if x_k^1 is in the intersection, then P_1 will receive e_k^1 (a value he initially chose) as OPPRF output; otherwise he will receiver a random value. A formal description of the protocol is in Figure 4.8.

Extending the above to $n > 3$ parties faces the following difficulty: If P_1 and P_j collude, they will learn the partial intersection $X_1 \cap \cdots \cap X_j$. Indeed, as an OPPRF receiver, P_j will receive the set of values which can be cross-checked with the encodings generated by P_1. More generally, colluding players P_i and P_j can

compute partial intersection $X_i \cap \cdots \cap X_j$ by comparing their encodings.

We note that this is not an issue in 3-PSI, since colluding P_1 and P_2 can compute $X_1 \cap X_2$ anyway; colluding P_2 and P_3 cannot learn any information about the decrypted key e_i^1 held by P_1 thus the corrupted parties compute $X_2 \cap X_3$ anyway; and colluding P_1 and P_3 can compute $X_1 \cap X_2 \cap X_3$ which is the desired PSI output.

With the above optimization, our 3-PSI protocol needs only 3 OPPRF executions, compared to the 4 OPPRF executions for the general protocol described in Section 4.5. The performance gain of the optimized protocol is not very strong when the network is slow since parties invoke OPPRF in turn and they have to wait for the previous OPPRF completed. We implemented both 3-PSI protocol variants and found this optimized variant to be $1.2 - 1.7\times$ faster.

4.7 Implementation and Performance

In order to evaluate the performance of our multi-party PSI protocols, we implement many of the variants described here. We do a number of experiments on a single server which has 2x 36-core Intel Xeon 2.30GHz CPU and 256GB of RAM. We run all parties in the same network, but simulate a network connection using the Linux *tc* command: a LAN setting with 0.02ms round-trip latency, 10 Gbps network bandwidth; a WAN setting with a simulated 96ms round-trip latency, 200 Mbps network bandwidth.

In our protocol, the offline phase is conducted to obtain an 128 base-OTs us-

ing Naor-Pinkas construction [NP01]. Our implementation uses OPRF code from [KKRT16, Rin]. All evaluations were performed with a item input length 128 bits, a statistical security parameter $\lambda = 40$ and computational security parameter $\kappa = 128$. The running times recorded are an average over 10 trials. Our complete implementation is available on GitHub: `https://github.com/osu-crypto/MultipartyPSI`

4.7.1 Optimized PSI, Augmented Model

In this section we discuss the PSI protocol from Section 4.6 that is optimized for the augmented semi-honest model. We implemented and tested the following variants (see Section 4.6.2 for discussion on variant techniques of sending hints) on different set sizes $m \in \{2^{12}, 2^{14}, 2^{16}, 2^{20}\}$:

- BLOOM FILTER: where the OPPRF used a single combined garbled Bloom filter hint. In our hashing-to-bin scheme, sender uses $h = 5$ hash functions to insert m items into bins. With the optimization in Section 4.6.2, there are only m pairs inserted into the table which has $m\lambda \log_2 e$ entries. The table uses an array of $h(\lambda + \log_2(m))$-bit strings.

- POLYNOMIAL combined: where the OPPRF used combined polynomial hints per hash index. Polynomial interpolation was implemented using the NTL library[Sho]. Each polynomial is built on m points. The coefficients of the polynomial are $\lambda + \log_2(m)$-bit strings.

- POLYNOMIAL separated: where the OPPRF used a separate polynomial hint per bin. The coefficient of the polynomial has $\lambda + \log_2(m)$-bit strings. The degree of polynomial is β_1 for each bin in first $m\zeta_1$ bins, and β_2 for each bin in last $m\zeta_2$ bins. Here $\zeta_1, \zeta_2, \beta_1$ and β_2 are discussed in Table 5.2.

- TABLE: where the OPPRF used a separate table hint per bin. The table has $2^{\lceil \log_2(\beta_1) \rceil}$ entries for each bin in first $m\zeta_1$ bins, and $2^{\lceil \log_2(\beta_2) \rceil}$ entries for each bin in last $m\zeta_2$ bins. Each row has $\lambda + \log_2(m)$-bit strings.

The running times and communication overhead of our implement with 5 parties are shown in Table 4.3. The leader party uses up to 4 threads, each operates OPPRF with other parties. As expected, our table-based protocol achieves the fastest running times in comparison with the other OPPRF constructions. Our experiments show that it takes only one second to sample vector v and check uniqueness for all 2^{20} bins. Thus, the table-based PSI protocol costs only 22 seconds for the set size $m = 2^{20}$. The polynomial-based PSI protocol with separated hint is the next fastest protocol which requires a total time of 38 seconds, a 1.7× slowdown. The slowest protocol is the polynomial-based protocol with combined hint per hash index, whose running time clearly grows quadratically with the set size. However, this protocol has the smallest communication overhead. For small set size $m = 2^{14}$, the polynomial-based PSI protocol with combined hint requires only 1.74MB for communication.

Protocol	Running time (second)				Communication (MB)			
	Set Size m							
	2^{12}	2^{14}	2^{16}	2^{20}	2^{12}	2^{14}	2^{16}	2^{20}
BLOOM FILTER	0.37	0.98	3.41	51.46	8.56	34.26	137.01	2496.2
POLY (combined hint)	7.36	194.96	-	-	**0.43**	**1.74**	-	-
POLY (separate hints)	0.32	0.74	2.33	37.89	1.46	5.98	**24.30**	**447.44**
TABLE	**0.29**	**0.57**	**1.48**	**21.93**	1.64	6.52	25.93	467.66

Table 4.3: The total runtime and communication of our Multi-Party PSI in augmented semi-honest model in LAN setting. The communication cost which ignore the fixed cost of base OTs for OT extension is on the *client's side*. Cells with − denote trials that either took longer than hour or ran out of memory.

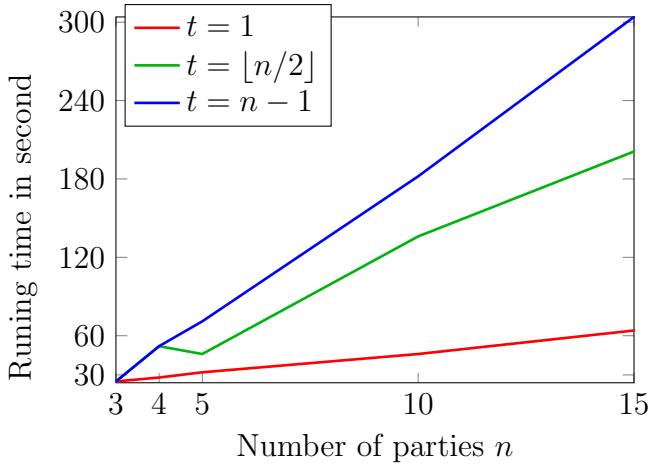


Figure 4.9: Total running time of our semi-honest Multi-Party PSI for the number of parties n, $t < n$ dishonestly colluding, each with set size 2^{20}, in LAN setting.

4.7.2 Standard Semi-Honest PSI

In this section we discuss the standard semi-honest variant of our protocol, using conditional zero-sharing (Section 4.5). From the empirical results discussed in the previous section, the most efficient OPPRF instantiation is the TABLE-based hint. Thus, the OPPRF was instantiated using the TABLE-based protocol in this

Setting	Number Parties n	Threshold Corruption t	Set Size m 2^{12}	2^{16}	2^{20}	2^{24}
LAN	3	$\{1,2\}$	0.21 (0.99)*	1.34 (1.19)*	**25.81 (25.23)***	409.90 (399.67)*
			0.30 (0.16)	**2.14 (1.97)**	**41.64 (41.10)**	702.3 (69.69)
	4	1	0.25 (0.12)	1.80 (1.60)	28.86 (28.27)	484.3 (478.2)
		$\{2,3\}$	0.34 (0.21)	3.16 (2.92)	52.25 (51.65)	865.7(859.4)
	5	1	0.26 (0.12)	1.99 (1.79)	32.13 (31.49)	505.2 (499.2)
		2	0.32 (0.19)	3.44 (3.23)	49.17 (48.54)	-
		4	0.39 (0.26)	4.87 (4.61)	**71.28 (70.60)**	-
	10	1	0.39 (0.17)	2.97(2.71)	46.08 (45.28)	-
		5	0.83 (0.55)	8.79 (8.47)	136.48 (135.44)	-
		9	1.01 (0.72)	12.33 (11.98)	**182.8 (181.60)**	-
	15	1	0.46 (0.23)	4.28 (3.97)	64.28 (63.27)	-
		7	1.37 (0.77)	13.47 (12.79)	201.12 (199.34)	-
		14	**1.85 (1.32)**	20.61 (20.02)	**304.36 (302.17)**	-
WAN	3	$\{1,2\}$	2.82 (2.34)*	10.48 (9.96)*	129.45 (128.64)*	-
			3.12 (2.64)	11.25 (10.73)	158.50 (157.64)	-
	4	1	2.65 (1.97)	12.40 (11.71)	151.9 (150.9)	-
		$\{2,3\}$	3.18 (2.51)	17.47 (16.74)	233.1 (232.1)	-
	5	1	2.66 (1.99)	13.76 (13.06)	185.5 (184.5)	-
		2	3.21 (2.53)	20.29 (19.56)	290.9 (289.8)	-
		4	3.45 (2.78)	25.52 (24.79)	378.5 (377.4)	-
	10	1	3.30 (2.63)	26.42 (25.73)	400.9 (399.8)	-
		5	5.67 (4.98)	76.43 (75.78)	1,194 (1,193)	-
		9	7.81 (7.14)	112.8 (112.1)	1,915 (1,914)	-
	15	1	3.63 (3.15)	39.11 (38.60)	664.08 (662.80)	-
		7	9.87 (9.38)	150.85 (150.31)	2641 (2,640)	-
		14	16.42 (15.96)	263.20 (262.67)	-	-

Table 4.4: Total running time and online time (in parenthesis) in second of our semi-honest Multi-Party PSI for the number of parties n, $t < n$ dishonestly colluding, each with set size m. Number with $*$ shows the performance of the optimized 3-PSI protocol described in Section 4.6.3. Cells with $-$ denote trials that either took longer than hour or ran out of memory.

section.

To understand the scalability of this protocol, we evaluate it on the range of

Number Parties n	Threshold Corruption t	Set Size m 2^{12}	2^{16}	2^{20}	2^{24}
3	$\{1,2\}$	3.28	51.87	935.32	14,860
$\{4,5\}$	1				
$\{10,15\}$					-
4	$\{2,3\}$	4.92	77.80	1,402	22,290
5	2	4.92	77.80	1,402	-
	4	6.56	103.74	1,870	-
10	5	9.84	155.61	2,805	-
	9	14.76	233.41	4,208	-
15	7	13.12	207.48	3,741	-
	14	22.96	363.09	6,547	-

Table 4.5: The numerical communication (in MB) of our Multi-Party PSI in semi-honest setting. The cost is on the *client's side* for the number of parties n, $t < n$ dishonestly colluding, each with set size m. Communication costs ignore the fixed cost of base OTs for OT extension. Cells with $-$ denote trials that either took longer than hour or ran out of memory.

the number parties $n \in \{3,4,5,10,15\}$ on the set size $m \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}\}$. We also wanted to understand the performance effect of the generalization discussed in Section 4.5.3 in which the protocol is tuned to tolerate an arbitrary number t of corrupted parties. In our experiments, we used $t \in \{1, \lfloor n/2 \rfloor, n-1\}$.

Our protocol scales well using multi-threading between n parties. In our implementation, the leader P_1 uses $n-1$ threads and other parties use $\min\{t+1, n-1\}$ threads so that each party operates OPPRF protocol with other parties at the same time. However, we use a single thread to perform the OPPRF subprotocol between two parties.

We proposed a better "hashing to bin" scheme (see the full version of our paper) than the state-of-art two-party PSI [KKRT16]. Specifically, our hashing scheme

removes the stash bins which consume nontrivial cost of the protocol [KKRT16] for sufficiently small sets. For example of 2^{12} set size, we see that our protocol requires 168 milliseconds compared to 211 milliseconds by [KKRT16], a difference of 1.2×.

Results. Table 6.3 presents the running time of our PSI protocol in both LAN and WAN setting. We report the running time for the total time and online phase. The offline phase consists of all operations which do not depend on the input sets. In the three-party case, our protocol supports the full corrupted majority. For $m = 2^{20}$, our general 3-PSI protocol (Section 4.5) in LAN setting costs 42 seconds while the optimized protocol (Section 4.6.3) takes 26 seconds which is 1.6× faster. When evaluating our 3-PSI in WAN setting, we found this optimized variant to be 1.2× faster. This is primarily due to the need to wait for previous OPPRF completed.

To address the possibility of at most t parties colluding, each party performs OPPRF with $\min\{t+1, n-1\}$ other parties. Therefore the cost of the protocol is the same for $t = n-1$ as $t = n-2$. Hence, we report the protocol performance with the $n = 4$ and $t \in \{2, 3\}$ on the same row of the Table 6.3.

As we can see in the table 6.3, our protocol requires only 72 seconds to compute a PSI of $n = 5$ parties for $m = 2^{20}$ elements. For the same set size, when increasing the number of parties to $n = 10$, our total running time is 3 minutes and if $n = 15$ our protocol takes around 5 minutes. Figure 4.9 shows that our protocol's cost is linear in the size of number parties. When assuming only one corrupt party, our

protocol takes only 64 seconds to compute PSI of 15 parties for $m = 2^{20}$ elements. For the small set size of $m = 2^{12}$, the PSI protocol of $n = 15$ parties takes an total time of 1.85 seconds with the online phase taking 1.32 seconds. We find that our protocol also scales to large input sets ($m = 2^{24}$) with $n \in \{3, 4, 5\}$ participants.

Table 4.5 reports the numerical communication costs of our implementation. The protocol is asymmetric with respect to the leader P_1 and other parties. Because the leader plays the role of receiver in most OPPRFs, the majority of his communication costs can be done in an offline phase. Hence we report the communication costs of the clients, which reflects the online cost of the protocol. For the small set size of $m = 2^{12}$, only 3.28MB communication was required in 3-PSI protocol on the client's sides. The communication complexity of our protocols is $\mathcal{O}(mt\lambda)$ bits. Thus, our protocol requires gigabytes of communication for a large set size ($m \in \{2^{20}, 2^{24}\}$). Concretely, for the large input set $m = 2^{24}$, our 3-PSI protocol uses 14.8GB of communication, roughly 0.88KB per item.

Chapter 5: Private Set Union

Scalable Private Set Union from Symmetric-Key Techniques by Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, Xiao Wang in Asiacrypt [KRTW19]

We present a new efficient protocol for computing private set union (PSU). Here two semi-honest parties, each holding a dataset of known size (or of a known upper bound), wish to compute the union of their sets without revealing anything else to either party. Our protocol is in the OT hybrid model. Beyond OT extension, it is fully based on symmetric-key primitives. We motivate the PSU primitive by its direct application to network security and other areas.

At the technical core of our PSU construction is the reverse private membership test (RPMT) protocol. In RPMT, the sender with input x^* interacts with a receiver holding a set X. As a result, the receiver learns (only) the bit indicating whether $x^* \in X$, while the sender learns nothing about the set X. (Previous similar protocols provide output to the opposite party, hence the term "reverse" private membership.) We believe our RPMT abstraction and constructions may be a building block in other applications as well.

We demonstrate the practicality of our proposed protocol with an implementation. For input sets of size 2^{20} and using a single thread, our protocol requires 238 seconds to securely compute the set union, regardless of the bit length of the items. Our protocol is amenable to parallelization. Increasing the number of threads from

1 to 32, our protocol requires only 13.1 seconds, a factor of 18.25× improvement.

To the best of our knowledge, ours is the first protocol that reports on large-size experiments, makes code available, and avoids extensive use of computationally expensive public-key operations. (No PSU code is publicly available for prior work, and the only prior symmetric-key-based work reports on small experiments and focuses on the simpler 3-party, 1-corruption setting.) Our work improves reported PSU state of the art by factor up to $7,600\times$ for large instances.

5.1 Introduction

Private set union (PSU) is a special case of secure two-party computation. PSU allows two parties holding sets X and Y respectively, to compute the union $X \cup Y$, without revealing anything else, namely what are the items in the intersection of X and Y.

5.1.1 Motivation

PSU (like the well-researched private set intersection, PSI) has numerous applications in practice, and tailored efficient solutions are highly desirable. Consider the following use cases. (We note that these use cases cover a wide range of PSU settings, such as multi-party or shared-output PSU. Our work does not address all of the settings, of course, but provides a building block and a baseline for the entire research direction.)

Cyber risk assessment and management via joint IP blacklists and joint vulnerability data. As noted in [LV04, HLS+16], organizations aim to optimize their security updates to minimize vulnerabilities in their infrastructure. Crucial role in the above is played by *joint* lists of blacklisted IP addresses, characteristic network traces and other associated data, as well as *joint* lists of data points reported by vulnerability scanners. At the same time, organizations are understandably reluctant to reveal details pertaining to their current or past attacks or sensitive network data. As convincingly argued in [HLS+16], the use of MPC in computing set unions of the above data sets will mitigate the organizations' concerns. [HLS+16] implements the computation of such set union and related data aggregation as generic MPC in the VIFF framework. As noted by the authors, the major performance bottleneck in their work is private computation of set union. Our tailored PSU algorithms will be applicable to this computation as the main building block.

More generally, privacy-preserving data aggregation is a well-appreciated goal in the network security and other communities. For example, SEPIA [BSMD10] is a library aimed to optimize generic MPC to securely and in real-time compute event correlation and aggregation of network traffic statistics. Our PSU protocol can potentially be helpful in that setting too.

Other applications and use cases. Imagine two Internet providers considering a merger, and they would like to calculate how efficient the resulting joint network would be without revealing the information of their existing networks [BS05]. Another application of combining set-intersection and set-union is the following sce-

nario discussed in [KS05]. A social services organization wants to determine the list of cancer patients who are on welfare. Some patients may have cancer treatment at multiple hospitals. By using a private set union protocol, the union of each hospital's lists of cancer patients can be computed (while removing duplicate patients without leaking the details of the patients), then a secure set intersection operation between the resulting union and the welfare rolls can be performed.

More generally, PSU is an essential building block for private DB supporting full join. Suppose there are two tables owned by two principals, say DMV (Department of Motor Vehicles) and SSA (Social Security Administration). With a PSU-based implementation, a query such as

$$\texttt{SELECT } ssn, dob$$

$$\texttt{FROM } dmv\_db \texttt{ FULL JOIN } ssa\_db$$

$$\texttt{ON } dmv\_db.ssn = ssa\_db.ssn \texttt{ WHERE } dob \geq Jan\ 1, 1980$$

will allow the players to learn the two columns of the union, but not learn whether the other player has the matching record.

Malicious model is of course the ultimate goal in this line of research. At the same time, we believe semi-honest guarantee is sufficient in many scenarios. Further, our work may serve as a stepping stone to the malicious-secure solution where it is required. We believe that our performance improvement of *four* orders of magnitude is surprising for a reasonably researched problem, and sets the baseline for the PSU performance.

5.1.2 Contribution

Over the last decade, there has been a significant amount of work on private set intersection [DKT10, DT10, ADT11, HEK12, DCW13, PSZ14, PSZ18, KKRT16, CDJ16, KMP+17, RR17a, CLR17, HV17, CCS18, FNO18, CO18]. However, there has been little work on PSU, with current PSU state-of-the-art not scalable for big data. Despite similarities between the two functionalities, many effective PSI techniques do not directly apply to PSU. We give a brief discussion about the unsuitability for PSU of several popular PSI techniques in Section 5.4.4 as well as throughout the paper.

We design a truly scalable PSU protocol, building on newly developed building blocks. In detail, our contribution can be summarized as follows:

1. We identify that existing fast private membership tests, used in leading PSI protocols are not immediately applicable for computing PSU (cf. Section 5.2.1), and a richer PMT of [CO18] carries 125× performance penalty (cf. Section 5.1.3). We propose a new building block *reverse private membership test* (PMT) in Section 5.3. We present an efficient instantiation of this building block, which serves as the basis of our symmetric-key based PSU protocol.

2. We apply the bucketing technique to further reduce the computation and communication overhead. We identify and overcome several new challenges unique to bucketing in the context of PSU (but not PSI). Details can be found in Section 5.4.

3. Integrating the above two components, we build a truly scalable system for PSU computation that is *three orders of magnitude* faster than the current reported performance for large two-party PSU instances. Specifically, we are $\approx$ 7,600$\times$ faster than [DC17], which is the current best reported numbers for larger sets of 1 million elements. [BA12] consider an easier setting with three parties and one corruption. Although our protocol works in a stronger model than [BA12], we are still 30$\times$ faster in terms of running time on sets of 2^{12} elements and have $100 - 125\times$ smaller communication (cf. Table 5.3). Our protocol evaluates PSU of two million-element datasets in about a minute on WAN and 13 seconds on a LAN.

4. Our implementation is released on Github: `https://github.com/osu-crypto/PSU`. To our knowledge, this will be the first publicly available PSU implementation.

5.1.3 Related Work

We start by reviewing previous PSU protocols, with particular emphasis on the semi-honest model.

Kissner and Song [KS05]. To our knowledge, the first PSU protocol was proposed by Kissner and Song [KS05]. The PSU of [KS05] is based on polynomial representations and additively homomorphic encryption (AHE). The core idea of their protocol is that if the sets X (respectively, Y) is represented as a polynomial f (respectively, g) whose roots are the set's elements, then the polynomial repre-

Protocol	Comm. (bits)	Comp. [#Ops symm/pub-key]
[KS05]	$O(\kappa^3 n^2)$	$O(n^2)$ pub-key
[Fri07]	$O(\kappa n)$	$O(n \log\log(n))$ pub-key
[BA12]	$O(\kappa \ell n \log(n))$	$O(n\ell \log(n))$ symm
[DC17]	$O(\kappa \lambda n)$	$O(\lambda n)$ pub-key
Ours	$O(\kappa n \log(n))$	$O(n \log(n))$ symm

Table 5.1: Asymptotic communication (bits) and computation costs of two-party PSU protocols in the semi-honest setting. Pub-key: public-key operations; symm: symmetric cryptographic operations. n is the size of the parties' input sets. ℓ is the bit-length item. λ is statistical security parameters. In [BA12] and our protocol, $\kappa = 128$ is computational security parameter, while $\kappa = 2048$ is the public key length in other protocols. We ignore the pub-key cost of κ base OTs.

sentation of the union $X \cup Y$ is $f \times g$. An important property is that an item x is in the set X if and only if $f(x) = 0$. Consequently, for each item e that appears in either set X or Y, it holds that $(f \times g)(e) = f(e) \times g(e) = 0$. The players compute the polynomial $f \times g$ under AHE, and figure out the set of elements based on a procedure called "element reduction", which can reduce the degree of the roots.

Frikken [Fri07]. Relying on the polynomial representation, Frikken [Fri07] proposed a faster PSU protocol with linear communication complexity in the size of the dataset. At the high level, it proceeds as follows. Suppose that $E(f)$ is an encrypted polynomial representation for the set X, a tuple of the form $(xE(f(x)), E(f(x)))$ achieves the specific property that this tuple will be $(0; 0)$ if $x \in X$. In other words, $x \notin X$ can be recovered from the decrypted tuple values. Therefore, instead of computing the encrypted $f \times g$ in [KS05], Bob just computes the above tuples after receiving the encrypted polynomial representation $E(f)$ from Alice, and sends them back to Alice in random order. Alice now decrypts the tu-

ples and learns the value that is not in the intersection. The work of Frikken [Fri07] requires $O(n\kappa)$ communication, where n is the size of the parties' input sets and κ is the length of public-key/ciphertext. Computational cost of generating each tuple is $O(n)$, thus this protocol requires $O(n^2)$ computation. Moreover, their protocol [Fri07] is expensive due to the multi-point evaluation on the encrypted polynomial, which requires the depth of the arithmetic circuit (leveled fully homomorphic encryption) to be logarithm of the input set size. The authors claimed that the computation of their protocol can be reduced to $O(n \log\log(n))$ by using the bucketing technique with minor modifications to their protocol, but it is not clear how to modify it. Indeed, using bucketing is quite tricky for PSU until our work. Based on the polynomial representation, Hazay and Nissim [HN12] extended the Frikken's protocol in the presence of malicious adversaries.

Blanton and Aguiar [BA12]. In 2012, Blanton and Aguiar [BA12] proposed a faster PSU protocol based on oblivious sorting and generic MPC protocols. The core idea of their protocol consists of combining the input sets into a new set, then sorting the resulting set, and comparing adjacent items of the sorted set in order to eliminate duplicates. They focus on constructing the circuit for PSU (and several other set operations) and relegate its evaluation to generic protocols. Their paper provides experimental results on small input set in a three-party and honest majority setting for $32-$bit sized elements. Their largest experiment, $n \leq 2^{11}$, runs in 25 seconds; our $n \leq 2^{12}$ experiment on larger element sizes runs in 1.42 seconds. Importantly, they run the experiments in the three-party setting, where evaluation is much faster as wire secrets can be 1-bit long.

We sketch approximate communication cost of their two-party garbled-circuit-based protocol based on state-of-the-art OT extension and half gates [ZRE15]. Oblivious sorting of 2^{22} elements per player involves sorting a 2^{23} array. Considering 32-bit elements, such 2PC will require approximately $23 \cdot (2^{23}) \cdot (32+32) \cdot 256 = 3,161,095,929,856$ bits = 395 GB. Here 256 is the half-gates garbled table size and 32 is the element size. Subsequent duplicate elimination will cost approximately the same as oblivious sort, so total communication cost is $\approx$ 790GB. Considering larger element size, say, 128 bits, results in the corresponding $4\times$ cost increase, bringing total to $\approx 3.1TB$. Transferring $3TB$ over a 400Mbps WAN will take $\frac{3 \cdot 8 \cdot 10^6}{400} = 60000$ seconds = 16.67 hours. For comparison, our protocol for this size runs in 250 seconds, a $240\times$ improvement.

[BA12] should perhaps be seen as an improvement over current public key-based protocols. As discussed above, our tailored solution outperforms [BA12] by a large factor even in the setting that is the most unfavorable for us. Because there is no reported data on the performance of [BA12] on larger set sizes and no existing generic MPC/2PC system supports large circuits generated by [BA12], we use calculated numbers in our comparison to [BA12] in Table 5.3.

Davidson and Cid [DC17]. Recently, Davidson and Cid [DC17] proposed an efficient protocol based on an encrypted Bloom filter and additively homomorphic encryption (AHE). In the [DC17] protocol, the receiver represents its input set Y using Bloom Filter (BF) with k hash functions, and inverts this filter by flipping the bit value of each entry. It then encrypts the inverted Bloom filter by using an IND-CPA secure AHE scheme, and sends it to the sender. For each item x of its

input set X, the sender uses the k hash functions to retrieve k encrypted BF entries corresponding to x. He then uses AHE homomorphism to sum up under encryption the k retrieved ciphertexts. Let c be the obtained (AHE-encrypted) sum. The sender sends (AHE-encrypted) pairs $\{cx, c\}$ to receiver. Receiver decrypts them and is able to obtain x iff $c \neq 0$. Indeed, if $x \in Y$, all k entries of x are not set in the inverted BF, resulting in $c = 0$. Therefore, the receiver only obtains $X \setminus Y$, from which it computes $X \cup Y$. [DC17] requires $O(\kappa\lambda n)$ communication and $O(\lambda n)$ modular exponentiations, where λ is the statistical security parameters, and κ is the length of public-key/ciphertext, which is in the range 1024-2048 due to their use of public-key primitives. In concrete terms, encrypted BF for the set size $n = 2^{20}$ requires 8.05 GB and 16.1 GB when using a $\kappa = 1024$ bit and $\kappa = 2048$ bit key length, respectively.

Other related work. We note that recent work [CO18] proposed private membership test with shared output, which can be used to instantiate our reverse private membership test. Our RPMT is much faster. For specific parameters used in our work (bucket size 61, bit length 128), [CO18] requires 80KB communication per test while our RPMT construction only needs 0.64KB, a 125× improvement in terms of communication. In addition, our construction requires 140× fewer symmetric-key operations than [CO18]. Because we work with small bucket sizes, our polynomial-based RPMT is fast computationally as well.

Outsourcing PSU was considered in the work of Canetti et al. [CPPT14]. In this problem, users outsource their encrypted data and computation to an untrusted cloud server, while keeping their data private. The main purpose is to

PARAMETERS: A set size n; two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$

FUNCTIONALITY:

- Wait for an input $x^* \in \{0,1\}^*$ from sender $\mathcal{S}$, and an input $X = \{x_1, x_2, \ldots, x_n\} \subseteq \{0,1\}^*$ from receiver $\mathcal{R}$
- Give the receiver $\mathcal{R}$ output 1 if $x^* \in X$ and 0 otherwise.

Figure 5.1: Reverse Private Membership Test Functionality $\mathcal{F}^n_{\mathsf{rpmt}}$.

minimize the computational overhead of the users by utilizing the powerful resources of the cloud server.

Table 5.1 provides a brief comparison to the prior highest-performing PSU protocols in the semi-honest setting. We emphasize that public-key operations are the workhorse of all prior work, while we do only $\kappa = 128$ such operations to initiate OT extension. This is the main reason for 7,600× performance improvement over prior work we observe. We report in detail the performance results and comparisons in Section 6.4.

5.2 Overview of Our Results & Techniques

We start with a special case. Suppose that the sender has only one item y in its set Y and the receiver holding the set X will receive the resulting union $\{y\} \cup X$. The protocol must satisfy the following:

(1) if $y \notin X$, the receiver is allowed to learn y as it is implied by the output. The sender learns nothing.

(2) if $y \in X$, the receiver knows that $y \in X$ (implied by the output), but not

allowed to learn which is the sender's item y. Sender learns nothing.

Receiver learns which of the cases (1) or (2) occurs. Based on the case, the sender's item y can be conditionally sent to the receiver using a "one-sided" OT, a version of OT that requires transfer of a single encrypted secret, rather than the usual transfer of two encrypted secrets, exactly one of which the receiver can decrypt.

5.2.1 Reverse Private Membership Test (RPMT)

We formalize the above basic functionality as the RPMT functionality (cf. Figure 5.1) and design a corresponding tailored efficient protocol, which we believe to be of independent interest. RPMT is related to the traditional Private Membership Test (PMT) [PSZ14], which is a two-party protocol in which the party with input y learns whether or not its item is in the input set X of other party (who learns nothing). In a RPMT, the output is given to the opposite party, i.e. the party holding the set X will learn whether $y \in X$ (and nothing else). We formally describe the ideal RPMT functionality in Figure 5.1.

We emphasize that, unlike PSI, use of PMT is not very natural for PSU. This is because the PMT output receiver holds an element, and gets the answer in plaintext whether the element belongs to a set held by the sender. This is implied by the PSI output, and hence can be used there. However, this is extra information in the PSU functionality. We don't know of a natural way to efficiently use PMT with PSU.

This seemingly simple functionality adjustment (PMT→RPMT) doesn't seem to be fixable by a small tweak of PMT. This is because the underlying primitive used to implement fast PMT [KKRT16] is a variant of OT extension, and the role of OT receiver naturally belongs to the player with a single-element input y; it is not clear how to amend the protocol to allow (only) the other player to receive the output.

The basic idea for our RPMT is to have the receiver represent a dataset X as a polynomial $\widetilde{P}(x)$ whose roots are its elements, and send the (plaintext) coefficients of the polynomial $P(x) = \widetilde{P}(x) + s$ to the other party, where s is a secret value chosen at random by the receiver. The sender evaluates the received polynomial on y and obtains $P(y) = s'$. It is easy to see that $s' = s$ if $y \in X$, i.e. y is a root of $\widetilde{P}(x)$. At this point, the receiver could compare s' and s in the clear and learn the output of RPMT. However, if $y \notin X$, the value $P(y)$ may leak partial information about y. To prevent this, instead of the receiver sending s to the sender, the parties perform a *private equality test/matching* (PM) to determine whether two strings s and s' are equal. The PM guarantees that the sender learns nothing about whether $y \in X$ while the polynomial presentation allows receiver to determine whether $y \in X$ but not the value of y (beyond what is implied by $y \in^? X$).

We note that full PM is actually not required, and a weaker and slightly efficient subprotocol is sufficient. For uninterrupted flow, we return to this observation in Sect. 5.2.2.

This brief overview of RPMT ignores an important security issue. In particular,

suppose $y \in X$, so the sender can evaluate $P(y) = s$. Then he/she can compute $P(\cdot) - s$: a polynomial whose roots are all of the elements of X! To address this issue, the parties invoke oblivious PRF (OPRF) on their inputs, and use the OPRF's outputs for the polynomial interpolation/evaluation. Recall that OPRF is a 2-party protocol in which the OPRF sender learns a PRF key k and the OPRF receiver learns $F_k(z)$, where F is a pseudorandom function (PRF) and z is the receiver's input. In RPMT, the RPMT sender acts as the OPRF receiver to receive $F_k(y)$ and the RPMT receiver acts as the OPRF sender to obtain the PRF key k. Now, the receiver interpolates a polynomial P over points[1] $\{(x, s \oplus F_k(x))\}\ \forall x \in X$, and sends the coefficients of this polynomial to the other party, who evaluates it on y, and outputs $P(y) \oplus F_k(y)$. Thanks to OPRF, the important properties needed for RPMT still hold: (i) $F_k(y) = F_k(x)$ if $x = y$. Therefore, the sender obtains the secret value s chosen by the receiver; (ii) even if $y \in X$, other elements of X can no longer be inferred from $P(\cdot)$ and $P(y)$. This is intended to make finding roots of $P(\cdot) - P(y)$ useless to the sender. Moreover, to learn X, the sender has to know its OPRF value $F_k(x)$, which is not possible because of the OPRF guarantees. A detailed overview of the RPMT protocol is presented in Section 5.3.

We note that RPMT and OPRF are fast cryptographic tools. Recently, Kolesnikov et al. [KKRT16] proposed an efficient protocol which performs many OPRF or PM with amortized cost of 5 μs. Therefore, the main computation cost of our RPMT is the multiplication/evaluation of the polynomial, which requires

[1] Of course, $x \in \{0,1\}^*$ needs to be "hashed down" to an element of the field we are working with. This can be done,e.g., by applying a collision resistant hash function. For simplicity, here we mention, but don't formalize this step.

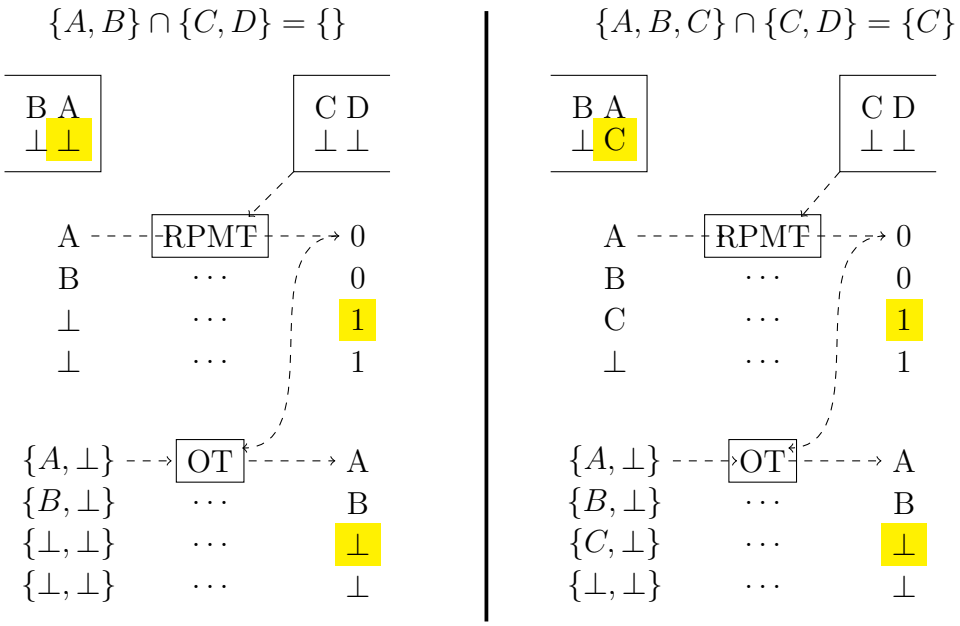


Figure 5.2: Illustration of the main idea behind our protocol: using RPMT and oblivious transfer to perform PSU on a sample bin. The left-hand side illustrates that the sender's bin contains 2 real items $\{A, B\}$ and the receiver's bin contains 2 real items $\{C, D\}$, these sets are disjointed. The right-hand side shows that the sender's bin contains 3 real items $\{A, B, C\}$ and the receiver's bin contains 2 real items $\{C, D\}$, these sets have a common item C. An item $\bot$ denotes the global item known by both parties.

time $O(n \log^2(n))$ using FFT or $O(n^2)$ using a more straight-forward algorithm. This is expensive for large set size $n = |X|$. We avoid the need to work with high-degree polynomials by hashing/bucketing (see below). The communication overhead is small and is equal to $O(n)$.

We can summarize the above gadget for the simple case of PSU (union of a set X and a single element y) as follows: using RPMT on X and y, the receiver learns a bit $b \in \{0, 1\}$ indicating whether $y \in X$. Next, the parties perform one-sided

OT to allow receiver obliviously obtain y if $b = 0$ (i.e. $y \notin X$), nothing otherwise.

5.2.2 An Efficiency Optimization

Going back to the discussion of our RPMT protocol in the previous section, while it uses a PM protocol to compare the output of the polynomial, this is in fact overkill for our application to PSU.

Indeed, suppose the sender instead just sends the output of the polynomial s' in the clear to the receiver. Consider the two cases. First, if $y \in X$, we have $s' = s$, so no information about y would be leaked, as desired. In the other case that $y \notin X$, we want (in the overall PSU protocol) the receiver to learn y anyway! So even if s' leaks information about y, this is fine. Hence, for the purpose of PSU, our protocol can conclude by a plaintext comparison, where the sender sends s' to the receiver.

As it turns out, this optimization, while elegant, is not substantial in terms of overall performance, providing $3 - 5\%$ improvement in running time and $\sim$10% improvement in communication. This can be seen by sketching relative costs of our subprotocols, and is also supported by our experiments. Because of this, we chose to present the paper in terms of the more general and conceptually simpler RPMT primitive.

However, we did formalize and prove secure the improved protocol. It is presented, together with a proof of security and experimental results in the full version of our paper. We feel that this presentation structure allows to focus our main

presentation on the simpler primitives, while at the same time devote sufficient attention to an interesting optimization.

5.2.3 General Case from RPMT

We now discuss how to extend the above approach to the general case of PSU with $|Y| > 1$. The idea is natural: for each item $y \in Y$ simply execute the above gadget on y and X. As a result, the receiver obliviously obtains all items in $Z \leftarrow Y \setminus X$ which directly allows him to learn the union $X \cup Y = X \cup Z$. However, this approach requires n instances of RPMT and n instances of OT (here, we assume that $|X| = |Y| = n$). This results in communication and computation complexity of $O(n^2)$ and $O(n^2 \log(n))$, respectively. Therefore, this PSU construction is only efficient when n is small. Our next trick is to use a hashing technique to overcome this limitation.

At the high level, the idea is that the parties use a hashing scheme to assign their items into bins, and then perform the quadratic-cost PSU on each bin efficiently. By applying a balls-into-bins analysis and minimizing the overall cost, our hashing scheme has $O(n/\log n)$ bins, where each bin contains $O(\log n)$ items. We review the hashing scheme in detail in Section 5.4.2. This optimization reduces costs to $O(n \log n)$ in communication and $O(n \log n \log \log n)$ in computation. However, bucketing introduces a challenge specific to the PSU – the receiver learns additional information on the intersection items, namely, the bucket where the match occurred/did not occur. Consider an example where the receiver's first bin X_1

contains three items and the sender's first bin contains y_1. In our protocol, parties perform RPMT on X_1 and y_1. Suppose $y_1 \in X$, which means, because of bucketing, that $y_1 \in X_1$. From RPMT output, the receiver learns that $y_1 \in X_1$, which cannot be inferred from just the PSU output.

To address this issue, both parties add dummy items $\perp$ into each of their bins to fill them to their maximal size prior to executing RPMT on the bins. Then even if the output of RPMT on $(X_1 \cup \perp)$ and y_1 gives the receiver a bit $b = 1$ (i.e. indicating that $y_1 \in X_1$), the receiver will not learn any information on y_1 since y_1 may be the dummy item $\perp$. We note that this high-level description of the use of dummy items hides some technical nuance, which is explained in detail in Section 5.4.

Figure 6.1 illustrates the main idea behind our protocol. It is easy to see from the Figure 6.1 that the receiver's view in both important cases (two bins are disjoint or two bins have a common item) are exactly same. As noted above, each bin must be padded with $\perp$ to the maximum number of items expected in a bin. In Figure 6.1, the maximum bin size is 4. Section 5.4 formally describes the full construction of our PSU.

5.2.4 Efficiency

Our PSU protocol requires only $O(\kappa)$ public-key operations to perform base OT (which can run in the offline phase). In the online phase, our protocol consists of $O(n)$ OPRF instances, $O(n)$ PM instances, and $O(n)$ OT instances. These

building blocks are based on symmetric-key operations, and can use the same base OTs. In terms of communication, our protocol requires $O(\kappa n \log(n))$, where κ is the computational security parameter.

Our protocol is 3-4 orders of magnitude faster than previous state-of-the art. We present detailed performance analysis and comparisons in Section 6.4.

5.2.5 Using Padding to Hide Input Set Sizes

If desired, it is easy to add padding to our protocol so as to hide the actual sizes of players input sets. This is done simply by setting the protocol parameters (number of bins, maximal bin size) based on the known upper bound of set size. It is easy to verify that this (higher parameter values) do not cause correctness or security violations. Intuitively, players will process more bins with higher maximal bin sizes, but fewer actual items. However, the number of actual items per bin is hidden by our protocol.

5.3 Reverse Private Membership Test (RPMT)

We describe our efficient construction of Reverse Private Membership Test (RPMT), which is a semi-honest secure protocol for the functionality specified in Figure 5.1. Throughout the paper we use the notations κ, λ for the computational and statistical security parameters, respectively. Our RPMT protocol is described in Figure 5.3. The formal protocol follows the intuition presented in the

first part of Section 6.2. Polynomial arithmetic is done in field $\mathbb{F}(2^\sigma)$ for some appropriate σ. We discuss using smaller field size in Section 5.4.3.

RPMT protocol is presented in Figure 5.1. We next argue it computes $\mathcal{F}^n_{\mathsf{rpmt}}$ correctly. Afterwards, we state and prove the security properties of the protocol.

Correctness. The main observation of OPRF is that the RPMT sender (acting as OPRF's receiver) obtains the output q^* which is equal to q_i, if $x^* = x_i$. In this case, it is not hard to see that $s^* = P(h(x^*)) \oplus q^* = P(h(x_i)) \oplus q^* = s$. From the $\mathcal{F}_{\mathsf{pm}}$-functionality, the receiver outputs 1. In case $x^* \notin X$, the OPRF functionality gives the sender q^* which is not in $\{q_i \mid i \in [n]\}$, thus $s^* \neq s$ and the receiver gets 0 from the $\mathcal{F}_{\mathsf{pm}}$-functionality.

We remark that our RPMT protocol is correct except in case of a collision $P(h(x^*)) = P(h(x_i))$ for $x^* \neq x_i$, which occurs with probability is $2^{-\sigma}$. By setting $\sigma = \lambda + \log(n)$, a union bound shows probability of collision is negligible $2^{-\lambda}$.

Security. We now state and prove security properties of RPMT.

Theorem 13. The construction of Figure 5.3 securely implements functionality $\mathcal{F}^n_{\mathsf{rpmt}}$ in the semi-honest model, given the OPRF and Private Equality Test primitives defined in Figure 2.2, and Figure 2.3, respectively.

Proof. We exhibit simulators $\mathsf{Sim}_{\mathcal{R}}$ and $\mathsf{Sim}_{\mathcal{S}}$ for simulating corrupt $\mathcal{R}$ and $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt Sender. $\mathsf{Sim}_{\mathcal{S}}(x^*)$ simulates the view of corrupt $\mathcal{S}$, which consists of $\mathcal{S}$'s randomness, input, output and received messages. $\mathsf{Sim}_{\mathcal{S}}$ proceeds as follows. It first chooses $q' \in_R \{0,1\}^\sigma$, calls OPRF simulator $\mathsf{Sim}_{S_{\mathsf{OPRF}}}(x^*, q')$, and appends its output to the view. We note that BaRK-OPRF is behaving the same as OPRF with respect to the security guarantee needed for simulating this step, namely that q^* obtained in Step 1 is pseudorandom. This is the only direct use of BaRK-OPRF in this protocol, and hence the rest of the argument made w.r.t. OPRF applies to our instantiation as well.

$\mathsf{Sim}_{\mathcal{S}}$ then simulates Step 3 as follows. It generates n random points $(h'_i, p'_i) \in_R (\{0,1\}^\sigma, \{0,1\}^\sigma)$. $\mathsf{Sim}_{\mathcal{S}}$ then interpolates the degree-n polynomial P over these points $\{h'_i, p'_i\}$ and appends its coefficients to the generated view.

Finally, to simulate Step 5, $\mathsf{Sim}_{\mathcal{S}}$ runs simulator $\mathsf{Sim}_{\mathsf{PM}}$ on input $(s' = P(h(x^*)) \oplus q')$ and appends the output of $\mathsf{Sim}_{\mathsf{PM}}$ to its output of the view.

We now argue that the output of $\mathsf{Sim}_{\mathcal{S}}$ is indistinguishable from the real execution. For this, we formally show the simulation by proceeding the sequence of hybrid transcripts T_0, T_1, T_2, T_3, where T_0 is real view of $\mathcal{S}$, and T_3 is the output of $\mathsf{Sim}_{\mathcal{S}}$.

Hybrid 1. Let T_1 be the same as T_0, except the OPRF execution is replaced with choosing a random q' and running the simulator $\mathsf{Sim}_{\mathsf{OPRF}}(x^*, q')$. By the OPRF/BaRK-OPRF pseudorandomness guarantee and the indistinguishability of the output of $\mathsf{Sim}_{S_{\mathsf{OPRF}}}$, it is easy to see that T_0 and T_1 are indistinguishable.

Hybrid 2. Let T_2 be the same as T_1, except that the polynomial is interpolated over points $\{h(x_i), s \oplus F_k(x_i)\}$. Because k is unknown to $\mathcal{S}$, the points sampled by $\mathsf{Sim}_{\mathcal{S}}$ and the points obtained from the real execution are indistinguishable. As a consequence, the polynomial from the real execution can be replaced with a degree-n polynomial P over points $\{h'_i, p'_i\}$. Thus, the polynomial coefficients are indistinguishable.

Hybrid 3. Let T_3 be the same as T_2, except the PM execution is replaced with running the simulator $\mathsf{Sim}_{R_{\mathsf{PM}}}(s')$. Because $\mathsf{Sim}_{R_{\mathsf{PM}}}$ is guaranteed to produce output indistinguishable from real execution, T_3 and T_2 are indistinguishable.

Corrupt Receiver. $\mathsf{Sim}_{\mathcal{R}}(x_1, ..., x_n, out)$ simulates the view of corrupt $\mathcal{R}$, which consists of $\mathcal{R}$'s randomness, input, output and received messages. $\mathsf{Sim}_{\mathcal{R}}$ proceeds as follows. It chooses a random $k' \in_r \{0,1\}^\kappa$, calls OPRF simulator $\mathsf{Sim}_{S_{\mathsf{OPRF}}}(\perp, k')$, and appends its output to the view. Finally, to simulate Step 5, $\mathsf{Sim}_{\mathcal{S}}$ runs simulator $\mathsf{Sim}_{\mathsf{PM}}$ on input (k', out) and appends the output of $\mathsf{Sim}_{\mathsf{PM}}$ to its output of the view.

The view generated by $\mathsf{Sim}_{\mathcal{R}}$ in indistinguishable from a real view because of the indistinguishability of the transcripts of the underlying simulators.

□

Communication Cost. Ignoring the fixed cost of base OTs for OT extension, the PMT communication cost (prior to further optimizations discussed in Section 5.4.3) includes:

- OPRF in Step 1: ρ bits, where ρ is the width of the pseudorandom code defined in Table 5.2 by referencing parameters from [KKRT16].
- Sending the coefficients of P in Step 3: $(n+1)\sigma$ bits
- $\mathcal{F}_{\mathsf{pm}}$ in Step 5: $\rho + \lambda$ bits

Therefore, the overall communication cost of our PMT protocol is

$$\Phi(n) = 2\rho + \lambda + (n+1)\sigma \tag{5.1}$$

5.4 Main Construction

We now present our main result, an application of our RPMT to PSU. The construction closely follows the high-level overview presented in the second part of Section 6.2. Recall, the RPMT functionality allows the receiver to learn one-bit output indicating whether the sender's item is in its (receiver's) set, while keeping this item secret (i.e. the receiver will not know which sender's item is among its set). The performance of our RPMT protocol is linear in the size of the receiver's set, resulting in a quadratic costs for PSU.

Next, in Section 5.4.1, we show how to use a hashing/bucketing technique to overcome this limitation. At the high level, the idea is that each party maps their items into bins using a public hash function. Each bin contains a small number of items which allows the two parties to evaluate RPMT on the elements of each bin separately.

Let m denote the maximum sender's bin size when mapping n items to β bins with no (expected) overflow. Within each bin, the protocol requires $(m+1)$ invocations of RPMT. Section 5.4.2 analyses hashing parameters to minimize the overall cost of our PSU.

5.4.1 PSU Construction

As described above, in our PSU protocol we place players' elements into β buckets of maximum size m each.

We describe the main construction of PSU in Figure 5.4. Correctness of our PSU protocol follows from the fact that the RPMT functionality gives the receiver the zero-bit output if its set does not contain the sender's item. In Step 6b, the receiver obliviously receives that item from OT functionality.

We now state and prove security of our PSU construction.

Theorem 14. The construction of Figure 5.4 securely implements the Private Set Union functionality $\mathcal{F}_{\mathsf{psu}}^{n_1,n_2}$ of Figure 2.5 in the semi-honest model, given the OT and Reverse Private Equality Test primitives defined in Figure 5.1.

Proof. We exhibit simulators $\mathsf{Sim}_{\mathcal{S}}$ and $\mathsf{Sim}_{\mathcal{R}}$ for simulating corrupt $\mathcal{S}$ and $\mathcal{R}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt Sender. When employing the abstraction of the RPMT and OT functionalities, simulating corrupt $\mathcal{S}$ is elementary. $\mathsf{Sim}_{\mathcal{S}}(X)$ simulates the view of corrupt $\mathcal{S}$, which consists of $\mathcal{S}$'s randomness, input, output and received messages.

The simulator simulates an execution of the protocol in which $\mathcal{S}$ receives nothing from the PTM and OT ideal functionality in Step 4. Thus, it is straightforward to see that the simulation is perfect.

Corrupt Receiver. $\mathsf{Sim}_{\mathcal{R}}(Y, Z)$ simulates the view of corrupt $\mathcal{R}$, which consists of $\mathcal{R}$'s randomness, input, output and received messages. We will view $\mathsf{Sim}_{\mathcal{R}}$'s input Z as the set $Z = Y \setminus X$, i.e. the set of elements that X "brings to the union." $\mathsf{Sim}_{\mathcal{R}}$ proceeds as follows.

$\mathsf{Sim}_{\mathcal{R}}$ simulates protocol of Figure 5.4 bucket-by-bucket. Consider the i-th bucket. Let X_i (respectively Y_i, Z_i) be the set of elements of X (respectively, Y, Z) that are mapped to the i-th bucket. $\mathsf{Sim}_{\mathcal{R}}$ pads Y_i to $m + 1$ elements as is done in Step 4. Now, $\mathsf{Sim}_{\mathcal{R}}$ has all the information to simulate Step 6. $\mathsf{Sim}_{\mathcal{R}}$ constructs the sequence simulating when $\mathcal{R}$ discovers new elements in the union. This is an m-element sequence S, where $\mathsf{Sim}_{\mathcal{R}}$ puts $|Z_i|$ elements z_i at randomly chosen slots, and fills the remaining $m - |Z_i|$ elements of the sequence with $\perp$.

$\mathsf{Sim}_{\mathcal{R}}$ then goes through the elements of S. Consider the j-the such element S_j. $\mathsf{Sim}_{\mathcal{R}}$ sets $out_j = 0$ if $S_j = \perp$, and otherwise sets $out_j = 1$. $\mathsf{Sim}_{\mathcal{R}}$ invokes the simulator of $\mathcal{F}_{\mathsf{rpmt}}$ with input (Y_i, out_j), and appends the output of the simulator to its own output. This simulates Step 6a.

$\mathsf{Sim}_{\mathcal{R}}$ proceeds by simulating Step 6b, as follows. $\mathsf{Sim}_{\mathcal{R}}$ invokes the simulator of OT with input (out_j, S_j). This corresponds to $\mathcal{R}$ providing input out_j and receiving output S_j from OT. $\mathsf{Sim}_{\mathcal{R}}$ appends the output of the simulator to its own output.

$\mathsf{Sim}_{\mathcal{R}}$ proceeds simulating each of β bins and terminates. This completes the description of the simulator.

We now argue that the output of $\mathsf{Sim}_{\mathcal{R}}$ is indistinguishable from the real execution. This is easy to see. $\mathsf{Sim}_{\mathcal{R}}$'s reconstruction of how/when the elements of $Z = Y \setminus X$ are discovered by $\mathcal{R}$ is distributed identically to the real execution. The remainder of the simulation refers to simulators of implementations of ideal functionalities.

□

5.4.2 Hashing Parameters

A natural first attempt is to hash n items into n bins, where each bin will contain $O(1)$ items on average. If we could have $O(1)$ items per bin in PSU, this would result in $O(n)$ total RPMT instances, a low cost. However, we must hide the actual number of items in each bin, and hence all bins must be padded to an upper bound m. Gonnet [Gon81] showed $m = \frac{\ln(n)}{\ln\ln(n)}(1 + o(1))$. The coefficient of little-o is not specified in [Gon81]; Pinkas *et al.* [PSZ18] empirically determined the concrete m given the number of bins β. In our case, n bins is not an optimal strategy. For example, hashing $n = 2^{20}$ elements into n bins, bin size $m = 20$ is required to ensure that overflow occurs with probability $\leq 2^{-40}$. As a result, for $n = 2^{20}$ our PSU protocol performs $21n$ RPMT instances in total, which requires 2^{28} OPRF ciphertexts sent and received. We can do better.

In the following, we analyze the effect of the number of bins β and maximum bin size m on the communication overhead of our protocol, and choose the best parameters to minimize our cost. We recall that the overall communication cost

of our PSU protocol is equal to $\beta m\Phi(m+1) + \beta m(\kappa+\sigma)$, where $\Phi(m+1)$ is the RPMT communication cost specified in Equation (5.1). To guarantee that mapping n items to β bins with no overflow, we compute the probability that there exists a bin with more than m items:

$$\Pr(\exists \text{bin with} \geq m \text{ items}) \leq \beta \sum_{m+1}^{n} \binom{n}{i} (\frac{1}{\beta})^{i} (1-\frac{1}{\beta})^{n-i} \tag{5.2}$$

Bounding (5.2) to be negligible in the statistical security parameter $\lambda = 2^{40}$, we obtain the required bin size m without overflow for a given n and β. To minimize the overall communication cost, we choose $\beta = O(n/\log n)$. According to standard balls-and-bins argument, the maximum bin size m is $O(\log(n))$. To determine the coefficients in the big "O", we first fix the number of bins with an initialization value $\beta = \epsilon n = 0.01n$, evaluate Equation (5.2) to obtain the necessary m, and calculate the required communication cost given β and m. In order to find "sweet spot" for our communication cost, we increase the scale ϵ by 0.001 after each time. We observe that our protocol yields the lowest communication when ϵ is in a range $[0.4, 0.6]$. Figure 5.5 shows the result for $n = 2^{16}$: we choose $\beta = \epsilon n = 0.058n$ and require $m = 60$ to achieve 2^{-40} hashing failure probability. We also report the set of our hashing parameters in Table 5.2.

parameters & comm.	set size n							
	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
ρ	424	432	432	440	440	448	448	448
β/n	0.043	0.055	0.05	0.053	0.058	0.052	0.06	0.051
m	63	58	63	62	60	65	61	68
Comm. cost (MB)	0.39	1.81	7.84	33.43	141.78	602.20	2544.7	10748

Table 5.2: Hashing parameters for different set sizes n, and our PSU's communication cost (MB). ρ is OT extension matrix width in OPRF ($\approx$ number of bits required per OPRF call) as reported in Table 1 [KKRT16], β is the number of bins, m is max bin size PSU with n elements per party. Total PSU communication reported in MB and excludes the fixed cost of base OTs for OT extension.

5.4.3 Discussion and Optimization

In our RPMT protocol described in Figure 5.3, the receiver computes a polynomial of degree n with the field of $\mathbb{F}(2^\sigma)$, where $\sigma = \lambda + \log(n)$. With hash-to-bin technique used in PSU, we are able to reduce the degree from n to $m = O(\log(n))$, which avoids an expensive computation at the cost of manipulating polynomials with high degree. However, we increase the field size by $10\% - 12\%$.

Recall that our PSU protocol requires $\beta(m+1)$ RPMT instances in total. For each RPMT protocol, its correctness is violated when a collision event occurs: $P(h(x_i)) = P(h(y_j))$ for $x_i \neq y_j$. To yield collision probability 2^λ over all bins, which is suited for most applications, the size of q_i values is $\sigma = \lambda + \log(\beta(m+1)^2)$. For example, for $n = 2^{20}$, we use the polynomial field size $\mathbb{F}(2^{68})$.

Polynomials with Dummy Points In Step 4, Figure 5.4, receiver pads each bin with one special item $\perp$ and additional different dummies to the maximum bin

size $m + 1$. This padding serves the purpose of hiding the number of items that were mapped to a specific bin, which would leak some information about the input set. In RPMT protocol (Step 2, Figure 5.3), the receiver generates the polynomial over points $\{h(y_i), s \oplus q_i\}$ where some of q_i are the OPRF of the dummy items d_i. Therefore, we simply replace these $q_i = F_k(d_i)$ by random values.

Another optimization, inspired by [KMP+17], is that the receiver computes $P(x)$ by first interpolating the polynomial over the non-dummy items only. That is, receiver interpolates P_0 over $m' \leq m$ points $\{h(y_i), s \oplus F_k(y_i)\}$, and also computes $P_1(x) = \prod_{i=1}^{m'}(x - h(y_i))$ over m' roots $h(y_i)$, where y_i are real items. Then receiver chooses a random polynomial $P_r(x)$ of degree $((m+1)-m')$; and computes $P(x) = P_0(x) + P_1(x)P_r(x)$. It is easy to see that $P(h(x_i)) = s \oplus F_k(x_i),\ \forall x_i \in X$. Using hashing parameters from Table 5.2, the expected value of m' is only 18 for $n = 2^{18}$, while the worst-case $m = 65$. This optimization reduces the cost of expensive polynomial generation (by approximately 200% in our implementation).

Relaxing RPMT Finally, as discussed in Section 6.2, the use of full-fledged RPMT for PSU is slightly overkill. It would suffice to use an RPMT protocol which leaked some information about the sender's item (in the case that $x^* \notin X$), since the PSU protocol will release that value anyway. In the full version of our paper, we describe a simple change to the RPMT protocol that remains secure in the context of our PSU protocol. Basically, instead of using PM to compare polynomial outputs, the sender just sends it polynomial output in the clear. This is safe in the context of PSU since the PSU simulator will have access to the sender's

RPMT input x^* whenever the polynomial output leaks information about x^*.

5.4.4 Discussion: Difficulties in Applying Other PSI Techniques

In addition to the optimizations mentioned above, we also explored other commonly used techniques developed in the context of PSI [FNP04, KKRT16, KMP+17, CLR17, HV17]. Interestingly, we found that many standard techniques for PSI do not directly work for our PSU paradigm, despite the apparent similarity of the two problems. In the following, we will discuss PSU-specific obstacles in applying these techniques. The reader may safely skip this section on the first reading as we discuss here only techniques that we *did not use* in our protocol.

Cuckoo hashing This hashing scheme was introduced by Pagh and Rodler [PR04]. It is the standard hashing scheme in current PSI protocols. At the high level, the receiver uses two (optionally, more) public hash functions h_1, h_2 to store its item in one of the bins $\{h_1(x), h_2(x)\}$. The hashing process uses eviction and the choice of which of the bins is used depends on the entire set. Using the same hash functions and simple hashing, the sender maps its item y into *both* bins $\{h_1(y), h_2(y)\}$ (i.e., item y appears twice in the hash table). Then the parties evaluate PSI bin-by-bin. This is efficient since the receiver has only one item per bin. This hashing scheme avoids a quadratic-cost PSI within a bin.

Unfortunately, this hashing scheme (and the corresponding performance improvement) does not immediately fit in the PSU case. The reason is that the

receiver may learn the Cuckoo hash positions of the sender's items, which may reveal information about sender's entire input. Concretely, suppose that in our protocol the sender uses Cuckoo hashing to map its item x into bin $h_1(x)$. If $x \notin Y$, the receiver will learn which bin x is mapped to. As noted above, the bin storing x depends on the whole input set of the sender and this leaks some information about the party input set that cannot be simulated.

Phasing Permutation-based hashing (phasing) was introduced by Arbitman *et al.* [ANS10] to reduce the bit length of the items that are mapped to bins (in our PSU, this would help reduce the polynomial field size). Phasing was used in [PSSZ15, HV17, RR17b, CLR17] to improve PSI performance when input items has short bit length. The idea is to view each item x as two parts: first $\log(\beta)$ bits used to define the bin to which the item is mapped, and the last bits used as a representation to store the item in the bin.

Concretely, the item x can be presented as $x = x_L|x_R$, where x_L has $\log(\beta)$ bit-length. The item x is mapped into bin $x_L \oplus f(x_R)$, where f is a random function that maps arbitrary strings to a range of $[0, \beta]$. That bin will store x_R as a representative of x. Clearly, x_R has $\log(\beta)$ bits shorter than the original item x. This permutation-based hashing technique achieves significant savings, especially when the original item x has small length (e.g. 32 bits or 64 bits). For instance, assume that the item x has 32-bit length, the set size is $n = 2^{20}$. Then bin elements are only 17 bits long, instead of 32 bits. As a result, we might hope to use the polynomial field size of only $\mathbb{F}(2^{17})$ in RPMT, yielding a significant improvement.

Unfortunately, this general phasing technique does not yield any performance benefit in our PSU paradigm. The underlying reason is that the items in each bin are first given as input to an OPRF for that bin, however the state-of-the-art OPRF protocol that we use ([KKRT16]) is insensitive to the item length. It is only the OPRF output length that determines the field size for polynomial interpolation. Since the OPRF outputs are random, their length must be chosen to avoid collisions with probability $1 - 2^{-\lambda}$.

5.5 Implementation

Our protocol requires the receiver to generate a polynomial of degree m, and the sender to evaluate it on one point, where m is the maximum bin size. Since the degree $m = O(\log(n))$ of the polynomial is relatively small, we use the straightforward Lagrange interpolation and evaluation algorithm which requires $O(m^2)$ field operations. As parties use the bit-string output of the OPRF as input to the polynomial operations, it is natural to interpolate and evaluate the polynomial over $GF(2^\sigma)$. Our polynomial implementation uses the `NTL` library [Sho] with `GMP` library and `GF2X` [GBZT] library installed for speeding up the running time. Inspired by Huang *et al.* [HEKM11], we applied pipelining optimization when the receiver sending all polynomials to the sender. In more detail, we find that by sending polynomial coefficients for 2^8 bins in a batch to the sender, we can minimize the overall wall-clock time of the execution.

As detailed in Section 6.2, our PSU protocol builds on a specific OPRF vari-

ant [KKRT16] and OT extension. We do $\kappa = 128$ Naor-Pinkas OTs [NP01]. We use the source code (OPRF and OT) from [KKRT16, Rin]. Our complete implementation will freely available on GitHub.

We implement our protocol in C++, and run our protocol on a single Intel Xeon with 2.30GHz and 256GB RAM. We emulate the network by using Linux `tc` command. In the following, we compare our protocol to the state-of-the-art PSU protocol [DC17] which provides empirical experiments for a larger set, and the work of [BA12] which reports experimental numbers for PSU of small sets $n \leq 2^{12}$). Additionally, we demonstrate the scalability and parallelizability of our protocol by evaluating it on sets of up to 2^{22} 128-bit items each.

All comparisons are total running time. We note that our protocols are very amenable to pre-computation (by precomputing and pre-sending OT extension and OPRF matrices).

5.5.1 Comparison with Prior Work

Since implementation of [DC17] and [BA12] are not publicly available, we use their reported experimental numbers. We perform a comparison on the range of set sizes $n = \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$ to match the parameters used in [DC17, Table 3&4] and [BA12, Table 3]. [DC17] ran experiments on Intel Xeon 3.30GHz 256GB RAM and 10Gbps LAN; we use a similar (1.32× slower) machine as reported above and same LAN. [BA12] reports running on 2.4GHz AMD Opteron.

	Protocol	Bit key length	Cryptographic strength	Set size n 2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
Time	[DC17]	1024	Legacy	11.78	44.73	175.7	702.4	2836.5	11341.2
		2048	112	78.02	312.44	1233.59	4952.94	19881.51	79272.48
	[BA12]	128	128	2.41	11.88	24.88	–	–	–
	Ours	128	128	**0.57**	**0.66**	**0.83**	**1.15**	**2.65**	**10.42**
		Speedup		4×	18×	30×	4306×	7502×	7607×
Comm.	[DC17]	1024	Legacy	2.83	11.32	45.28	181.12	724.49	2897.97
		2048	112	4.06	16.25	65.01	260.04	1040.18	4160.74
	[BA12]	128	128	75.5	369.1	1744.83	8053.06	36507.22	163208.76
	Ours	128	128	**0.45**	**2.05**	**8.48**	**34.98**	**144.65**	**652.09**
		Speedup		9.02×	7.92×	7.66×	7.43×	7.19×	6.38×

Table 5.3: Comparison of total runtime (in seconds) and communication (in MB) between our protocol, [DC17] and [BA12]. Both parties have n 128-bit elements as input, except [BA12] running time is based on 32-bit elements. [DC17] implementation is in Go, using 8 threads. Our implementation is in C++, 8 threads. [DC17] and us use fast emulated LAN (10Gbps, 0.02ms RTT). Cryptographic strength refers to the computational security of the protocol, according to NIST recommendations. [BA12] runtime is taken from their 3-party experiments, and [BA12] communication is calculated by us for 2PC and 128-bit elements. Best results are marked in bold.

Runtime Comparison In the [DC17] protocol, a Bloom filter (BF) of $44n$ elements is used to yield the false-positive probability 2^{-30}. Each element requires expensive encryption, decryption and further manipulation under an additively-homomorphic encryption (AHE).

We report detailed comparisons in Table 5.3, and here we highlight some numbers. Our protocol runs in 0.94 seconds for $n = 2^{10}$, while [BA12] requires 11.88 seconds, a factor of 18× improvement; and [DC17] requires 312.44 seconds with 2048-bit key length (which corresponds to the security level considered in our protocol), a factor of 332× improvement. As the set size n increases, [DC17] runs

correspondingly slower. When increasing the set size to $n = 2^{18}$, [DC17]'s overall running time is $79,272.48$ seconds while ours is only 10.42 seconds.

This is a 7607× improvement in running time compared to [DC17] (2048-bit key length). A higher improvement factor as we move to higher set size likely indicates that non-protocol-essential system overheads take a higher fraction of resources in smaller set size executions in our protocol. In Section 5.5.2, we demonstrate the scalability and parallelizability of our protocol.

Bandwidth Comparison The receiver in [DC17] sends a large encrypted BF. For $n = 2^{20}$, BF size is 8.05 GB and 16.1 GB when encrypted with 1024-bit and 2048-bit key, respectively. [BA12] relies on generic 2PC/MPC to run their protocol. We sketch approximate communication cost of their protocol in the two-party setting based on state-of-the-art OT extension and half gates (cf. discussion in Section 5.1.3). Oblivious sorting of n elements per party involves sorting an array of size $2n$. Considering ℓ-bit elements, this will require approximately $2n \cdot \log(2n) \cdot 2\ell \cdot 256$ bit. Here 256 is the half-gates garbled table size. The communication complexity of the duplicate elimination [BA12] costs approximately the same as oblivious sort. For the bandwidth comparison, we only report the [BA12]'s communication cost of oblivious sorting and duplicate elimination, which is in favor of their protocol.

We compare bandwidth for the set sizes explored in [DC17], and summarize their and our results in Table 5.3. The communication cost of our protocol is significantly less than that of the prior work. Concretely, for $n = 2^{18}$, our protocol

requires 652.09 MB of communication, a 6.38 × improvement. For very small set size $n = 2^8$, our protocol requires only 0.45 MB while [DC17] needs 4.06 MB and [BA12] requires at least 75.5 MB.

Correctness error probability In [DC17] protocol, Bloom filter introduces a false positive error in the output. Recall, the false positive rate (FPR) is the probability that a *single* element is mistakenly marked as being in the set. The [DC17]'s implementation chooses FPR of 2^{-30}. Thus, computing the set union of 2^{-18} items each, the probability that the entire output includes a false positive is 2^{-12}. We use simple hashing with probability of existence of an overflowed bin of 2^{-40}. Thus, in our protocol, the correctness error probability 2^{-40} is per *whole* set, not per single item.

5.5.2 Scalability and Parallelizability

We demonstrate the scalability and parallelizability of our protocol by evaluating it on set sizes $n = \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}\}$. We run each party in parallel with $T \in \{1, 4, 16, 32\}$ threads. We report the performance of our protocol in Table 6.3, showing running time in both LAN and WAN settings: a LAN setting with 10 Gbps network bandwidth and 0.02 ms round-trip latency; a WAN setting with 400 Mbps network bandwidth and a simulated 40 ms round-trip latency.

Our protocol indeed scales well. Small-size problems are sub-second; medium-size problems ($n = 2^{14}$) are 3.54 seconds and larger sizes ($n = 2^{20}$) is under 4

Setting	T	Set size n							
		2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
LAN	1	0.66	0.86	1.42	3.54	12.41	61.34	238.88	1039.64
	4	0.59	0.69	0.98	1.46	4.03	17.94	69.07	301.76
	16	0.55	0.66	0.78	0.97	1.82	6.29	21.9	90.99
	32	0.53	0.63	0.69	0.84	1.56	4.1	13.09	54.63
WAN	1	1.38	1.73	2.61	6.96	23.29	102.5	406.15	1679.85
	4	1.33	1.56	1.99	3.29	8.58	31.05	118.79	463.51
	16	1.25	1.39	1.76	2.55	5.61	18.67	70.55	280.15
	32	1.22	1.33	1.57	2.4	5.02	17.08	62.96	250.97
Speedup		1.13-1.24 ×	1.3-1.36×	1.66-2.06×	2.9-4.22×	4.64-7.98×	6-14.9×	6.5-18.2×	6.7-19.1×

Table 5.4: Scaling of our protocol with set size and number of threads. Total running time is in seconds. n elements per party, 128-bit length element, and threads $T \in \{1, 4, 16, 32\}$ threads. LAN setting with 10Gbps network bandwidth, 0.02ms RTT. WAN setting with 400Mbps network bandwidth, 40ms RTT.

minutes, all single-threaded. Increasing the number of threads runs the $n = 2^{20}$ instance in 13.09 seconds, a four orders of magnitude improvement over prior work. Benchmarking our implementation in the WAN setting, our protocol also scales well due to the fact that the communication cost is reasonable (for $n = 2^{18}$, our protocol needs 652.09 MB of communication).

Our protocol is very amenable to parallelization. Specifically, our algorithm can be parallelized at the level of bins. For example, when increasing the number of threads from 1 to 32, our protocol shows a factor of 19× improvement as the running time reduces from 1039.64 seconds to 54.63 seconds for an input of $n = 2^{22}$ elements.

Of particular interest is the last row, which presents the ratio between the runtime of the single thread and 32 threads. Our protocol yields a better speedup

when the set size is larger. For smallest set size of $n = 2^8$, the protocol achieves a moderate speed up of about 1.13. When considering the larger database size $n = 2^{22}$, the speed up of $3.4 - 3.6$ is obtained at 4 threads and $6.7 - 19.1$ at 32 threads.

PARAMETERS:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$
- Set X is of size n of elements.
- The bit-length of field elements $\sigma = \lambda + \log(n)$.
- Ideal OPRF, $\mathcal{F}_{\mathsf{pm}}$ primitives specified in Figure 2.2, and Figure 2.3, respectively. Let $F_k(x) : \{0,1\}^* \mapsto \{0,1\}^\sigma$ be the underlying OPRF function.
- A collision-resistant hash function $h(x) : \{0,1\}^* \mapsto \{0,1\}^\sigma$.

INPUT OF $\mathcal{S}$: $x^* \in \{0,1\}^*$

INPUT OF $\mathcal{R}$: $X = \{x_1, x_2, \ldots, x_n\} \subseteq \{0,1\}^*$

PROTOCOL:

1. $\mathcal{S}$ acts as OPRF *receiver*, sends x^* to OPRF. $\mathcal{S}$ receives $q^* = F_k(x^*)$ and receiver $\mathcal{R}$ receives k.
2. $\mathcal{R}$ chooses $s \leftarrow \mathbb{F}(2^\sigma)$ at random. $\mathcal{R}$ interpolates a $\mathbb{F}(2^\sigma)$-polynomial $P(x)$ over points $\{(h(x_i), s \oplus q_i)\}$, where $q_i = F_k(x_i), \forall i \in [n]$. Here $s \oplus q_i$ is computed as operation on σ-bit strings.
3. $\mathcal{R}$ sends the coefficients of P to $\mathcal{S}$.
4. $\mathcal{S}$ computes $s^* = P(h(x^*)) \oplus q^*$.
5. $\mathcal{S}$ and $\mathcal{R}$ invoke the $\mathcal{F}_{\mathsf{pm}}$-functionality:
   - $\mathcal{R}$ acts as *receiver* with input s.
   - $\mathcal{S}$ acts as *sender* with input s^*.
   - $\mathcal{R}$ receives output from $\mathcal{F}_{\mathsf{pm}}$.

Figure 5.3: Reverse Private Membership Test Protocol $\mathcal{F}^n_{\mathsf{rpmt}}$.

PARAMETERS:

- Set sizes n_1 and n_2, and two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$
- A bit-length ℓ. Let $n = \max(n_1, n_2)$.
- Number of bins $\beta = \beta(n)$, and max bin size m, suitable for our hashing scheme (Table 5.2)
- Ideal $\mathcal{F}_{\mathsf{rpmt}}$ primitive defined in Figure 5.1, and ideal OT primitive.
- A special dummy item $\perp \in \{0,1\}^*$

INPUT OF $\mathcal{S}$: $X = \{x_1, x_2, \ldots, x_{n_1}\} \subseteq \{0,1\}^\ell$

INPUT OF $\mathcal{R}$: $Y = \{y_1, y_2, \ldots, y_{n_2}\} \subseteq \{0,1\}^\ell$

PROTOCOL:

1. Randomly pick a hash function H from all hash functions with domain $\{0,1\}^\ell$ and range $[\beta]$.
2. $\mathcal{S}$ and $\mathcal{R}$ hash elements of their sets X and Y into β bins under hash function H. Let $B_{\mathcal{S}}[i]$ and $B_{\mathcal{R}}[i]$ denote the set of items in the sender's and receiver's i-th bin, respectively.
3. $\mathcal{S}$ pads each bin $B_{\mathcal{S}}[i]$ with (several copies, as needed) the special item $\perp$ up to the maximum bin size $m+1$, and randomly permutes all items in this bin.
4. $\mathcal{R}$ pads each bin $B_{\mathcal{R}}[i]$ with one special item $\perp$ and (several, as needed) different dummy items to the maximum bin size $m+1$.
5. $\mathcal{R}$ initializes set $Z = \{\}$.
6. For each bin $i \in [\beta]$, for each item $x_j \in B_{\mathcal{S}}[i]$:
   (a) $\mathcal{S}$ and $\mathcal{R}$ invoke the $\mathcal{F}_{\mathsf{rpmt}}$-functionality:
      - $\mathcal{S}$ acts as *sender* with input x_j
      - $\mathcal{R}$ acts as *receiver* with input set $B_{\mathcal{R}}[i]$
      - $\mathcal{R}$ obtains bit b_j.

   (b) $\mathcal{S}$ and $\mathcal{R}$ invoke the OT-functionality:
      - $\mathcal{S}$ acts as *sender* with pair-input $\{x_j, \perp\}$
      - $\mathcal{R}$ acts as *receiver* with bit input b_j
      - $\mathcal{R}$ obtains the OT output z_j and sets $Z = Z \cup z_i$.
7. $\mathcal{R}$ outputs $Y \cup Z$.

Figure 5.4: Private Set Union Protocol $\mathcal{F}_{\mathsf{psu}}^{n_1,n_2}$.

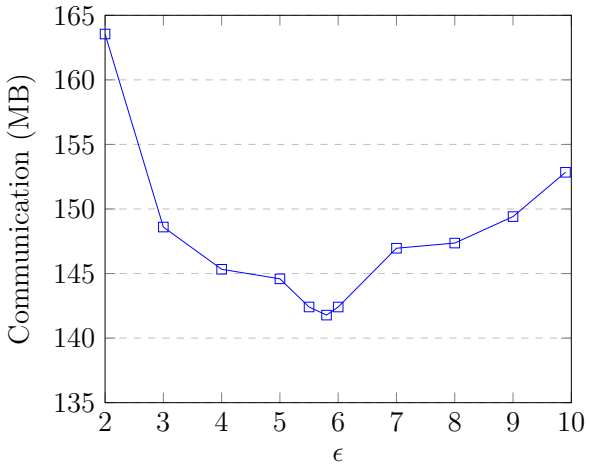


Figure 5.5: Communication cost (MB) of our PSU protocol for $n = 2^{16}$ given the number of bins $\beta = 10^{-2}\epsilon n$

Chapter 6: PM Advanced Variant

SWiM: Secure Wildcard Pattern Matching From OT Extension by Vladimir Kolesnikov, Mike Rosulek, Ni Trieu in Financial Cryptography [KRT18]

Suppose a server holds a long text string and a receiver holds a short pattern string. Secure pattern matching allows the receiver to learn the locations in the long text where the pattern appears, while leaking nothing else to either party besides the length of their inputs. In this work we consider secure *wildcard* pattern matching (WPM), where the receiver's pattern is allowed to contain wildcards that match to any character.

We present SWiM, a simple and fast protocol for WPM that is heavily based on oblivious transfer (OT) extension. As such, the protocol requires only a small constant number of public-key operations and otherwise uses only very fast symmetric-key primitives. SWiM is secure against semi-honest adversaries. We implemented a prototype of our protocol to demonstrate its practicality. We can perform WPM on a DNA text (4-character alphabet) of length 10^5 and pattern of length 10^3 in just over 2 seconds, which is over two orders of magnitude faster than the state-of-the-art scheme of Baron et al. (SCN 2012).

6.1 Introduction

Secure two-party computation allows mutually untrusted parties to perform a computation on their private inputs without revealing any additional information except for the result itself. Over the last few years, secure two-party computation has been extensively studied and has become practical for a variety of applications [MNPS04, KS08, ZRE15, GLMY16, KNR+17, WRK17] . Two adversarial models are usually considered. In the semi-honest model, the adversary is assumed to follow the protocol, while trying to learn information from the protocol transcript. In the malicious model, the adversary can follow an arbitrary polynomial-time strategy. We consider the semi-honest model in this work.

Pattern matching is a basic problem in secure computation. It has been extensively studied in the past decade, e.g., [HL08, BEDM+12, DF13, DCFT13, FHV13, YSK+13, HT14, CS15, YSK+14, WJW+15, WZX17]. Pattern matching is frequently used in text processing, database search [GHS10, CS15], network security [NN10], DNA analysis [OPJM10], and other practical algorithms. The most commonly considered variant of secure pattern matching, which we will call exact PM, is the setting where a server with input a text $x \in \Sigma^n$ (over some alphabet Σ) interacts with a receiver with input a pattern $p \in \Sigma^m$ (for $m < n$). The receiver learns where the pattern occurs as a substring of the server's text without revealing any additional information. There are several important variants of pattern matching, including approximate pattern matching and outsourced pattern matching, which we discuss in Section 6.1.2.

In this work, we focus on secure pattern matching with *wildcards*, which we will call WPM. In this variant, the receiver's pattern can include wildcard characters that can match any character in the data, hence $p \in (\Sigma \cup \{\star\})^m$. With wildcards, the security requirements are more demanding: the server should not learn which positions of p contain wildcards, and in the case of a match the receiver should not learn the text character that matches a wildcard character in the pattern (unless this could be inferred from the presence or absence of an overlapping match).

Allowing wildcards in a pattern matching functionality has been well studied in the absence of a security requirement [CH02, CWZ+06, CC07, CEPR09, SOF10, Tha11, BGVV14, BI14, SSSS15, AWY15], and is motivated by the goal of providing the facility of searching with errors/unknowns. Privacy issues arise in searching on sensitive data and secure pattern matching with wildcards has applications, e.g., in computational genetics and DNA analysis. Indeed, consider the case of a hospital or biomedical research center holding patient genomic data, and a researcher holding a specific cancer marker sequence with some errors. The researcher wishes to know the frequency and positions of the gene occurrences in the database. Due to the genome's highly sensitive nature, the hospital is required keep genomic data private, while the researcher needs to protect specific genome sequence he is working on. The abundance of WPM applications, such as privacy-preserving DNA matching described above, is our main motivation for improving the state-of-the-art in secure wildcard pattern matching.

6.1.1 Pattern Matching with Wildcards

In this section, we discuss directions and related work that achieves, or can be naturally used to achieve, the WPM functionality in the semi-honest setting.

Circuit based. Generic secure computation protocols [Yao86, GMW87], allowing evaluation of arbitrary functions, have seen tremendous performance improvements in the last decade. Modern garbled circuit (GC) protocols evaluate two million AND gates per second on a 1Gbps LAN. Several garbled circuits for Pattern Matching and its variants were studied in [JKS08, KM10]. The best protocol using this technique were proposed by Katz and Malka [KM10]. The authors showed how to modify Yao's garbled circuit to solve Pattern Matching where the size of circuit is linear in the (*a priori* upper bound on the) number of occurrences of the pattern in text. While it is possible to extend circuit-based protocol [KM10] to allow wildcards, it would still require a bound on the number of matches to be provided *a priori* for the circuit construction. When such bound is high or simply unknown, their protocol suffers corresponding performance penalty. The work [KM10] does not provide implementation or experimental results.

Homomorphic encryption based. To our knowledge, Hazay and Toft [HT10] were the first to explicitly consider wildcard secure pattern matching. The core idea of their protocol is that the receiver provides the wildcard positions to the server in an encrypted form, and the substrings of the server's text are obliviously modified so as to match the pattern at those positions. Later, Vergnaud [Ver11] improved the

work of [HT10] by employing Fast Fourier Transform. Both works rely on the fact that if a pattern bit p_i is equal to a text bit t_i, then $(t_i - p_i)^2$ equals 0, and otherwise it is equal to 1. The work of Vergnaud [Ver11] requires $O((n+m)\kappa^2)$ communication and $O(n \log m)$ computational cost in both semi-honest and malicious settings, where κ is computational security parameters. As [HT10, Ver11] do not provide the experimental results, we do not compare execution times with their work.

In 2012, Baron et al. [BEDM+12] proposed an efficient pattern matching protocol called 5PM, for 5ecure Pattern Matching. They consider both malicious and semi-honest models. 5PM works with character (non-binary) wildcards, and was the first to provide an accompanying implementation. The protocol is based on an insecure pattern matching algorithm proposed by Hoffmann [HHD11]. To obtain a secure pattern matching, 5PM modifies the algorithm [HHD11] to work with basic linear operations, which allowed instantiation with additive homomorphic encryption. 5PM requires $O(n\kappa)$ communication and $O(n + m)$ computational costs in semi-honest setting. In Section 6.4 we compare our performance to that of 5PM and report $2 - 499\times$ performance improvement even on medium-size instances.

Yasuda et al. [YSK+14] extend the exact pattern matching protocol of [YSK+13] to support wildcards. The security of [YSK+14] is based on the polynomial LWE assumption. Their scheme operates by blocks, limited by the lattice dimension; for larger inputs $\boldsymbol{x}$, inefficiency is introduced either by using a larger lattice, or by the difficulty and cost of handling boundaries of blocks. In [YSK+14], the authors do not present the performance comparison with 5PM protocol, but indirectly this can be calculated. Yasuda et al. [YSK+14] mention

that their protocol only $4-5\times$ slower than the protocol [YSK+13], which does not allow wildcards. In addition, [YSK+13] estimated that their work is about $10\times$ faster than 5PM when using much stronger hardware than 5PM ([YSK+13] experiments were performed on Intel Xeon X3480 3.07 GHz machine with 16 GB RAM, while 5PM [BEDM+12] used Intel dual quad-core 2.93GHz machine with 8GB RAM). Putting all together, we conclude that [YSK+14] is approximately $2-2.5\times$ faster than 5PM. In contrast, our protocol is $2-499\times$ times faster than 5PM, while running on weak commodity hardware (same at 5PM, cf. Section 6.4.1); this translates into the corresponding improvement over [YSK+14] as well. Further, our approach is simpler and easier to implement.

We mention recent work of Saha and Koshiba [SK17], which improves on the work of [YSK+14] by proposing a new packing method that efficiently addresses continuous wildcards occurring in the pattern (e.g., pattern `10⋆⋆⋆⋆01⋆⋆⋆110` has $k = 3$ sub-patterns: `10`, `01`, and `110`). The main idea of their packing method is to let the receiver break down the pattern into k sub-patterns and have the parties perform the traditional pattern matching on these patterns. This solution is about $k\times$ faster than previous work [YSK+14]. However, it reveals significant information about the pattern, especially for larger k.

6.1.2 Variants of Pattern Matching

For completeness, we briefly discuss work on several additional variants of secure pattern matching.

Exact pattern patching. To our knowledge, Troncoso-Pastoriza et al. [TKC07] were the first to consider secure pattern matching. Their protocol is based on oblivious automaton evaluation. The protocol [TKC07] requires $O(nm)$ communication and computational cost. Several follow-up works [Fri09, MNSS12] improved the computational cost and reduced the round complexity. Another line of work [HL08, GHS10] is based on oblivious pseudorandom functions (OPRF), and obtains security in the malicious setting using $O(nm)$ communication and computational cost with $O(m)$ rounds. De Cristofaro et al. [DCFT13] consider a secure and efficient pattern matching protocol which hides the length of the pattern.

Approximate/fuzzy pattern matching. The functionality of this problem is to find the text positions matches approximately (rather than exactly). This problem can be solved by determining whether the Hamming distance between each text substring and the pattern is less than a threshold t. Hazay and Toft [HT10, HT14] proposed a malicious-secure solution with $O(nt)$ communication and $O(nm)$ computation costs.

Outsourcing pattern matching. In this setting, parties outsource their encrypted data and computation to an untrusted server, while maintaining data privacy. The main goal here is to minimize the communication and computational overhead of the parties by relying on the powerful resources of the untrusted server. The first work that considered secure pattern matching in the cloud setting can be traced back to Faust et al. [FHV13]. Other follow-up works are [WJW+15]. Recently, Wei et al. [WZX17] proposed an efficient solution by combining a secret

Protocol	Computation		Communication		Rounds		Security
	Online	Offline	Online	Offline	Online	Offline	Model
[Ver11]	$\mathcal{O}(n \log m)$		$\mathcal{O}((m+n)\kappa^2)$		$O(1)$		semi-honest & malicious
[BEDM+12]	$\mathcal{O}(mn)$		$\mathcal{O}((m+n)\kappa^2)$		8		malicious
	$\mathcal{O}(m+n)$		$\mathcal{O}(n\kappa)$		2		semi-honest
Ours	0	$\mathcal{O}(\kappa)$	$\mathcal{O}(m+(\lambda+\kappa)n)$	$\mathcal{O}(nm)$	2	2	semi-honest

Table 6.1: Communication (bits) and computation (number of exponentiations) complexities of WPM protocols, where n is length of text, m is length of pattern; and λ and κ are the statistical and computational security parameters, respectively. $\lambda = 40$ and $\kappa = 128$ in our protocols, while κ is in the range 1024-2048 in [Ver11, BEDM+12] protocols (due to their use of public-key primitives).

sharing scheme and oblivious transfer which requires $O(\kappa)$ computation and $O(mn)$ communication costs. Outsourcing pattern matching can be viewed as substring searchable encryption which are studied in [CS15, ÇCL+17]

6.2 Overview of Our Results and Techniques

In this work we present SWiM (**S**ecure **Wi**ldcard Pattern **M**atching), a protocol for WPM based on two fast cryptographic tools: oblivious transfer and secure string equality test (given two strings of equal lengths, without wildcards, determine whether they are equal). Thanks to recent optimizations in oblivious transfer protocols [Bea96, IKNP03, KK13, ALSZ13], it is possible to realize a large number of OT instances with amortized cost of only a few μs. Kolesnikov et al. [KKRT16] give a protocol for secure string equality test based on techniques for efficient OT. With their protocol, one can perform many private equality tests with amortized cost of 5 μs.

Overview of techniques. Suppose the sender holds a string $\boldsymbol{x} \in \{0,1\}^*$ and the receiver holds a pattern $\boldsymbol{p} \in \{0,1,\star\}^*$.

As a very simple warm up, consider the case that $|\boldsymbol{x}| = |\boldsymbol{p}| = 1$. The receiver will first encode its pattern $\boldsymbol{p} \in \{0,1,\star\}$ as a pair of bits $(\boldsymbol{p}^\star, \bar{\boldsymbol{p}})$ ("p-star & p-bar"), using the following encoding:

$\boldsymbol{p}$	$\boldsymbol{p}^\star$	$\bar{\boldsymbol{p}}$
$\star$	1	0
1	0	1
0	0	0

(6.1)

The significance of this encoding is the following:

$$\boldsymbol{x} \text{ matches pattern } \boldsymbol{p} \iff \boldsymbol{x} = \boldsymbol{p}^\star \cdot \boldsymbol{x} \oplus \bar{\boldsymbol{p}} \tag{6.2}$$

Indeed, if $\boldsymbol{p} = \star$, then $(\boldsymbol{p}^\star, \bar{\boldsymbol{p}}) = (1,0)$, so the RHS of (6.2) simplifies to $\boldsymbol{x}$ and the two sides equal (regardless of $\boldsymbol{x}$). On the other hand, if $\boldsymbol{p} \neq \star$, then $(\boldsymbol{p}^\star, \bar{\boldsymbol{p}}) = (0, \boldsymbol{p})$, so the RHS simplifies to $\boldsymbol{p}$ and the two sides equal if and only if $\boldsymbol{p} = \boldsymbol{x}$.

Our next trick is to blindly evaluate equation (6.2) using a single OT evaluation. The parties invoke an instance of 1-out-of-2 bit-OT, where the sender gives inputs $(\boldsymbol{k}, \boldsymbol{k} \oplus \boldsymbol{x})$, and the receiver gives input $\boldsymbol{p}^\star$. Here $\boldsymbol{k}$ is a random bit chosen by the sender. Note that the receiver's output from this OT is $\boldsymbol{k} \oplus \boldsymbol{p}^\star \cdot \boldsymbol{x}$.

Now, adding $\boldsymbol{k}$ to both sides of the equation in (6.2), we have that $\boldsymbol{x}$ matches pattern $\boldsymbol{p}$, if and only if $\boldsymbol{k} \oplus \boldsymbol{x} = (\boldsymbol{k} \oplus \boldsymbol{p}^\star \cdot \boldsymbol{x}) \oplus \bar{\boldsymbol{p}}$. Importantly, the LHS of this

equation is known to the sender, while the RHS is known to the receiver. At the same time, the random mask $\boldsymbol{k}$ hides all information about $\boldsymbol{x}$ from the receiver. We can summarize the above gadget as follows: using a single OT of bits, the sender and receiver each compute a bit which is *the same bit if and only if* $\boldsymbol{x}$ *matches pattern (possibly wildcard)* $\boldsymbol{p}$.

This technique can be easily extended to the case of WPM with $|\boldsymbol{x}| = |\boldsymbol{p}| = n$ by simply doing the above gadget n times, bit-by-bit. After doing so, each party will hold an n-bit string (without wildcards); these two strings will be equal if and only if $\boldsymbol{x}$ matches the pattern $\boldsymbol{p}$. An example is given in Figure 6.1 (we simply extend the notation $\oplus$ and $\cdot$ to bit-vectors). In short, we have reduced the problem of WPM with $|\boldsymbol{x}| = |\boldsymbol{p}|$ to the problem of secure (exact, no wildcards) equality test of strings. We complete the wildcard pattern matching by actually testing the equality of these strings, using the efficient protocol of Kolesnikov et al. [KKRT16].

The security of this protocol (in the semi-honest model) is easy to understand: the only new information is that the receiver obtains output $\boldsymbol{k} \oplus \boldsymbol{p}^\star \cdot \boldsymbol{x}$, which leaks no information about the sender's input $\boldsymbol{x}$ since $\boldsymbol{k}$ is uniform.

Now consider extending this approach to the general case of WPM with $|\boldsymbol{x}| > |\boldsymbol{p}|$. The idea is the natural one: for each $i \in \{1, |\boldsymbol{x}| - |\boldsymbol{p}| + 1\}$ simply perform the above approach on the substring $x[i \dots i + |\boldsymbol{p}| - 1]$ and $\boldsymbol{p}$. Unpacking the abstractions reveals room for optimizations, as follows. While the previous constructions were presented in terms of OT of *bits,* the OT of strings is significantly more efficient in practice. We observe that in each subprotocol, the receiver's OT choice bits are always the same $\boldsymbol{p}^\star$, allowing corresponding OT instances to be

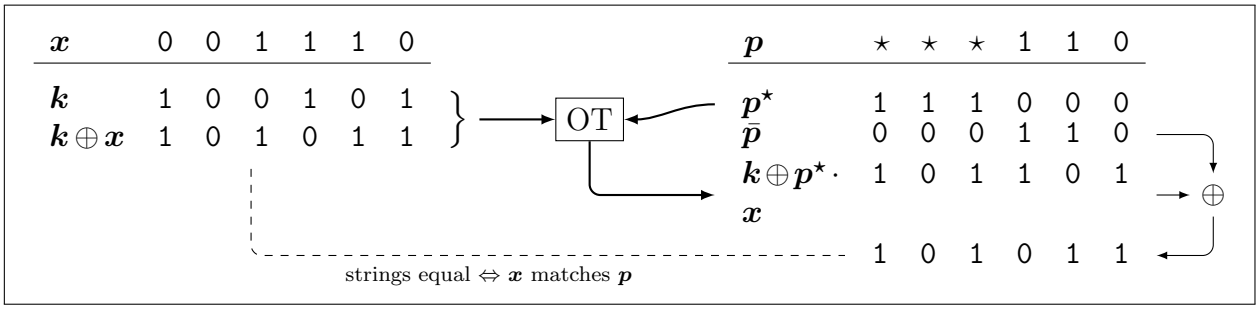


Figure 6.1: Illustration of the main idea behind our protocol: using oblivious transfer and private string equality test to perform private string equality with wildcards.

combined easily. Hence instead of $|\boldsymbol{p}|(|\boldsymbol{x}|-|\boldsymbol{p}|+1)$ instances of bit-OT, we can use $|\boldsymbol{p}|$ instances of string-OT, with strings of length $|\boldsymbol{x}|-|\boldsymbol{p}|+1$. This optimization actually reduces costs by a multiplicative factor of the security parameter. The details are given in Section 6.3.

In Section 6.3.1 we present additional optimizations and extensions, such as moving almost all of the cost to the offline, amortization and efficient handling of non-binary alphabets.

Efficiency. SWiM requires only $O(\kappa)$ public-key operations (all in the offline phase). In terms of communication, our protocol requires $O(mn)$ in the offline phase, but only $O(m+(\lambda+\kappa)n)$ in online phase. Here, κ, λ are the computational and statistical security parameters, respectively. As noted previously, all constants under the big-O are small, as we use fast optimized building blocks. We describe the performance of representative Secure Wildcard Pattern Matching protocols in Table 6.1.

We note that SWiM is efficient *concretely*. This is because we carefully optimize both computation and communication. Further, we use algorithmically- and implementation-optimized building blocks, namely the OT extension of [ALSZ13] and private equality test of [KKRT16]. In particular, the [KKRT16] equality test is *independent* of the length of the players' inputs.

This significantly improves over the state-of-the-art secure wildcard pattern matching protocol of [BEDM+12]. In Section 6.4, we report in detail on implementation and evaluation, and find that SWiM is a $2-499\times$ faster than 5PM, and continues to scale well on larger instances. 5PM considers WPM instances on DNA text of length up to 10^5 and pattern of length up to 10^3. These larger instances require only 1.96 seconds in our protocol, in comparison with 304.53 seconds with 1024-bit key and 978.94 seconds with 2048-bit key using [BEDM+12].

6.3 SWiM: the Main Construction

We present SWiM, our main construction for the WPM functionality in Figure 2.6. It closely follows and formalizes the high-level overview presented in Section 6.2. For readability, we present SWiM for binary alphabet $\Sigma = \{0, 1\}$. In Section 6.3.1 we show how to easily extend it to an arbitrary Σ. We first run OT extension with the chosen inputs, which will allow the receiver to compute $\boldsymbol{\alpha} = \boldsymbol{k} \oplus \boldsymbol{p}^\star \cdot \boldsymbol{x} \oplus \bar{\boldsymbol{p}}$. Recall, as discussed in Section 6.2, $\boldsymbol{x}$ matches $\boldsymbol{p}$, iff $\boldsymbol{\alpha}$ equals to $\boldsymbol{k} \oplus \boldsymbol{x}$ held by the sender. This equality is efficiently checked in bulk by calling instances of Private Equality Test defined in Figure 2.3, with the result delivered to the receiver and

output. The SWiM protocol is presented in Figure 6.2 and is proven secure against semi-honest adversaries.

Correctness. The main observation of OT-extension is that the receiver obtains output $\boldsymbol{q}_i$ such that:

$$\boldsymbol{q}_i = \boldsymbol{k}_i \oplus p_i^\star \cdot \boldsymbol{x}_{[i,i+n'-1]} = \begin{cases} \boldsymbol{k}_i, & \text{if } p_i^\star = 0 \\ \boldsymbol{k}_i \oplus \boldsymbol{x}_{[i,i+n'-1]}, & \text{if } p_i^\star = 1 \end{cases}$$

Therefore, the i-th row of U is equal to $\boldsymbol{u}_i = \boldsymbol{k}_i \oplus p_i^\star \cdot \boldsymbol{x}_{[i,i+n'-1]} \oplus C(\bar{p}_i)$. Let K denote the $m \times n'$ matrix such that the i-th row of K is the vector $\boldsymbol{k}_i$. When viewing the matrices U and T column-wise, we see that the receiver holds $\boldsymbol{u}^i = \boldsymbol{k}^i \oplus \boldsymbol{p}^\star \cdot \boldsymbol{x}_{[i,i+m-1]} \oplus \bar{\boldsymbol{p}}$ while the sender holds $\boldsymbol{t}^i = \boldsymbol{k}^i \oplus \boldsymbol{x}_{[i,i+m-1]}$. Following the high-level idea described Section 6.2, and specifically the pattern match test of Equation 6.2, it is clear that the pattern matches the text $\boldsymbol{x}$ at the i-th position if and only if $\boldsymbol{u}^i = \boldsymbol{t}^i$.

Theorem 15. The SWiM protocol in Figure 6.2 securely computes the WPM functionality (Figure 2.6) in semi-honest setting, given the ideal OT and $\mathcal{F}_{\mathsf{pm}}$ primitives.

Proof. The proof of security of our construction is based on the fact that the OT and $\mathcal{F}_{\mathsf{pm}}$ are secure.

Simulating $\mathcal{S}$. It is easy to argue that the view of the sender $\mathcal{S}$ can be perfectly simulated since the semi-honest $\mathcal{S}$ receives nothing from the protocol.

Simulating $\mathcal{R}$. The view of the receiver $\mathcal{R}$ consists of two kinds of messages:

(1) output of the form $\boldsymbol{q}_i$ from the OT primitive in Step 2c, which is equal to $\boldsymbol{k}_i \oplus p_i^\star \cdot \boldsymbol{x}_{[i,i+n'-1]}$ and hence information-theoretically hides $\boldsymbol{x}$; (2) outputs of $\mathcal{F}_{\mathsf{pm}}$ in step 4b, which correspond exactly to the WPM protocol output itself. Hence both can be perfectly simulated. □

Cost. Using OT extension, some initial "base OT" instances are required. These base OTs consist of $O(\kappa^2)$ communication and $O(\kappa)$ exponentiations. Thereafter, any number of OTs can be obtained with communication and computation proportional only to total size of parties' inputs. The computation consists of only *symmetric-key* operations. In our case, there are m OT instances, each on strings of length n', so $O(n'm)$ total communication and symmetric-key operations.

The $\mathcal{F}_{\mathsf{pm}}$ protocol of [KKRT16] has a statistical security parameter which we denote λ. Specifically, the protocol allows for a false positive (output 1 for input strings which are different) with probability $2^{-\lambda}$. The protocol also uses OT extension, but the base OTs can be shared/reused from the base OTs mentioned above. The amortized cost of an equality test is $448 + \lambda$ bits of communication (using typical parameters) and a constant number of symmetric-key operations.

6.3.1 Additions, optimizations

Online/offline phase. We briefly describe how the protocol can be modified so that most of the cost can be incurred in an offline phase, before the parties' inputs are known.

First, we can run all OTs in Step 2 of the protocol before the receiver's input $\boldsymbol{p}$ is known, by taking advantage of a well-known technique of Beaver [Bea95]. The following modifications are required: First, the receiver uses a random $\boldsymbol{\pi} \in \{0,1\}^m$ (rather than $\boldsymbol{p}^\star$) as its OT choice bits in Step 2 (note that $\bar{\boldsymbol{p}}$ is not used until Step 3). Later, upon learning $\boldsymbol{p}$, the receiver sends $\boldsymbol{\delta} = \boldsymbol{p} \oplus \boldsymbol{\pi}$ to the sender. The sender sets $\boldsymbol{k}'_i = \boldsymbol{k}_i \oplus \delta_i \cdot \boldsymbol{x}_{[i,i+n'-1]}$. It is easy to see that the receiver holds

$$\boldsymbol{q}_i = \boldsymbol{k}_i \oplus \pi_i \cdot \boldsymbol{x}_{[i,i+n'-1]} = \boldsymbol{k}_i \oplus (\delta_i \oplus p^\star_i) \cdot \boldsymbol{x}_{[i,i+n'-1]} = \boldsymbol{k}'_i \oplus p^\star_i \cdot \boldsymbol{x}_{[i,i+n'-1]}.$$

In other words, $\boldsymbol{k}'_i$ and $\boldsymbol{q}_i$ satisfy the appropriate condition, now with respect to the receiver's true input $\boldsymbol{p}$. The rest of the protocol continues as usual, with $\boldsymbol{k}'_i$ instead of $\boldsymbol{k}_i$.

There is also a standard Beaver technique for preprocessing OTs before the sender's OT input is known. Applying here naively would require the sender to send online correction strings of total length $O(|\boldsymbol{p}||\boldsymbol{x}|)$ since that is the combined length of all the sender's OT inputs.

Instead, we propose the following technique that is similar in spirit but takes advantage of the fact that the sender's OT inputs are derived from a single $\boldsymbol{x}$ value. The parties run step 1, but with the sender using a random $\boldsymbol{\chi} \in \{0,1\}^n$ instead of the true input $\boldsymbol{x}$ (which is not yet known). After the online phase described above, the sender will have $\boldsymbol{k}'_i$ strings and the receiver will have $\boldsymbol{q}_i = \boldsymbol{k}'_i \oplus p^\star_i \cdot \boldsymbol{\chi}_{[i,i+n'-1]}$. As the sender learns its input $\boldsymbol{x}$, it sends $\boldsymbol{\gamma} = \boldsymbol{x} \oplus \boldsymbol{\chi}$ to the receiver. The receiver

can compute

$$\begin{aligned}\boldsymbol{q}'_i \stackrel{\text{def}}{=} \boldsymbol{q}_i \oplus p_i^\star \cdot \boldsymbol{\gamma}_{[i,i+n'-1]} &= (\boldsymbol{k}'_i \oplus p_i^\star \cdot \boldsymbol{\chi}_{[i,i+n'-1]}) \oplus p_i^\star \cdot \boldsymbol{\gamma}_{[i,i+n'-1]} \\ &= \boldsymbol{k}'_i \oplus p_i^\star \cdot (\boldsymbol{\chi}_{[i,i+n'-1]} \oplus \boldsymbol{\gamma}_{[i,i+n'-1]}) \\ &= \boldsymbol{k}'_i \oplus p_i^\star \cdot \boldsymbol{x}_{[i,i+n'-1]}\end{aligned}$$

In other words, $\boldsymbol{k}'_i$ and $\boldsymbol{q}'_i$ satisfy the appropriate condition, now with respect to the sender's true input $\boldsymbol{x}$. The protocol can proceed, using $\boldsymbol{q}'_i$ instead of $\boldsymbol{q}_i$.

By having precomputation, we are able to shift the bulk of the $O(nm)$ communication to the offline phase. In the online phase, each party only sends a "correction string" whose length is proportional to its input size, followed by the equality tests. Similarly to the standard Beaver's technique, it is easy to see that the resulting protocol is secure, namely that the separation of the offline and online phases can be simulated.

Amortization. In certain multiple-execution scenarios, the cost of our protocol can be further significantly reduced by reusing the OT/PM outputs.

First, notice that in SWiM (Figure 6.2), the OT step is independent of the non-wildcard characters of the pattern string (i.e., independent of $\bar{\boldsymbol{p}}$). Therefore, if the *positions of wildcards* in the receiver's pattern (i.e., $\boldsymbol{p}^\star$) are the same across several executions, OT in subsequent executions can be implemented as length extension of the OT in the first execution. Further, if additionally the sender's text is the same across the executions (and the only variation is the non-$\star$ pattern), then only

the equality tests need to be run in the subsequent executions.

Further, in the PM protocol of [KKRT16], the receiver can check his input for equality against a polynomial number sender's inputs at the cost λ per check (vs $4\kappa + \lambda$ for full KKRT PM). Indeed, on the KKRT BaRK-OPRF output $(R, S) \leftarrow (F_k(x), k)$, KKRT sender $\mathcal{S}$ can send to receiver $\mathcal{R}$ a set of $\{F_k(y_i)\}$, and R will determine $x = y_i \iff F_k(x) = F_k(y_i)$.

To use this in the amortization, we let the WPM sender play the role of PM's receiver. We note that this amortization will reveal whether the WPM receiver has used the same pattern in different instances. Additionally, PM receiver learns the comparison output, and so will the WPM sender. Both restrictions may be acceptable in certain scenarios.

Non-binary alphabets. The protocol extends naturally to alphabets Σ beyond $\Sigma = \{0, 1\}$. Without loss of generality let $\Sigma = \mathbb{Z}_b$ for some b. The receiver holds a pattern $\boldsymbol{p} \in (\Sigma \cup \{\star\})^m$ and will encode the pattern into $\boldsymbol{p}^\star \in \{0,1\}^m$ and $\bar{\boldsymbol{p}} \in \Sigma^m$, as follows:

$\boldsymbol{p}_i$	$\boldsymbol{p}_i^\star$	$\bar{\boldsymbol{p}}_i$
$\star$	1	0
$a \neq \star$	0	a

Consider the corresponding amendment to SWiM (Figure 6.2), where the parties hold strings of length m and n, both over the alphabet Σ. The parties still perform m 1-out-of-2 OT, using $\boldsymbol{p}^\star$ as the receiver's choice bits. All other vectors ($\boldsymbol{k}$, $\boldsymbol{q}$, etc) become vectors over Σ, and the $\oplus$ operation is replaced by component-wise

addition mod $|\Sigma|$. Note that the "$\cdot$" operation in the protocol is only used between a *binary* vector $\boldsymbol{p}^\star$ and a Σ-vector, so its meaning can still be taken as component-wise multiplication. Finally, the KKRT PM can be naturally amended to support equality tests of non-binary strings, e.g. by translating the strings into binary.

6.4 SWiM Implementation and Performance

Our SWiM implementation uses code from [KKRT16, Rin, WMK16]. All running times are reported as the average over 10 trials. Our complete implementation is available on `https://github.com/osu-crypto/PatternMatching`.

6.4.1 Experimental Performance: Comparison with Prior Work

We compare our prototype to the state-of-art WPM protocols [BEDM+12, YSK+14]. While the implementations [BEDM+12, YSK+14] are not publicly available, [BEDM+12] reports experimental numbers in the semi-honest model. Further, as we discussed in Section 6.1.1, [YSK+14] numbers can be indirectly estimated to be around $2 - 2.5\times$ faster than 5PM. We give detailed comparisons to 5PM protocol [BEDM+12]; comparison to other works can be appropriately derived.

Runtime Comparison. For the most direct comparison, we matched the test system's computational performance to that of [BEDM+12], as reported in their Table 13. Since 5PM [BEDM+12] experiments were performed on Intel dual quad-

core 2.93GHz Linux machine with 8GB RAM, we evaluate our protocol on a virtual Linux machine with 8GB RAM and 2 cores (the host machine is Intel Core i7 2.60GHz with 12GB RAM). Table 6.2 presents the running time of our protocol compared with 5PM [BEDM+12]. For our protocol, we report both the total running time and the online time. We use $\lceil \log(\Sigma) \rceil$ bits to encode the text and pattern alphabet into binary alphabet.

When comparing the two protocols, we find that the total running time of SWiM is significantly less than that of the prior works, requiring 1.96 seconds to perform a wildcard pattern matching with 4-symbol alphabet for text size $n = 10^5$ and pattern size $m = 10^3$. This is a 155× improvement in running time compared to 5PM [BEDM+12] which used 1024-bit key length. When considering 5PM [BEDM+12] with 2048-bit key length (which better corresponds to our security level), our improvement is 499×.

SWiM is optimized for the typical use case, where the length of the text is greater than that of the pattern. If this doesn't hold (indeed, an unusual setting for the motivating examples we consider), our performance improvement is moderate. For instance when $m = n = 10^3$, our protocol requires 0.61 seconds. Using the same parameters, the protocol of [BEDM+12] results in an execution time of 1.39 seconds. The moderate 2× improvement is due to the constant-cost overheads of OT extension and PM, which do not pay off without amortization in a larger execution. Even in these cases, our protocol achieves great improvement in the online phase (e.g., running in just 3ms for $m = n = 10^3$).

Bandwidth Comparison. We calculate the bandwidth requirements of our protocol on the range of the length text $n \in \{2^{16}, 2^{18}, 2^{20}, 2^{22}\}$ and the length pattern $m \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$, for the binary alphabet. For comparison, we calculate the communication cost of 5PM [BEDM+12], for the same parameters. 5PM bandwidth requirements is independent on the length of pattern, and is roughly $(n+2)\kappa$ bits. 5PM protocol relies on public-key operations, and needs 1024-2048-bit key lengths.

Table 6.4 reports the communication overhead of the protocols. Our protocol requires less communication for smaller pattern sizes. Concretely, for $n = 2^{22}$ and $m = 2^8$, our protocol requires 392.1 MB of communication, a 1.37 to 2.7× improvement compared to 5PM [BEDM+12]. Increasing the pattern length to $m = 2^{12}$ the communication cost of 5PM protocol (at a great performance penalty!) becomes preferable to ours, since their bandwidth is independent of the length of pattern. Note, the bulk of the communication cost in our protocol is OT extension in the offline phase.

We note that Table 6.4 does not show off SWiM algorithmic improvement for non-binary alphabet, which reduces the number of OT calls. For larger Σ, we (but not other approaches, to our knowledge) get factor $\approx \log |\Sigma|$ bandwidth reduction in the *offline* phase over the simple mapping of Σ to a binary alphabet.

6.4.2 SWiM performance at scale: experiments and discussion

To understand the scalability of SWiM, we evaluate it on the range of the text/pattern lengths $n \in \{2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}\}$, $m \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$, for the binary alphabet. We report SWiM detailed performance results in Table 6.3, showing total running time and online time in both LAN and WAN settings.

This set of experiments was ran on a larger machine (a single server with 2x 36-core Intel Xeon 2.30GHz CPU and 256GB of RAM), whose resources were carefully limited by us to provide a good understanding of the performance. Specifically, we ran each party *single threaded*, both on the same machine, communicating via `localhost` network. We simulated a network connection using the Linux `tc` command. We configured LAN setting with 0.02ms round-trip latency, 10 Gbps network bandwidth, and WAN setting with a simulated 40ms round-trip latency, 400 Mbps network bandwidth.

The step of forming the matrices in SWiM is relatively costly. We push it into the preprocessing phase, which will include creating OT matrices and performing the matrix transposition. Our experiments show that the offline phase takes $60-90\%$ of the total running time. For instance, with text size $n = 2^{22}$ and pattern size $m = 2^{14}$ our overall running time is 60.10 seconds with an offline phase of 53.64 seconds, a 89% of the overall cost.

We find that SWiM scales well in the experiments. For text size $n = 2^{16}$ and pattern size $m = 2^8$, our protocol takes only 0.21 seconds in which 0.04 seconds is for online time. When increasing the lengths to $n = 2^{24}$ and $m = 2^{12}$, we see that

our protocol requires roughly 52 seconds in total.

When evaluating our implementation in the WAN setting, we still have a fast online phase due to the fact that OTs can be precomputed in the offline phase. For $n = 2^{24}$ and $m = 2^{12}$, we obtain an overall running time of 363.06 seconds and an online time of 50.15 seconds which contains only 13% of the total cost. For the small text and pattern, the protocol requires only a few seconds. With $n = 2^{16}$ and $m = 2^8$, our protocol takes an overall running time of 1.04 seconds with the online phase requiring 0.4 seconds.

PARAMETERS:

1. Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$
2. A length n of text, a length m of pattern. Define $n' = n - m + 1$
3. A repetition encoding $C : \{0,1\} \to \{0,1\}^{n'}$ defined by $C(a) = a^{n'}$ for $a \in \{0,1\}$.
4. Ideal OT and $\mathcal{F}_{\mathsf{pm}}$ primitives.

INPUT OF $\mathcal{S}$: a text $\boldsymbol{x} \in \{0,1\}^n$

INPUT OF $\mathcal{R}$: a pattern $\boldsymbol{p} \in \{0,1,\star\}^m$ encoded into $\bar{\boldsymbol{p}}, \boldsymbol{p}^\star \in \{0,1\}^m$, as described in Section 6.2.

PROTOCOL:

1. **[Random Keys]** $\mathcal{S}$ chooses $\{\boldsymbol{k}_i\}_{i\in[m]} \leftarrow \{0,1\}^{n'}$ at random
2. **[OT]** For each $i \in [m]$, $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathsf{OT}_{n'}$-functionality
   (a) $\mathcal{R}$ acts as *receiver* with a input-bit $p_i^\star$.
   (b) $\mathcal{S}$ acts as *sender* with a ordered pair input $(\boldsymbol{k}_i, \boldsymbol{k}_i \oplus \boldsymbol{x}_{[i,i+n'-1]})$
   (c) $\mathcal{R}$ receives output $\boldsymbol{q}_i$
3. **[Matrix Form]**
   (a) $\mathcal{S}$ forms $m \times n'$ matrix T such that the i-th row of T is the vector $\boldsymbol{t}_i = \boldsymbol{k}_i \oplus \boldsymbol{x}_{[i,i+n'-1]}$
   (b) $\mathcal{R}$ forms $m \times n'$ matrix U such that the i-th row of U is the vector $\boldsymbol{u}_i = \boldsymbol{q}_i \oplus C(\bar{p}_i)$.
4. **[PEQ]**
   (a) For each $i \in [n']$, $\mathcal{S}$ and $\mathcal{R}$ invoke the $\mathcal{F}_{\mathsf{pm}}$-functionality:
      - $\mathcal{S}$ acts as *sender* with input $\boldsymbol{t}^i$ as the i-th column of T
      - $\mathcal{R}$ acts as *receiver* with input $\boldsymbol{u}^i$ as the i-th column of U
   (b) $\mathcal{R}$ outputs $\{i \in [n'] \mid i\text{th instance of } \mathcal{F}_{\mathsf{pm}} \text{ outputs } 1\}$

Figure 6.2: SWiM: **S**ecure **Wi**ldcard Pattern **M**atching Protocol for $\Sigma = \{0,1\}$.

Protocol	Bit key length	Pattern length m	Text length n 10^3	10^4	10^5
5PM	1024	10	0.42	4.08	40.43
		10^2	0.67	6.81	64.76
		10^3	0.39	29.15	304.53
	2048	10	1.50	14.18	140.52
		10^2	2.27	22.37	216.27
		10^3	1.39	92.29	978.94
SWiM	128	10	**0.29 (0.006)**	**0.36 (0.03)**	**0.76 (0.32)**
		10^2	**0.37 (0.005)**	**0.62 (0.09)**	**1.82 (0.49)**
		10^3	**0.61 (0.003)**	**0.73 (0.04)**	**1.96 (0.39)**

Table 6.2: 5PM vs SWiM. Comparison of 5PM and SWiM of the total runtime (in seconds) for wildcard pattern matching of length n, the pattern of length m, and the alphabets of sizes 4 (DNA). In SWiM, the online time is presented in parenthesis. Best results marked in bold. SWiM experiment ran on Intel Core i7 2.60GHz with 8GB RAM. 5PM timings reported on comparable hardware.

Setting	Pattern length m	Text length n 2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
LAN	2^8	**0.21 (0.04)**	0.40 (0.15)	0.94 (0.48)	4.07 (2.78)	16.11 (11.38)
	2^{10}	0.24 (0.03)	0.48 (0.12)	1.41 (0.57)	5.21 (2.38)	20.61 (10.00)
	2^{12}	0.37 (0.03)	0.97 (0.17)	3.40 (0.78)	12.92 (3.34)	**51.88 (14.44)**
	2^{14}	1.02 (0.07)	3.91 (0.37)	15.14 (1.66)	**60.10 (6.46)**	246.24(43.51)
WAN	2^8	**1.04 (0.40)**	1.90 (1.02)	5.10 (3.10)	17.84 (12.04)	70.45 (48.43)
	2^{10}	1.28 (0.40)	2.81 (0.95)	8.62 (3.04)	31.29 (12.00)	127.92 (48.08)
	2^{12}	2.28 (0.36)	6.46 (0.96)	21.61 (3.17)	84.52 (12.48)	**363.06 (50.15)**
	2^{14}	6.16 (0.34)	22.24 (1.07)	85.98 (3.87)	318.23 (15.45)	1,382.03 (65.86)

Table 6.3: Total running time and online time (in parenthesis) in second of SWiM for the text of length n, the pattern of length m, binary alphabet. The results mentioned in the discussion is marked in bold. Experiment ran sender and receiver *single-threaded* on 2x 36-core Intel Xeon 2.30GHz CPU and 256GB of RAM.

Protocol	Bit key length	Pattern length m	Text length n 2^{16}	2^{18}	2^{20}	2^{22}
5PM	1024	$\{2^8, 2^{10}, 2^{12}, 2^{14}\}$	8.4	33.5	134.2	536.9
	2048	$\{2^8, 2^{10}, 2^{12}, 2^{14}\}$	16.8	67.1	268.4	1073.7
SWiM	128	2^8	**7.6(3.9)**	**25.9(16.1)**	**99.2(64.1)**	**392.1(256.4)**
		2^{10}	**13.7(3.9)**	**50.9(15.9)**	**199.7(64.1)**	**794.6(256.3)**
		2^{12}	36.7(3.7)	149.5(15.8)	600.2(63.8)	2403.1(256.1)
		2^{14}	105.2(2.9)	519.9(15.1)	2178.6(63.1)	8813.3(255.4)

Table 6.4: Communication (in MB) for wildcard pattern matching of text length n, pattern length m, binary alphabet. In SWiM, the online communication cost is presented in parenthesis. Compared to 5PM, best results marked in bold.

Bibliography

[ACT11] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (if) size matters: Size-hiding private set intersection. In *PKC*, pages 156–173, 2011.

[ADT11] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (If) size matters: Size-hiding private set intersection. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 156–173. Springer, Heidelberg, March 2011.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 673–701. Springer, Heidelberg, April 2015.

[ANS10] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation. In *51st FOCS*, pages 787–796. IEEE Computer Society Press, October 2010.

[ATD15] Aydin Abadi, Sotirios Terzis, and Changyu Dong. O-PSI: delegated private set intersection on outsourced datasets. In *ICT Systems Security and Privacy Protection*, pages 3–17. Springer, 2015.

[AWY15] Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *26th SODA*, pages 218–230. ACM-SIAM, January 2015.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, Heidelberg, August 1995.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th ACM STOC*, pages 479–488. ACM Press, May 1996.

[BEDM+12] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. SCN'12, 2012.

[Ber06] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *PKC*, pages 207–228, 2006.

[BGVV14] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory of Computing Systems*, 2014.

[BI14] Carl Barton and Costas S. Iliopoulos. On the average-case complexity of pattern matching with wildcards. *CoRR*, abs/1407.0950, 2014.

[BPSW07] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *ACM CCS*, pages 498–507, 2007.

[BS05] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 236–252. Springer, Heidelberg, December 2005.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security 2010*, pages 223–240. USENIX Association, August 2010.

[BST01] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 2001.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CC07] Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53 – 54, 2007.

[ÇCL+17] Gizem S. Çetin, Hao Chen, Kim Laine, Kristin Lauter, Peter Rindal, and Yuhou Xia. Private queries on encrypted genomic data. *BMC Medical Genomics*, 2017.

[CCS18] Andrea Cerulli, Emiliano De Cristofaro, and Claudio Soriente. Nothing refreshes like a repsi: Reactive private set intersection. In *Applied Cryptography and Network Security (ACNS)*. Springer Berlin Heidelberg, 2018.

[CDJ16] Chongwon Cho, Dana Dachman-Soled, and Stanislaw Jarecki. Efficient concurrent covert computation of string equality and set intersection. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 164–179. Springer, Heidelberg, February / March 2016.

[CEPR09] Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. From coding theory to efficient pattern matching. In Claire Mathieu, editor, *20th SODA*, pages 778–784. ACM-SIAM, January 2009.

[CGT12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *CANS 2012*, pages 218–231, 2012.

[CH02] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *34th ACM STOC*, pages 592–601. ACM Press, May 2002.

[CJS12] Jung Hee Cheon, Stanislaw Jarecki, and Jae Hong Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Transactions*, 95-A(8):1366–1378, 2012.

[CKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT*, pages 213–231, 2010.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1243–1255. ACM Press, October / November 2017.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 464–482. Springer, Heidelberg, September 2018.

[CPPT14] Ran Canetti, Omer Paneth, Dimitrios Papadopoulos, and Nikos Triandopoulos. Verifiable set operations over outsourced databases. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 113–130. Springer, Heidelberg, March 2014.

[CRS03] Artur Czumaj, Chris Riley, and Christian Scheideler. Perfectly balanced allocation. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, 2003.

[CS15] Melissa Chase and Emily Shen. Substring-searchable symmetric encryption. *PoPETs*, 2015.

[CT12] Emiliano De Cristofaro and Gene Tsudik. Experimenting with fast private set intersection. In *TRUST 2012*, pages 55–73, 2012.

[CWZ+06] Gong Chen, Xindong Wu, Xingquan Zhu, Abdullah N. Arslan, and Yu He. Efficient string matching with wildcards and length constraints. *Knowledge and Information Systems*, 10(4):399–419, Nov 2006.

[DC17] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In Josef Pieprzyk and Suriadi Suriadi, editors, *ACISP 17, Part II*, volume 10343 of *LNCS*, pages 261–278. Springer, Heidelberg, July 2017.

[DCFT13] Emiliano De Cristofaro, Sky Faber, and Gene Tsudik. Secure genomic testing with size- and position-hiding private substring matching. WPES '13, 2013.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *ACM CCS 2013*, pages 789–800, 2013.

[DF13] K. El Defrawy and S. Faber. Blindfolded data search via secure pattern matching. *Computer*, 46(12):68–75, Dec 2013.

[DKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 213–231. Springer, Heidelberg, December 2010.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018(4):159–178, 2018.

[DT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 143–159. Springer, Heidelberg, January 2010.

[FHV13] Sebastian Faust, Carmit Hazay, and Daniele Venturi. Outsourced pattern matching. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP 2013, Part II*, volume 7966 of *LNCS*, pages 545–556. Springer, Heidelberg, July 2013.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[FNO18] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private set intersection with linear communication from general assumptions. ePrint Archive, Report 2018/238, 2018.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 1–19. Springer, Heidelberg, May 2004.

[FNW96] Ronald Fagin, Moni Naor, and Peter Winkler. Comparing information without leaking it. *Commun. ACM*, 1996.

[Fri07] Keith B. Frikken. Privacy-preserving set union. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 237–252. Springer, Heidelberg, June 2007.

[Fri09] Keith B. Frikken. *Practical Private DNA String Searching and Matching through Efficient Oblivious Automata Evaluation*. 2009.

[GBZT] Pierrick Gaudry, Richard Brent, Paul Zimmermann, and Emmanuel Thomé. https://gforge.inria.fr/projects/gf2x/.

[GHS10] Rosario Gennaro, Carmit Hazay, and Jeffrey S. Sorensen. Text search protocols with simulation based security. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 332–350. Springer, Heidelberg, May 2010.

[GLMY16] Adam Groce, Alex Ledger, Alex J. Malozemoff, and Arkady Yerukhimovich. CompGC: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive, Report 2016/458, 2016. http://eprint.iacr.org/2016/458.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[GN19] Satrajit Ghosh and Tobias Nilges. An algebraic approach to maliciously secure private set intersection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 154–185. Springer, Heidelberg, May 2019.

[GNN17] Satrajit Ghosh, Jesper Buus Nielsen, and Tobias Nilges. Maliciously secure oblivious linear function evaluation with constant overhead. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017,*

Part I, volume 10624 of *LNCS*, pages 629–659. Springer, Heidelberg, December 2017.

[Gol04] Oded Goldreich. *Foundations of Cryptography, Volume 2: Basic Applications*. Cambridge University Press, 2004.

[Gon81] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, April 1981.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security 2011*. USENIX Association, August 2011.

[HFH99] Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *EC*, pages 78–86, 1999.

[HHD11] H. Hoffmann, M. D. Howard, and M. J. Daily. Fast pattern matching with time-delay neural networks. In *The 2011 International Joint Conference on Neural Networks*, pages 2424–2429, July 2011.

[HL08] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 155–175. Springer, Heidelberg, March 2008.

[HLS+16] K. Hogan, N. Luther, N. Schear, E. Shen, D. Stott, S. Yakoubov, and A. Yerukhimovich. Secure multiparty computation for cooperative cyber risk assessment. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 75–76, Nov 2016.

[HMFS17] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *ACM CCS*, pages 1389–1406, 2017.

[HN12] Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. *Journal of Cryptology*, 25(3):383–433, July 2012.

[HS13] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *ASIA CCS*, pages 437–446, 2013.

[HT10] Carmit Hazay and Tomas Toft. Computationally secure pattern matching in the presence of malicious adversaries. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 195–212. Springer, Heidelberg, December 2010.

[HT14] Carmit Hazay and Tomas Toft. Computationally secure pattern matching in the presence of malicious adversaries. *Journal of Cryptology*, 27(2):358–395, April 2014.

[HV17] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 175–203. Springer, Heidelberg, March 2017.

[IKN+17] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. *ePrint Archive 2017/738*, 2017.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

[JKS08] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy*, pages 216–230. IEEE Computer Society Press, May 2008.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *20th ACM STOC*, pages 20–31. ACM Press, May 1988.

[KK12] Vladimir Kolesnikov and Ranjit Kumaresan. Improved secure two-party computation via information-theoretic garbled circuits. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 205–221. Springer, Heidelberg, September 2012.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

[KM08] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, September 2008.

[KM10] Jonathan Katz and Lior Malka. Secure text processing with applications to private DNA matching. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010*, pages 485–492. ACM Press, October 2010.

[KMP+17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1257–1272. ACM Press, October / November 2017.

[KMR11] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. http://eprint.iacr.org/2011/272.

[KMW08] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *ESA 2008*, pages 611–622, 2008.

[KNR+17] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 3–20. ACM Press, October / November 2017.

[Kol05] Vladimir Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 136–155. Springer, Heidelberg, December 2005.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KRT18] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. SWiM: Secure wildcard pattern matching from OT extension. In Sarah Meiklejohn and Kazue Sako, editors, *FC 2018*, volume 10957 of *LNCS*, pages 222–240. Springer, Heidelberg, February / March 2018.

[KRTW19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 636–666. Springer, Heidelberg, December 2019.

[KS05] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257. Springer, Heidelberg, August 2005.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgαrd, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[Lam16] Mikkel Lambæk. Breaking and fixing private set intersection protocols. Master's thesis, Aarhus University, 2016.

[Lip03] Helger Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In Chi-Sung Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 416–433. Springer, Heidelberg, November / December 2003.

[LV04] Arjen K. Lenstra and Tim Voss. Information security risk assessment, aggregation, and mitigation. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 391–401. Springer, Heidelberg, July 2004.

[LW07] Ronghua Li and Chuankun Wu. *An Unconditionally Secure Protocol for Multi-Party Set Intersection*, pages 226–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[MB72] R. Moenck and Allan Borodin. Fast modular transforms via division. In *Switching and Automata Theory*, pages 90–96, 1972.

[Mea86] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *IEEE S&P*, 1986.

[MN15] Atsuko Miyaji and Shohei Nishida. A scalable multiparty private set intersection. In *Network and System Security*, pages 376–385. Springer, 2015.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *USENIX 2004*, pages 20–20, 2004.

[MNSS12] Payman Mohassel, Salman Niksefat, Seyed Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious DFA evaluation and applications. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 398–415. Springer, Heidelberg, February / March 2012.

[MPP10] Mark Manulis, Benny Pinkas, and Bertram Poettering. Privacy-preserving group discovery with linear complexity. In *ACNS 2010*, pages 420–437, 2010.

[Nie07] Jesper Buus Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. ePrint Archive, Report 2007/215, 2007.

[NN10] Kedar Namjoshi and Girija Narlikar. Robust and fast pattern matching for intrusion detection. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 740–748, Piscataway, NJ, USA, 2010. IEEE Press.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, 1999.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In S. Rao Kosaraju, editor, *12th SODA*, pages 448–457. ACM-SIAM, January 2001.

[Ode09] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[OPJM10] Margarita Osadchy, Benny Pinkas, Ayman Jarrous, and Boaz Moskovich. SCiFI - a system for secure face identification. In *2010 IEEE Symposium on Security and Privacy*, pages 239–254. IEEE Computer Society Press, May 2010.

[PCR08] Arpita Patra, Ashish Choudhary, and C. Pandu Rangan. Unconditionally secure multiparty set intersection re-visited. *IACR Cryptology ePrint Archive*, 2008:462, 2008.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431. Springer, Heidelberg, August 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Psi from paxos: Fast, malicious private set intersection. In *EUROCRYPT*, 2020.

[PSS17] Arpita Patra, Pratik Sarkar, and Ajith Suresh. Fast actively secure OT extension for short secrets. In *NDSS*, 2017.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 797–812. USENIX Association, August 2014.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21, 2018.

[RA17] Amanda C. Davi Resende and Diego F. Aranha. Unbalanced approximate private set intersection. *ePrint Archive 2017/677*, 2017.

[Rin] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[RR17a] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 235–259. Springer, Heidelberg, April / May 2017.

[RR17b] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In Bhavani M. Thuraisingham, David

Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1229–1242. ACM Press, October / November 2017.

[RT20] Mike Rosulek and Ni Trieu. Revisiting & improving diffie-hellman-based private set intersection. In *submitted to CRYPTO*, 2020.

[SEK03] Peter Sanders, Sebastian Egner, and Jan Korst. Fast concurrent access to parallel disks. *Algorithmica*, 35(1):21–55, 2003.

[Sha80] Adi Shamir. On the power of commutativity in cryptography. In *Automata, Languages and Programming*, 1980.

[Sho] Victor Shoup. NTL: a library for doing number theory. http://www.shoup.net/ntl.

[SK17] Tushar Kanti Saha and Takeshi Koshiba. *An Enhancement of Privacy-Preserving Wildcards Pattern Matching*, pages 145–160. Springer International Publishing, Cham, 2017.

[SOF10] Pattern matching with don't cares and few errors. *Journal of Computer and System Sciences*, 2010.

[SS08] Yingpeng Sang and Hong Shen. Privacy preserving set intersection based on bilinear groups. In *Proceedings of the Thirty-first Australasian Conference on Computer Science - Volume 74*, ACSC '08, pages 47–54, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[SSSS15] Riku Saikkonen, Seppo Sippu, and Eljas Soisalon-Soininen. *Experimental Analysis of an Online Dictionary Matching Algorithm for Regular Expressions with Gaps*. 2015.

[Tha11] Chris Thachuk. *Succincter Text Indexing with Wildcards*. 2011.

[TKC07] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient dna searching through oblivious automata. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 519–528. ACM Press, October 2007.

[TLP+17] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. In *ASIA CCS*, pages 31–44, 2017.

[Ver11] Damien Vergnaud. Efficient and secure generalized pattern matching via fast fourier transform. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT 11*, volume 6737 of *LNCS*, pages 41–58. Springer, Heidelberg, July 2011.

[WJW+15] D. Wang, X. Jia, C. Wang, K. Yang, S. Fu, and M. Xu. Generalized pattern matching string search on encrypted data in cloud systems. In *INFOCOM*, 2015.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[WZX17] Xiaochao Wei, Minghao Zhao, and Qiuliang Xu. Efficient and secure outsourced approximate pattern matching protocol. *Soft Computing*, Mar 2017.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YSK+13] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. CCSW '13, 2013.

[YSK+14] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. *Privacy-Preserving Wildcards Pattern Matching Using Symmetric Somewhat Homomorphic Encryption*. 2014.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.