A Report on Substituting Cuckoo Hashing with Common Variants in OT-Based PSI Protocol

Huiyang He
School of Cyber Science and Technology
University of Science and Technology
Anhui, China
hhe@mail.ustc.edu.cn

line 1: Chenglin Li
School of Cyber Science and Technology
University of Science and Technology
Anhui, China
li1116@mail.ustc.edu.cn

***Abstract*—PSI problems has been extensively studied for several decades and already has numerous application cases in real life. While OT-based PSI appears in relatively later period compared to other PSI schemes, it was proved to have the most balanced performance under different application scenes and its advantage on huge set with large elements is prominent. In the state-of-art OT-Based PSI protocol, Cuckoo hashing is a key component which greatly helps to reduce the number of OTs. In the past years, many variants of cuckoo hashing have been constructed and utilized in different applications. As a result, we are curious about how those variants perform in the OT-based protocol. We chose two main variants of cuckoo hashing and first tested their occupancy rate and then substitute them into the state-of-art OT-based PSI protocol and planned further optimization on parallelization and memory usage based on those variants. The results showed very limited improvement on performance, but we got better understand of the protocol and built a better road map to improve the performance of OT-based PSI.**

Keywords—Cuckoo Hashing, PSI Protocol, Hashing Variants, OT

I. INTRODUCTION

Our Work involves Oblivious Transfer (OT), Oblivious Pseudorandom Function (OPRF), Private Set Intersection (PSI), and Cuckoo Hashing, so this report starts with brief overview of those four primitives.

A. PSI

As one of the most active research fields of Secure Multiparty Computation (MPC), PSI has been subjected to intense improvements ever since it is introduced. This is the era of big data, an efficient privacy-preserving computation protocol is very appealing not only to those people who cares about their privacy, but also to organizations whose business heavily relies on data. JOIN is a very common operation that is needed in lots of scenes, but sometimes, it is irrational for two participants to provide all their data to find the intersection. Consequently, PSI was introduced, and it enables two or more party to perform JOIN operations on their data set in a privacy-preserving manner. Without doubt, this greatly eases the privacy problem for many big data applications.

PARAMETERS:
Two parties: Sender S and Receiver R.
S has set X.
R has set Y.
FUNCTIONALITY:
S outputs $\perp$.
R outputs $X \cap Y$.

Fig. 1. Basic Functionality of PSI

One of the most usual applications of PSI is advertisement conversion rate measurement. Usually, the first step to calculate Advertisement conversion rate is to identify the people who have seen the advertisement, also have completed a transaction with the merchant. This problem can be generalized as that the merchant holds a list of customers who have done a transaction and the advertiser holds a list of users who have seen the advertisement. Then they want to find the intersection of those two lists. This, of course, can be done by one of them sending its whole list to the other. However, in real worlds, those data are valuable property and neither of them would be willing to provide the other with its own data. However, if those two parties use a safe PSI protocol, then the computation is secured, and they can learn nothing more than the intersection. Thus, the privacy problem is solved, and they both protect their legal property through utilizing PSI.

Among different approaches, although OT-based PSI appears relatively later, it was proved to have the most balanced performance under different application scenes, especially on large data sets with elements of more representation bits. In (Dong, Chen, & Wen), Dong etc. proposed an OT-based PSI protocol by utilizing Blossom Filter (BF). Later in 2014, Pinkas etc. constructed a fancy PSI protocol based on OT extension (Pinkas, Schneider, & Zohner, Faster Private Set Intersection Based on OT Extension, 2014). Then in 2015, serval optimization was applied to the PSZ protocol and thus an even more efficient protocol (PSSZ protocol) was constructed. Finally, in 2017, (Kolesnikov, et al.) further boosted this OT extension-based protocol. They showed their efficient Batch-Related-key-OPRF (BaRK-OPRF) and made use of this OPRF with PSSZ protocol to build their BaRK-OPRF PSI. Our work is mainly based on BaRK-OPRF PSI, and therefore a more detailed description of BaRK-OPRF PSI and explanation of what role cuckoo hashing plays in the protocol will be given in §2.

B. OT

OT is a basic primitive of MPC. OT is a kind of tool which enables a sender to transfer one of potentially many messages to a receiver but remains oblivious as to what piece of information has been transferred and the receiver cannot know messages other than its choice. Naor-Pinkas OT proposed in 2001 is one of the most used OT protocol. They proved their protocol is secure under semi-honest setting based on the decisional discrete logarithm problem [1].

While it has been showed that OT can only be built on public key scheme, OT extension has been proposed to make OT more practical by building a large number of OTs from small number of base OTs. Ishai, Kilian, Nissim, Petrank continued the thought of Beaver, and constructed their IKNP OT extension [2] which is truly a treasure that makes most MPC become practical than they have ever been. Let OT_l^m means m instances of OT, each of which transfers l bits. IKNP OT extension realize $\binom{2}{1} OT_l^m \rightarrow \binom{2}{1} OT_m^k$, where $k \ll$

m. OT has been subjected to very little implements after [2] until when Kolesnikov and Kumaresan suggested their KK OT extion in 2013 [3]. They creatively showed their fancy way of interpreting the IKNP OT extension from the view of coding theory and their construction realize $\binom{n}{1} OT_l^m \rightarrow \binom{n}{1} OT_m^k$. It is also mentioned by them that it can be further reduced from $\binom{2}{1} OT_m^k$ to $\binom{2}{1} OT_k^k$, using random OT (ROT). ROT is a very important optimization technic for OT. What is difference in ROT is that rather than transfer one of the secrets to receiver, the sender now holds no messages at the beginning of the protocol, but the receiver still has a choose bit. They use a PRG and an agreed key to generate messages, such that the sender holds two messages, and the receiver can generate exactly one of the messages based on his choose bit. It is worth noticing that ROT is also a very powerful optimization technic for the KK OT Extension and OT-based PSI protocols which will be discussed below, but the way ROT is built into those protocols are not the same and might take some time to understand.

Preliminaries:

- *The protocol operates over a group Z_q of prime order. g is the generator.*
- *The computational Diffie-Hellman Assumption holds.*
- *The protocol uses a function H which is assumed to be a random oracle.*

Initialization:

The sender chooses a random element $C \in Z_q$ and publishes it. (It is only important that the receiver will not know the discrete logarithm of C to the base g)

Protocol:

1) *The receiver picks a random $1 \leq k \leq q$, sets public key $PK_r = g^k$ and $PK_{1-r} = C/PK_r$, and sends PK_0 to the sender.*
2) *The sender computes $PK_1 = C/PK_0$ and chooses random s $\subset Z_q$. The sender encrypts x_0 by $E_0 = [g^s,\ H(PK_0^s) \oplus x_0]$, x_1 by $E_1 = [g^s,\ H(PK_1^s) \oplus x_1]$, and the sends E_0 , E_1 to the receiver.*
3) *The receiver computes $H((g^s)^k) = H(PK_r^s)$, and uses it to decrypt E_r.*

Fig. 2. Naor-Pinkas 1 out of 2 OT

C. OPRF

PARAMETERS:

- Two parties: Sender S and Receiver R.
- S holds a random seed s for the OPRF
- R has an input r for the OPRF

FUNCTIONALITY：

- S outputs $\perp$
- R outputs $F(r, s)$

Fig. 3. Functionality of OPRF

OPRF is another atomic operation of MPC. In OPRF, the sender holds a seed s for the PRF, and the receiver has an input i for the PRF. For the receiver, it can only evaluate $f(s,i)$ and other $f(s,j)$ such that $i \neq j$ looks totally random to it (this kind of OPRF is also referred as Relaxed-OPRF, where the Strong-OPRF requires that the receiver cannot get any information about the send's seed). For the sender, it cannot infer any information about the receiver's input i, but since it has the seed, any value can be evaluated at the server side using the PRF f. In the PSI protocols we are going to discuss, they used one-time OPRF. This kind of OPRF can be only used one time to ensure the security guarantee. It looks like to be very irrational to use one-time OPRF, but OT extension can realize lots of one-time OPRF using a small amount of OT and thus is practical to be used in PSI protocols.

D. Cuckoo Hashing

Cuckoo Hashing was first introduced by Pagh and Rodler in 2001 [4], but here we will give a brief description of a variant of it from [5]. Compared with the normal hashing, the first difference one may notice is that it uses two hash functions $h_1,\ h_2$. To avoids collision in cuckoo hashing, we need to relocating elements when a collision is found. First randomly choose a hash function and insert element e to a bin $B_{h(e)}$. If the bin $B_{h(e)}$ is already occupied, then the original element is evicted. However, we need to find a new place for the evicted elements, so it is then relocated using the same procedure. The reallocation is repeated until no more collision, or the threshold number of reallocations is reached, in which case we consider the last evicted element failed to take a proper place in the bins. If an element failed, then it will be put in the stash. The benefit from using more hash function is the raise of occupancy rate of the hash table and thus a smaller table is needed to hold many elements. Although cuckoo hashing is slower than the simple hashing, it is a space-efficient hashing scheme.

While the original cuckoo hashing has only one table of one dimension, i.e., each bin in the table can only holds one element, many variants of cuckoo hashing have been suggested and was showed to be more practical in some scenes. Here, we will mainly discuss two variants:

- **Cuckoo hashing with more buckets.** This was first proposed by Fan etc. in 2014 [6]. The main idea is that they make the bins or slots in the table to hold more than one element by introducing buckets. It becomes that bucket is now the container for element such that each bucket can hold only one element, but a bin can contain multiple buckets. In [6], they have demonstrated that the occupancy rate can raise with the increment of buckets in the bins. Our work used two buckets in each bin, which can be viewed as cuckoo hashing which has a two-dimension table. We denote it as two-way cuckoo hashing with two buckets.

- ***Cuckoo hashing with separate tables.*** This second variant was introduced by Facebook in their RocksDB. RocksDB is a persistent key-value store and engineers from Facebook uses cuckoo hashing to build SST file format which is optimized for fast point lookups. In this variant, instead of using only one table, each hash function has its own table, e.g., three-way cuckoo hashing has three hash tables. When a relocation is needed, the element is first evicted and then tries to occupy a bin in another table with the help of corresponding hash function. Intuitively, if we use those hash function in turn, i.e., the first hash function is tried, then the second, the third…, the occupancy rate will decrease with the increase of the hash function index.

E. An Technical Overview of Our Work

We first tested the load factor of simple hashing, three-way cuckoo hashing, two-way cuckoo hashing with two buckets and three-way cuckoo hashing with separate tables. The results showed that two-way cuckoo hashing with two buckets have slightly lower occupancy rate than three-way cuckoo hashing, while separate tables does not affect the occupancy rate at all. We also compare the results with past work. This step is indispensable in that it gave us a clearer

PARAMETERS

P_1 *has input* X

P_2 *has input* Y

with $|X| = |Y| = n$

s *is an upper bound on the stash size for Cuckoo hashing*

1) P_2 *specifies random hash functions* $h_1, h_2, h_3 : \{0,1\}^* \to [1.2n]$ *and tells them to* P_1

2) P_2 *assigns his items* Y *into* $1.2n$ *bins using Cuckoo hashing. Let* P_2 *keep track of* $z(y)$ *for each* y *so that if* $z(y) = \perp$ *then* y *is in the stash; otherwise,* y *is in* $h_{z(y)}(y)$*. Arrange the items in the stash in an arbitrary order*

`P_2 *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* $r_i \in [s]$*, 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 instance, with* P_2 *the receiver with input* $(r_1, \dots, r_{1.2n+s})$*.* P_1 *receives* $(k_1, \dots, k_{1.2n+s})$ *and* P_2 *receives* $F(k_i, r_i)$ *for all i.*

4) P_1 *computes:*

$$H_i = \{F(k_{h_i(x),\, x||i})|x \in X\},\ for\ i \in \{1,2,3\}$$

$$S_j = \{F(k_{1.2n+j\ x||i})|x \in X\},\ for\ j \in \{1, \dots, s\}$$

And sends a permutation of each set to P_2

5) P_2 *initializes an empty set* O*, and does the following for* $y \in Y$*:* $If\ z(y) = \perp$ *and* y *is at position* j *in the stash and* $F(k_{1.2n+j,y||i},\ y) \in S_j$ *Then* P_2 *adds* y *to* O*. If* $z(y) \neq \perp$ *and* $F(k_{h_i(y),\, y||i},\ y||z(y)) \in H_{z(y)}$*, then* P_2 *adds* y *to* O*.*

6) P_2 *sends* O *to* P_1 *and both parties output* O

Fig. 4. PSI based on BaRK-OPRF

view of the following works and helped us to explain the results.

We then tried to use two-way cuckoo hashing with two buckets and three-way cuckoo hashing with separate tables in the PSI protocol and tested the implementation on two different machines with different hardware and system. On the two test machines with Ubuntu 18, we got results that are explainable, where two-way cuckoo hashing with two buckets makes the performance a little worse than the original protocol, while separate tables does not give much performance boost.

We also tried to make the protocol works in parallel, and if everything went right, we would finish this on time and went on working on the memory optimization, but we faced a great challenge that the authors of [7] wrote their own network module, and this part is tightly built into the project. We actually finished the parallelization work pretty quickly based on the knowledge from previous work, but it took us a long time to modify and debug the network module. However, it turns out we were unable to finish this and we decide to rewrite the whole network module in the future.

II. CUCKOO HASHING IN OT-BASED PSI PROTOCOL

In this section, we will give a comprehensive description of how Cuckoo hashing was introduced in PSI protocols and how it evolved and helped the protocols to become more efficient. We will also briefly describe the BaRK-OPRF PSI protocol, but the details of BaRK-OPRF itself will be omitted for the reason that it involves too much coding theory, which are hard to explain and are not closely related to our works.

To begin with, Pinkas etc. constructed a fancy PSI protocol based on OT extension [7]. They start the construction with *private equality test* (PEQT), which aims at telling whether two elements are equal in a privacy-preserving manner. Each PEQT will consume some rows in the OT extension matrix. Then, the PEQT protocol was extended to *private set inclusion* protocol. To securely identify whether an element is in a set, the PEQT invoked for each element in the set against the target element, and this can be done paralleled in OT. Finally, a prototype of their protocol (PSZ protocol) was shown: one party can simply use the privates set inclusion protocol against the other party's set using every element in his own set. In PSZ protocol, the number of OTs needed is proportional to $O(n^2)$, where n is the size of the set, and this is where *hashing* steps in. Given that if two elements are equal, they must be hashed to the same bin using the same hash function. Consequently, the comparisons need only to be done between those two bins with same index. Pinkas etc. tried three different hashing schemes: simple hashing, balanced hashing and cuckoo hashing. If we use two hash functions, then simple hashing just put element into both hashed positions; balanced hashing would first calculate both hash value and put the element into the bin which contains less element; Cuckoo hashing involves kick and reinsert progress as described in §2 and requires that each bin can only contain one element and an element failed to be inserted is put into the stash. PSI with two-way cuckoo hashing was shown by them have the minimum number of OTs, considering that there is only one element in the bin, then we can use just *private set inclusion* protocol rather than the private set intersection protocol between bins. But in this case, instead of both parties using cuckoo hashing, it must be that the receiver uses the cuckoo hashing, and the sender uses simple hash, so that the sender maps each element to both of two possible bins, which later can be captured by the receiver. If we use cuckoo hashing on both sides, it might happen that the receive maps element e to bin i, but the sender maps the element to bin j and as a result the computed mask are not equal, and then the element is missed in the intersection. Also, to prevent leaking of extra information, both parties must fulfill all the bins and stash, so that the adversary cannot infer any information from the position and hash function. After utilizing cuckoo hashing, the receiver now can have at most one element in each bin, and thus the number of OTs needed is reduced to $O(w \cdot n)$.

Later in [9], serval optimization was applied to the PSZ protocol and thus an even more efficient protocol (PSSZ protocol) was constructed. One optimization is that they make use of *three-way cuckoo hashing*. three-way cuckoo hash has been proven to achieve better occupancy of the hash table and thus a smaller table and stash is needed on the receiver side to hold the elements. Thus, the number of comparisons is reduced because there are less bins, but each element on the sender side must produce one more mask use the third hash function so that all possible positions and masks can be covered.

Then in 2017, [9] further boosted this OT extension-based protocol. They continued the thinking of [7] [8] and presented their efficient construction of OPRF based on KK OT extension. With BaRK-OPRF, each comparison now consumes only one row in the OT extension matrix, and therefore make the number of OTs irrelevant to the element size. This, again, make the PSI protocol has a performance closer to the unsecure naïve hash method and therefore more practical. The full BARK-OPRF PSI protocol is presented by Figure 4

III. Our Works

A. Occupancy Rate of Different Cuckoo Hashing Variants

Occupancy rate of hash table is an important metrics for good hashing scheme when applied to the PSI protocols. The higher the occupancy rate, the smaller table size and smaller stash size, and therefore less privacy-preserving comparisons are needed. Although both [6] and [7] have conducted experiments on occupancy rate of three-way cuckoo hashing and two-way cuckoo hashing with two buckets respectively, we noticed that in the implementation of [4], the hash function were used in fixed order in turn instead of a randomly picked manner.

Therefore, we tested all the hashing schemes, built in a similar way to how it is done in [4], for their occupancy rate. In more details, the hash functions are realized by AES256 encryption functions in ECB mode with different keys, and they are used in a fixed order in turn. We tested simple hashing, three-way cuckoo hashing, two-way cuckoo hashing with two buckets and three-way cuckoo hashing with separate tables. The results are very useful for the next two experiments.

B. Substituting Cuckoo Hashing with Common Variants

Cuckoo Hashing efficiency in PSI is mainly affected by hash functions and hash table structure. For various reasons that to be mentioned in challenge part, we decided to stick to this paper version of implementation, so the hash functions are picked in fixed order. Our main attempts focus on the transformations on the hash table structure.

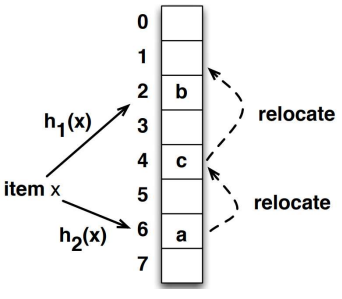

Fig. 5. Base two-way cuckoo hashing

As the stash in cuckoo hashing is always what we do not want to see, and it can not be controlled directly, stash size is usually used as an indicator for hashing effect. We tried on the numbers of hash tables and bucket size. PSI Based on Cuckoo Hashing with Two Buckets

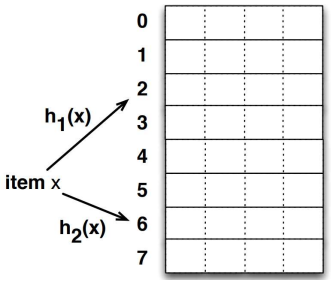

Fig. 6. Two-way Cuckoo hashing with buckets

It was showed by Fan etc. that the occupancy rate will also raise if we add more buckets to each bin in cuckoo hashing, and our test reconfirmed their states and gave even better result. Considering that in [8], Pinkas etc. changed the hashing scheme in their protocol from two-way cuckoo hashing to three-way cuckoo hashing because of its better occupancy rate, it is a straightforward thought to use cuckoo hashing with more buckets as another option to raise the occupancy rate.

In [8], they discussed how the number of hash functions impact the communication cost. Using more hash functions does reduce the number of OTs needed in the protocol, but it also induces extra communication cost, because for every element hold by the sender, it has to be hashed using all the hash functions and thus generated more masks which will then be transported to the receiver. They finally decided to use three-way cuckoo hashing based on their calculation.

Here we also want to perform a calculation to compare three-way cuckoo hashing with two-way cuckoo hashing with two buckets. Incidentally, a simple estimation and intuitive feelings told us that both three-way and two-way cuckoo hashing with three buckets are obviously far less efficient comparing with those with two buckets, so there is no need to have a try on three or more buckets. In the later hashing scheme, our test told that it can has an occupancy rate of 89%, and is very close to what can be achieved by the three-way cuckoo hashing. Moreover, the implementation of [9] utilize Random OT and double OT extension to maximumly reduce the OT cost. Considering those two facts, we ignore the difference on variant OT communication cost and focus on the number of masks by the sender. Although there are only two hash functions in the variant, every hash value in a bin of the simple hashing (hold by sender) needs to be compared to two elements in the corresponding bin of the cuckoo hashing (hold by receiver) separately. Thus, we need $2 \times 2 \times n_1 l = 4n_1 l$ masks, which is a greater value more than three-way cuckoo hashing's $3n_1 l$. Therefore, a worse performance is expected.

C. PSI Based on Cuckoo Hashing with Separate Hash Tables

Different from the first variant, cuckoo hashing with separate hash tables would have subtle influence on neither the number of OTs, nor the number of masks sent by the sender, and thus slightly affect the performance of the protocol.

We implemented two different version of separate hash tables, which are separate tables for cuckoo hashing only and separate tables for both cuckoo hashing and simple hashing. Both of the experiments should give similar result, where they might be slightly better than the original protocol.

While this variant may not bring much performance boost, it is not meaningless. There are two more future works from this variant. First of all, because we are using separate tables, then it is possible to concurrently calculate three sets of masks and send them in three threads, which can reduce the time of calculation. Nevertheless, we noticed that in [9], the authors used a machine with 256G RAM to test their PSI protocol, which indicates that their construction consumes lots of memory on large data set, and our experiment also confirms this: our machine with 32G RAM and 32G SWAP space quickly run out of memory on data set which has 2^{20} elements. Thus, if we use three separate tables, then it is possible to swap out one or two tables that are not in use on disk to save some memory space. Moreover, as we mentioned in §1.4, the elements inclines to the first table, then maybe we can use those three tables in a cache-like structure. However, at the same time, we admit that we are not sure whether this optimization would *introduce new security problem* or *undermine the privacy guarantee*. Considering those problems and more tables will lead to more mask consumption as well, three or more separate tables are not included in the scope of our experiment.

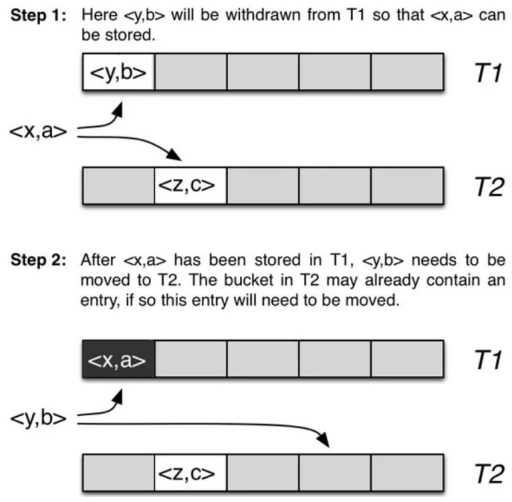

Fig. 7. Two-way cuckoo hashing using separate tables

We also implemented two versions when using this variant. For the first one, we only use separate tables in the cuckoo hashing side while the simple hashing still holds a single table. For the second version, both sender and receiver have three separate tables for their hashing scheme, and thus more structured and would be more rational for the parallelization work.

Table I. Overall communication for a larger number of hash functions h. Communication is given for a) $n1 = n2 = 2^{20}$ and b) $n2 = 2^8 \ll n1 = 2^{20}$ elements of σ = 32- bit length

h	Util. [%]	#OTs	#Masks	Comm. [MB]	
				$n_1 = n_2$	$n_2 \ll n_1$
2	50.0	$2.00n_2$ t	$\mathbf{2}n_1\ell$	148.0	**17.0**
3	91.8	$1.09n_2$ t	$3n_1\ell$	**99.8**	25.5
4	97.7	$1.02n_2$ t	$4n_1\ell$	105.3	34.0
5	**99.2**	$\mathbf{1.01}n_2$ **t**	$5n_1\ell$	114.6	42.5

IV. Technical Challenges

A. Compilation

Surprisingly, the very first problem that bothers us is to compile the codes from (Kolesnikov, et al.). The authors, on their Github Repository, claimed that their code can be complied on Windows, Linux and MacOS, so we first tried the MacOS, on which we spent nearly three days to solve all the complier and dependency problems to get an executable program. However, it outputs "illegal hardware instruction", an error that both of us have no idea of. Then we switched to Windows, and after a few hours of simple attempts, we did not want to waste more time on compilation, thus gave up and decided to use Ubuntu 18.04LTS, which should have the best compatibility.

Even on Ubuntu, both of the configuration script and the dependency specified by the authors do not work, so we have to try different versions of dependency to find a working one and then manually config the dependent path according to the script content. The program has a very poor management of the dependencies and more than half of the issues on Github are complains about building error. The reason why the repo is lack of support might be that the authors had coded a polished version which is included in their research group's cryptography kit. However, they also mentioned that if anyone wants to compare his/her own protocol with theirs, this version of implementation should be used. As a result, we decided to stick to this paper version of implementation.

B. Understand the BaRK-OPRF PSI Implementation and the Optimization Technics

If one follows the thought of [7] [8] (Kolesnikov,et al.), as we presented in §2, he or she might find it pretty easy to understand the whole protocol. However, the authors of [9] make use of a very important optimization technic, called Radom Oblivious Transfer (ROT). The main idea of ROT is that instead of making the sender transfer one message to the receiver obliviously, the sender holds no message at the beginning but can generate two messages using the secrets from OT and the receiver can also generate one of the messages, which would be generated by the sender, depends on its choice bit. This becomes more complicated when applied to the OT extension, and we are not going to give a detailed description of it here, but the main point is that implementation looks totally different from the protocol presented in the paper, and it again took us a long time to understand how ROT helped the protocol to become more efficient while does not break its security assumption.

C. Not Suitable Hardware

The authors of [9] used Intel(R) Xeon(R) CPU E5-2699 v3 2.30GHz CPU and 256 GB RAM to test their work, but, apparently, it is not feasible for us to find a machine that has comparable hardware. As a result, our tests on set size of 2^{24} cannot complete, because our machine with only 32GB RAM run out of memory. It is bad news to us, because in [9], the authors stated that the advantage of this PSI protocol is more obvious on large data set. However, we tried to give a comprehensive analysis based on the test on smaller sets.

D. Unexplainable Result on Machine 2

Another challenge we encountered is the unstable performance in the laptop machine which will be suggested

Protocol	Phase	set size n			
		2^8	2^{12}	2^{16}	2^{20}
BaRK-OPRF	Online	18	26	199	4611
	Offline	98	102	118	984
Two buckets	Online	19	24	188	4258
	Offline	96	95	113	**735**
One-side multi tables	Online	18	26	196	**3533**
	Offline	92	88	115	942
Two-side multi tables	Online	18	27	164	4742
	Offline	92	89	107	947

Table II. Running time in ms for PSI protocols with n elements in online and offline phases. Experiments in this table were implemented on a computer with Intel(R) Core(R) i7-8750H CPU **unfixed frequency** and 32GB 2400MHz

later in the benchmark environment. At first, the test results on the laptop were quite different from what we expected.

Table III below is a representative experimental result intercepted from those unexplainable results in machine 2. It shows that PSI protocol with cuckoo hashing with one-side multi tables has the best performance and the PSI protocol with two-way cuckoo hashing with two buckets is slightly better results than the other two. However, we cannot give any reasonable explanation for this, because not every test result follows the regulation above since the BaRK-OPRF protocol may become the best in the next result.

After eliminating various possible causes such as the version of dependent packages, swap partition and irrelevant human operations, we find that there was still a huge gap of up to 30% between different laptop test results. It took us a few days to find out that the CPU does not operate at a fixed frequency on this machine. Although the CPU frequency is fixed in the following tests, the test results on the laptop still slightly fluctuate compared with those on the desktop machine. Since this fluctuation has been within the error range we can accept, the laptop test results presented in this report later are the average results in many tests.

E. Use the Cuckoo Hashing Variants

The next technical challenge is to really substitute the cuckoo hashing with its variants. First, follow from 4.2, the authors of [9] utilize ROT to boost the performance, but this progress is not that simple. ROT does not exist as a low-coupling part in the implementation but was highly aggregated into the program. This is because of that when using ROT combined with the OT extension, the real matrix rows would be generated at the very end of the protocol. Intuitively, the OT extension can become a pre-computation, but in the construction of BaRK-OPRF, the set element hold by receiver is also part of the input to the OT extension and thus the authors split the OT extension to multiple steps and inserted them into different place. Second, the protocol is complicated in that it involves three rounds of interaction, and because the set size might be very large, each communication instance is further divided into smaller steps. Thus, we must carefully deal with those network communication codes. Those two facts greatly increase the difficulty of changing the hashing scheme in the implementations.

F. Poor Network Module

In the original implementation, the authors wrote their own network module, which is a wrap of the boost asio library. As stated in §1, we planned to finish the parallelization work and then works on memory optimization. Considering that we have played a lot with the protocol at the time when we tried to parallelize the protocol, it did not take us a long time to make it. However, it turns out that the biggest problem is the network module. From the interfaces, we first thought that the network module did support multi-threads because the authors left comments indicates how to use the network module in multi-threads. But when we test our code, there were always errors with the network. Thus, we tried to find out what was going wrong and tried a lot to fix the bugs, but after more than a week stuck on the network part, we still cannot make it work.

From our current view, the only solution is to rewrite the whole network module, but this is a tremendous work because this module is tightly built into the implementations. We currently decided to use GPRC as the tool for network communication. Although this higher-level protocol might introduce extra network consumption, but it is still meaningful as long as we make the all the implementations use the same network module and then the results from comparisons are consistent.

G. Version Conrtol

We have suggested above that we came across much troubles during the compilation of this project which also brought many obstacles to our project management. Although we use git as our version control system which did bring much convenience to us, complex and fragile dependencies made this project less easy to use, especially when we try to checkout different branches to implement different protocols, because those headache and complicated dependencies also need to be checkout with the branch. Moreover, we cannot edit our proposal, survey and report using word simultaneously, so annoying versions from different periods filled the whole folder.

V. Experiment Results

A. Implementation details

We used python to implement simple tests to find out the occupancy rate of different kind of hashing schemes. For simplicity, all the implementations are hard coded and run on a *Jupyter notebook*. Because we only need the occupancy rate and do not care about the execution time, so no optimization technic is used. We built the program in a way to be as consistent as possible to the implementation of [4], So the hash functions are realized by AES256 encryption functions in ECB mode with different keys, and they are used in a fixed order in turn. We run each test for 100 times and take the average occupancy rate as the result.

For the PSI protocols, we programmed our work based on the implementations from [4], which can be found on the GitHub page: https://github.com/osu-crypto/Bark-oprf. We tried our best to only change the codes which are related to the hashing schemes, so that we could clearly find out the

difference of using cuckoo hashing variant, but it turns out that the hashing scheme are tightly built into the protocol, so we had to recode many parts of the protocol. We built our program both on Ubuntu 18.04LTS on portable SSD and on Ubuntu 18.04LTS in WSL of Windows10. The offline execution and online execution are timed separately. Each data size was tested for 10 times, and the average result would be shown when all the trails were finished.

B. Cuckoo Hashing Variants

After estimating the theoretical performance of different cuckoo hashing variants, we choose three of them that may bring improvement or has similar performance after being integrated into OT-based PSI protocol. In the experiment of this paper, different type of protocols has applied their corresponding cuckoo hashing variant. Moreover, the code of transmission layer, sender and receiver has been modified adaptively.

1) Cuckoo hashing with two buckets

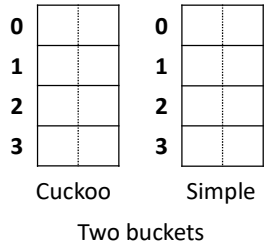

Fig. 8. Cuckoo hashing using two buckets

We expand the hash table of single bucket to double buckets which applied to both sender and receiver. This type of variant reduces the length of the hash tables while it can greatly improve the load factor comparing to the single bucket cuckoo hashing.

2) Cuckoo hashing with one-side multi tables

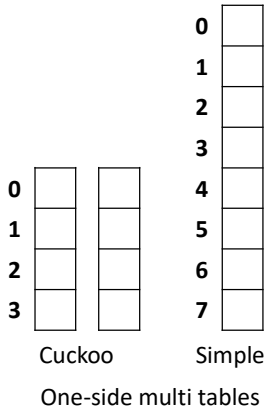

Fig. 9. Cuckoo hashing using one-side multi tables

We split the hash table on the cuckoo hashing side, while the simple side remains in the original state and only simply corresponds with the other side. The number of split tables is consistent with the number of hash functions. The above figure shows the split form with two hash functions.

3) Cuckoo hashing with two-side multi tables

Different from the variant 2), we also split the hash table in the simple hashing side, which is more convenient for correspondence and more friendly to batch transmission and multi-threaded operation. Meanwhile, the number of split tables of both sides should be consistent with the number of hash functions.

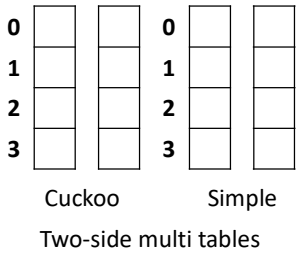

Fig. 10. Cuckoo hashing using one-side multi tables

C. Benchmark Environment

All our experiments were conducted on two machines. The first one has Intel CPU 10900K with 32GB 3000MHz RAM and the second one has Intel CPU 8750H with 32GM 2400MHz RAM. The sender and receiver are run on the same machine to simulate a LAN setting, which would have a 0.2ms latency. Both sender and receiver use a single thread to execute the protocol.

D. Parameters

For most parameters, we decided to use ones that are consistent to the parameters in the original protocol, but for the number of bins in cuckoo hashing, we must slightly modify the way it is organized to match the variant scheme. Because our test showed that the variants have similar occupancy rate with the original cuckoo hashing scheme, we still used $1.2n$ slots in total, but for the cuckoo hashing with two buckets, we need to make the table in two dimension and thus $0.6n$ bins and each bin have two buckets, also the simple hashing would also have 0.6n bins under this scenario. Cuckoo hashing with separate tables has similar setting, there will be 1.2n slots in total, but because there are three separate tables, each table would contain 0.4n bins and it is the same for the simple hashing.

E. Results

1) Occupancy Rate

Table III. Average occupancy of different hashing technique while mapping to hash tables with size of $n = 2^{20}$

hashing technique	**occupancy rate** [%]
simple hash	63.2
three-way cuckoo hash	93.5
two-way cuckoo hash with two buckets	88.0
three-way cuckoo hash using separate tables	93.5

As we can see in the table VI, the relative ranks are consistent to our normal knowledge, where the three-way cuckoo hashing has the best occupancy rate, then two-way cuckoo hashing with two buckets and simple hashing is the last. Also, we found that using separate tables in three-way cuckoo hash with size of $n=2^{20}$ has a negligible influence on the occupancy rate.

However, the exact occupancy rate we got are small different from what we have learned from the past works. In [6], the authors declare that using two buckets gives occupancy rate of 84%, while our test gave a result of 88%, which is small factor larger, but closer to the occupancy rate of three-way cuckoo hashing which is around 93%. The

Protocol	Phase	set size n			
		2^8	2^{12}	2^{16}	2^{20}
BaRK-OPRF	Online	4	10	100	2459
	Offline	55	55	67	326
Two buckets	Online	5	11	179	4052
	Offline	56	56	67	278
One-side multi tables	Online	4	14	118	2829
	Offline	56	55	67	281
Two-side multi tables	Online	4	10	102	**2437**
	Offline	59	54	67	**263**

Table IV. Running time in ms for PSI protocols with n elements in online and offline phases. Experiments in this table were implemented on a computer with Intel(R) Core(R) i9-10900K CPU and 32GB 3000MHz RAM

Protocol	Phase	set size n			
		2^8	2^{12}	2^{16}	2^{20}
BaRK-OPRF	Online	18	26	169	3341
	Offline	95	96	118	941
Two buckets	Online	16	26	189	4185
	Offline	92	95	108	635
One-side multi tables	Online	17	25	176	3343
	Offline	86	88	125	912
Two-side multi tables	Online	17	26	156	**3295**
	Offline	86	89	107	**678**

Table V. Running time in ms for PSI protocols with n elements in online and offline phases. Experiments in this table were implemented on a laptop with Intel(R) Core(R) i7-8750H CPU and 32GB 2400MHz RAM

reason might be that in our implementation, the hash functions are picked in fixed order rather than used randomly. Also, the implementation in [7] gives a really high limit for reinsertion tries and this might exceed the maximum depth for stack in python, so we limit this to 128 which is also a very large number for retries, and this might count for the rise in load factor.

2) PSI Protocols Performance

On both machines mentioned above, we got similar results as table IV and table V show above.

As stated in §3 we expect to see that two-way cuckoo hashing with two buckets induces a slightly worse performance for the PSI protocol, and PSI protocol with cuckoo hashing with separate tables has a similar performance with the original one. Both of the result from the tables affirmed our guess.

Notice that when using two-way cuckoo hashing with two buckets, the sender needs to compute 25% more masks, and thus induces around 700ms computation time in the online stage. On using separate tables, both of the two implementations give similar result and are slightly better than the original protocol, we denote this performance improvement to the fact that when using separate tables, there is no need to record the hashing function used, and thus less calculation and record.

VI. Conclusion & Future Works

Although we did not get much performance boost on using cuckoo hashing variants in the state-of-art OT-based PSI protocol, we have a better understand of both the PSI as a whole and the BaRK-OPRF PSI protocol itself after several attempts in transforming the hash structures. Also, we know a new optimization technic of OT, ROT, and learns how it can be applied to OT extension and BaRK-OPRF to reduce the unnecessary computations to boost the performance.

As the bad situation such as the poor program management and the poor dependency management left by the BaRK-OPRF project, we need to do some effort to organize and improve the project architecture. Also, as we claimed in §3, our implementations with separate tables make the masks generations stage can be easily paralleled which might boost the protocol performance. Furthermore, we faced the challenge of a poor network module as stated in §4, we decided to rewrote the network part using GRPC to simplify the implementation and after this, we should go on to optimize the memory usage, which is also very meaningful for large sets intersection, and test the protocols with WAN environment to perfect these attempts with cuckoo hashing variants.

VII. Reference

[1] M. Naor and B. Pinkas, "Efficient Oblivious Transfer Protocols," 2001.

[2] Y. Ishai, J. Kilian, K. Nissim and E. Petrank, "Extending Oblivious Transfers Efficiently," 2003.

[3] V. Kolesnikov and R. Kumaresan, "Improved OT Extension for Transferring Short Secrets," 2013.

[4] R. Pagh and F. Rodler, "Cuckoo Hashing," 2001.

[5] A. Kirsch, M. Mitzenmacher and U. Wieder, "More Robust Hashing: Cuckoo Hashing with a Stash," 2009.

[6] B. Fan, D. G. Andersen, M. Kaminsky and M. D. Mitzenmacher, Cuckoo Filter: Practically Better Than Bloom, 2014.

[7] V. Kolesnikov, Kumaresan, Ranjit, Rosulek, Mike, Trieu and Ni, "Efficient Batched Oblivious PRF with Applications to Private Set Intersection".

[8] B. Pinkas, T. Schneider and M. Zohner, "Faster Private Set Intersection Based on OT Extension," 2014.

[9] B. Pinkas, T. Schneider, G. Segev and M. Zohner, "Phasing: Private Set Intersection using Permutation-based Hashin," 2015.

VIII. APPENDIX

A. Open Source

All our code can be found on Github page:
https://github.com/yuany3721/PIoBD-CHV

The name PIoBD-CHV comes from its full name of *Privacy Issues of Big Data – Cuckoo Hashing Variants in OT-Based PSI protocol*.

B. Member Contribution

Research: Huiyang He, Chenglin Li
Proposal: Huiyang He, Chenglin Li
Coding & Test: Huiyang He, Chenglin Li
Survey: Huiyang He
Report: Huiyang He, Chenglin Li
Presentation: Chenglin Li
Project Management: Huiyang He