FABLE: Batched Evaluation on Confidential Lookup Tables in 2PC

Zhengyuan Su*, Qi Pang†, Simon Beyzerov†, Wenting Zheng†
*Tsinghua University, †Carnegie Mellon University

Abstract

Secure two-party computation (2PC) is a cryptographic technique that enables two mutually distrusting parties to jointly evaluate a function over their private inputs. We consider a 2PC primitive called confidential lookup table (LUT) evaluation, which is useful in privacy-preserving ML inference and data analytics. In this setting, a server holds a confidential LUT and evaluates it over an input secret shared between a client and the server, producing a secret-shared output. Existing approaches suffer from high asymptotic complexity and practical inefficiency, with some designs lacking confidentiality guarantees for the LUT. Recognizing that many applications of confidential LUT evaluation require processing multiple inputs with the same LUT, we propose FABLE, a system designed to efficiently evaluate a LUT on a large batch of queries simultaneously. Compared to the state-of-the-art confidential LUT evaluation methods, FABLE achieves up to 28-101× speedup in LAN environments and up to 50-393× speedup in WAN environments.

1 Introduction

In recent years, users have become increasingly concerned about how their personal information is being collected and used by third parties. In response, many privacy regulations have been implemented to restrict data collection, such as the General Data Protection Regulation [66] and the California Consumer Privacy Act [29]. At the same time, it is important for many organizations to compute on their users' data to extract valuable insights. Secure multi-party computation (MPC) [12, 13, 59] (as well as the more specialized two-party version, 2PC) is a promising cryptographic technique for addressing this tussle between providing privacy and preserving functionality. MPC allows two or more parties to collaboratively evaluate a function without revealing the parties' private inputs or the function's intermediate outputs to each other. MPC can be used for privacy-preserving workloads, including privacy-preserving machine learning (ML) [56, 59] and analysis of financial and medical data [6, 58, 64].

In this paper, we study a useful primitive in 2PC: confidential lookup table (LUT) evaluation. This primitive allows two parties to jointly evaluate a confidential LUT hosted by one of them on a secret-shared input (i.e., input is an intermediate output in 2PC). The evaluation result is also secret-shared so that it can continue to be used for further computations in 2PC. Confidential LUTs are useful to secure not only applications that rely on table lookups but also the evaluation of arbitrary functions with LUT approximation. We observe that many workloads in both ML and data analytics need to evaluate LUT over thousands of different inputs, and thus can benefit from batched execution. We motivate confidential LUTs with two concrete use cases:

Privacy-preserving ML — Secure embedding lookup. In privacy-preserving inference, a client and a server jointly execute 2PC on the client's private input tokens and the server's proprietary model. State-of-the-art language models [33, 46] contains an embedding layer, which are LUTs that map input word tokens to embedding vectors that carry meaningful information. Embeddings are often confidential for proprietary models [54]. In end-to-end 2PC inference workloads, the embedding lookup may come after ML guardrail (e.g., input sanitization) but before the rest of the model inference [44, 62]. Thus, the inputs and outputs of the embedding LUT are both secret-shared in 2PC. Additionally, since a user's input often contains hundreds of tokens, the same embedding LUT needs to be evaluated many times on different input tokens.

Private data analytics — Secure joins. Privacy-preserving SQL analytics usually involves executing queries on tables held by different parties. SQL benchmarks that simulate a business data warehouse workload, such as [61], can also be translated into the two-party setting, where customer information is held by a financial institution and transaction information is held by a payment company (e.g., Stripe). Using 2PC, these organizations can securely execute cross-party

FABLE stands for **FA**st **B**atched **L**ookup **E**valuation.

queries for business insights, such as joins over their confidential tables. In certain scenarios, FABLE can be used to speed up such cross-party, multi-way joins by treating one party's table as a confidential LUT when it is joined with some 2PC outputs of other nested queries. All the rows in the other table in 2PC form a batch of secret-shared inputs, so the join becomes a batched lookup of the local table.

While confidential LUT protocols are suitable for these applications, existing works have two key limitations.

Inefficient for large LUTs and lightweight client. Many applications require large LUTs and fit into a client-server model, so an ideal protocol should work with a powerful server and a lightweight client. Existing protocols either suffer from communication cost linear in the size of the LUT [14, 23] or distribute heavy computations to both parties [24, 65], restricting them to small tables or resource-intensive clients.

Lack of LUT confidentiality. Some existing 2PC protocols [14, 40] cannot guarantee LUT confidentiality because the client learns information about the LUT content from the 2PC circuit structure, which is unacceptable when the table contains sensitive or proprietary information.

We propose FABLE, an efficient protocol for confidential LUT evaluation in 2PC, designed for batch workloads and lightweight clients. Our key insight is that private information retrieval (PIR) offers a similar lookup functionality. PIR schemes are extensively developed for communication efficiency and scalability. Recently, batch PIR protocols [4,49,51] have been introduced to improve PIR for batch workloads. This technique is our starting point for building an efficient confidential LUT evaluation system.

However, adapting batch PIR to 2PC is challenging. In standard batch PIR, the client can preprocess the plaintext input queries locally. In contrast, in 2PC, the client does not have access to the plaintext input queries, so the preprocessing must be performed within 2PC. Naively doing so is inefficient and introduces significant overhead. Additionally, in 2PC, the server's LUT is confidential, whereas in traditional PIR, the table is typically assumed to be public.

FABLE addresses this challenge through a set of novel techniques. Firstly, batch PIR encodes the database to multiple buckets, and the client translates the original queries for the database to queries for each bucket. Cuckoo hashing is typically used in this query translation process, but there doesn't exist a practical 2PC cuckoo hashing protocol. Therefore, we propose to offload this computation to the client side after obfuscating the inputs with oblivious pseudorandom functions (OPRF) [15], and adapt PIR by keywords [3] to enable query construction based on OPRF-obfuscated input. Secondly, cuckoo hashing requires the uniqueness of the input indices, so we need to remove the duplicates from the input (deduplication) before query translation and add them back to the output (expansion) after the lookup is done. To implement these procedures, we adapt algorithms in oblivious storage systems [22] to 2PC, and improve the communication efficiency by over 6× with a novel caching mechanism. Finally, LUT confidentiality needs to be carefully examined because PIR does not protect the database content. Based on our insight that recent batch PIR schemes [49, 51] only include *requested items* in the response, we effectively protect the requested items by masking and hiding additional leakage about the database with function privacy techniques [5].

By addressing these challenges, FABLE can efficiently evaluate *large* and *confidential* LUT on *a batch of secret inputs*, while enjoying *lightweight client computation*, *concretely efficient server computation*, *scalable communication*, and *LUT confidentiality*. With various network configurations, LUT sizes, and batch sizes, our end-to-end implementation of FABLE achieves up to 28-101× speedup in LAN and up to 50-393× speedup in WAN compared to state-of-the-art 2PC protocols. For the two end-to-end applications mentioned above, FABLE brings 456× (178×) and 10× (27×) speedup in LAN (WAN), respectively. In conclusion, we make the following contributions:

1. We design FABLE, a scalable, confidential LUT evaluation protocol in 2PC.

2. We develop new cryptographic protocols to adapt batch PIR to 2PC, achieving lightweight client computation, efficient server computation, and sublinear communication. Specifically, FABLE

   (a) offloads query translation from 2PC to client local processing via new OPRF-based techniques and PIR by keywords;

   (b) improves the deduplication (and expansion) protocol by 14.0% (and 84.1%) via a novel caching mechanism; and

   (c) ensures LUT confidentiality with database masking and noise flooding.

3. We implement FABLE with extensive evaluations and show its applicability to ML and analytics workloads.

2 Preliminaries

2.1 Notation

We refer to the server as P_0 and the client as P_1. They jointly evaluate a lookup table T held by the server on a secret value x to obtain the secret output $v = T(x)$. The secret value x is encoded as a *secret-share* (denoted by $[\![x]\!]$) between the two parties. We denote by $[\![x]\!]_0$ and $[\![x]\!]_1$ the secret shares held by the server and the client, respectively. Similarly, the output is shared as $[\![v]\!]_0$ and $[\![v]\!]_1$. We denote string concatenation as $\|$ and the set of integers $\{1, 2, \ldots, k\}$ as $[k]$. Table 1 concisely outlines the notations frequently used throughout this work.

Table 1: Frequent notations used throughout this paper.

Notation	Description
T	the LUT to be evaluated
δ	input bit size of the table
σ	output bit size of the table
N	size of T
b	# of queries in a batch
B	# of buckets
N_B	size of a bucket
n	polynomial modulus degree in FHE
w	# of hash functions used in cuckoo hashing

2.2 Secure Two-Party Computation (2PC)

2PC enables a pair of mutually distrusting parties P_0 and P_1 to securely compute a joint function f over a set of mutually private inputs a_0, a_1 without revealing any additional information beyond the output of the function itself.

2PC Primitives. In addition to specialized 2PC procedures for specific functions, open-source 2PC libraries [13, 59] can compile arbitrary functions into secure computation protocols. We utilize the following well-studied and widely implemented primitive 2PC protocols as subroutines in FABLE:

1. $\mathsf{MUX}(\llbracket cond \rrbracket, \llbracket x \rrbracket, \llbracket y \rrbracket)$: the multiplexer, returns $\llbracket x \rrbracket$ if condition $\llbracket cond \rrbracket$ is true and $\llbracket y \rrbracket$ otherwise.
2. $\mathsf{CondSwap}(\llbracket cond \rrbracket, \llbracket x \rrbracket, \llbracket y \rrbracket)$: conditional swap, swaps $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$ if $\llbracket cond \rrbracket$ is true.
3. $\mathsf{Sort}(\llbracket x_1 \rrbracket, \ldots, \llbracket x_p \rrbracket)$: a Bitonic Sorter [8], sorts the secret-shared values $(\llbracket x_1 \rrbracket, \ldots, \llbracket x_p \rrbracket)$. It is a well-studied oblivious algorithm (i.e., input independent) that can be easily converted into 2PC.

2.3 Oblivious Pseudo-Random Functions and 2PC-Friendly Hash Functions

Oblivious pseudo-random functions (OPRF) [15] are pseudo-random functions (PRF) that allow interactive, oblivious computation where the server provides the key and the client provides the input. It is ensured that the client learns the PRF outputs and the server learns nothing. We focus on an efficient OPRF construction that evaluates a PRF inside 2PC using generic MPC techniques. As for the PRF, the Advanced Encryption Standard (AES) is the most common choice; however, alternatives like LowMC [2], notably used in DORAM and PSI [25, 37], offer lower multiplicative complexity due to fewer non-linear gates. It has 552 AND gates with 128-bit keys and 64-bit blocks, which is significantly fewer than AES's 5,120 AND gates [10].

In our notation, we assume the input and output are secret-shared. Considering the underlying PRF $f_K(x)$, we define two procedures: $\mathsf{OPRF.EvaluateLocal}$ takes input x and key k and returns $o = f_K(x)$; $\mathsf{OPRF.Evaluate}$ takes secret-shared input $\llbracket x \rrbracket$ and key k and returns secret-shared output $\llbracket o \rrbracket$. Their formal definitions can be found in Appendix A.

2.4 Private Information Retrieval

A private information retrieval (PIR) protocol allows a client to retrieve a record from a *public* database held by the server without revealing the index of the retrieved record to the server. Practical single-server PIR protocols [38, 50] can be constructed using somewhat homomorphic encryption (HE). We call such PIR schemes *HE-based PIR*. They generally consist of three steps: encoding queries into HE ciphertexts (query), processing these query ciphertexts with the database to produce answer ciphertexts (answer), and decoding the answer ciphertexts to retrieve the plaintext answers (extract). PIR protocols usually have *sublinear* communication in the size of db. However, for concrete computation costs, the server of non-preprocessing PIR must inherently perform a linear amount of work over the database, although the client often only needs sublinear work to construct a query. Finally, we list two PIR variants that are useful in our protocol construction. Computational symmetric PIR (CSPIR) [27] is a variant that ensures the privacy of db, i.e., the client learns nothing more than the requested item from the database. This stronger security guarantee is closer to our LUT setting and can provide a simple LUT protocol with small changes. Batch PIR [4, 49, 51] is a line of work that optimizes *amortized per-query* cost under batch workloads, where the server hosting a database serves multiple private queries from the client. We will use the idea to enhance the efficiency of our protocol.

2.5 Batch Codes and Their Application in PIR

Batch Codes. Batch codes are proposed in [41] to amortize the cost of PIR queries and reduce redundant server-side work. Formally, a (N, Q, b, B) batch code encodes a database db with N elements into B buckets $(d_1, \ldots, d_B)$ of total size Q where $d_j \subseteq \mathsf{db}$ for all $1 \leq j \leq B$, such that for any batch of b indices $(i_1, \ldots, i_b) \in [N]$, the entries $(\mathsf{db}_{i_1}, \ldots, \mathsf{db}_{i_b})$ can be retrieved by reading at most one entry from each bucket.

Probabilistic batch code (PBC) [4] is a variant of efficient batch code that has a weaker correctness guarantee (i.e., allowing for a small failure probability). The total number of items in buckets encoded by PBC is much smaller than other batch code constructions. As PIR's computation cost is typically linear to the total database size, the design exhibits superior performance and thus is widely adopted in batch PIR [49, 51]. The PBC construction is based on w-way cuckoo-hashing with $p \approx 2^{-40}$ probability of failure, achieving total encoded database size $Q = wN$ with $B = 1.5b$ buckets. During the encoding phase, each database index is mapped to w buckets among B buckets in total, into which the items are replicated. Given a set of queries, the client assigns each query to a bucket by cuckoo-hashing and translates it into a query to the bucket,

such that querying each bucket once serves all the queries. To encode b items, this PBC construction needs only w times larger storage, and $1.5\times$ more communication with $\frac{b}{w}$ times less server computation cost.

Batch PIR. Batch codes can be used to encode a PIR database into a set of smaller databases called *buckets* to enable efficient batch queries. For a database of size N and a batch of b queried indices, rather than performing b independent PIR queries of $O(N)$ server-side work, clients send $B > b$ queries over a set of buckets with total elements less than bN, resulting in a significantly reduced total cost. For example, when using PBCs with $w = 3$ and batch size $b = 4096$, the server computation cost with batch PIR is only 0.073% of the cost with normal PIR. Thus, it is capable of handling a large batch with significant performance gain over subcube batch codes.

We define several batch PIR procedures that we will use later: Encode, PIR.Query, PIR.Answer, and PIR.Extract. In a PIR execution, the server first prepares the database with Encode to encode a size-N database db into B buckets (i.e., the encoded database). The client calls PIR.Query to translate the plain query $i_1, \ldots, i_p$ into queries for each bucket, which will be sent to the server. The server processes the query with PIR.Answer and the buckets to obtain the response and send it back to the client. Finally, the client executes PIR.Extract on the response to extract plain responses $\mathsf{db}[i_1], \ldots, \mathsf{db}[i_p]$. The formal definitions can be found in Appendix A.

3 Overview

Figure 1 illustrates an example of how FABLE can be used in the context of a privacy-preserving SQL query. The SQL query first performs a nested query, followed by an equi-join and an aggregation. The input is the intermediate result of a nested query, expressed as a batch of b secret-shared values $(\llbracket x_1 \rrbracket, \ldots, \llbracket x_b \rrbracket) \in \{0,1\}^{\delta}$. The server holds a private table T, which can be viewed as an δ-to-σ-bit LUT. Thus, the equi-join can be efficiently executed by FABLE. The output is secret shares of the search results $(\llbracket T(x_1) \rrbracket, \ldots, \llbracket T(x_b) \rrbracket) \in \{0,1\}^{\sigma}$, which serve as input for the subsequent aggregation.

Threat Model. We adopt the *semi-honest* (a.k.a. *honest-but-curious*) threat model. We assume that the server P_0 and the client P_1 mutually distrust each other, and our protocol is secure against a semi-honest adversary running in probabilistic polynomial time. They faithfully follow the protocol while attempting to gain extra information that cannot be inferred from the output. Our protocol ensures that neither party can gain additional knowledge by analyzing the protocol transcript. This assumption is typical in 2PC and is useful in real scenarios when restricted trust can be established or the goal is to avoid passive, undetectable malware on one party [14].

Security Guarantees. We term FABLE as π_{FABLE} and state the security theorem in Theorem 1. The formal versions and the proof are deferred to Appendix B.

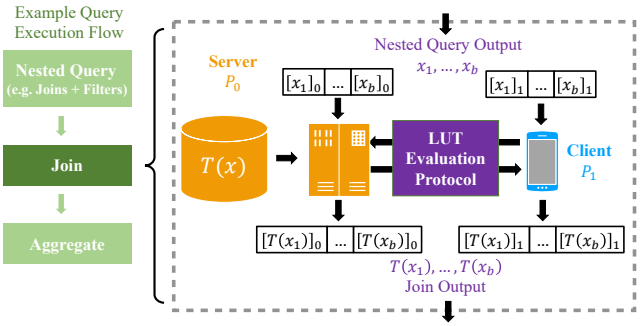

Figure 1: An example workflow for two-party SQL joins. The client and the server first run a nested query in 2PC and output a batch of secret-shared inputs $(\llbracket x_1 \rrbracket, \ldots, \llbracket x_b \rrbracket)$. They then evaluate an equi-join between the nested query's results and a large table held by the server. This join can be evaluated using FABLE as it is equivalent to a lookup on a large table T held by the server to retrieve the set of secret-shared outputs $(T(x_1), \ldots, T(x_b))$. The output of the equi-join stays secret-shared, as the query continues with an aggregation operation.

Theorem 1 (Security (Informal))**.** *With properly chosen b, B, N_B, for any LUT T and input $\mathbf{x}$, π_{FABLE} securely computes the confidential LUT evaluation functionality in the presence of static semi-honest adversaries.*

3.1 Design Goals

We propose the following design goals based on the applications we want to support:

1. *Lightweight client computation.* Typically, the client has much fewer computational resources than the server. So we expect the client's computation to be sublinear in the table size and concretely efficient.

2. *Efficient total communication.* 2PC protocols are typically communication-bounded [23], so we aim to achieve sublinear total communication in table size.

3. *LUT confidentiality.* Some existing protocols assume a public LUT, which is undesirable if the table's content is sensitive. LUT confidentiality enables the server to apply its secret functions on secret-shared data without worrying about the leakage of the function.

4. *Efficient server computation for batch workloads.* We aim to achieve practical server-side computational efficiency optimized for batched workloads.

There are two lines of work on confidential LUT evaluation, but none of the schemes meet all four design goals. Using LUT as a primitive in MPC to accelerate general MPC computation was introduced by Ishai et al. [40], who also propose the one-time truth table (OTTT) protocol as the first

Table 2: Comparison of existing works in terms of our design goals. FABLE is the only protocol that achieves all of them.

Protocol	lightweight client	sublinear comm.	LUT confidentiality	efficient server
OTTT [40]	✗	✗	✗	✗
OP-LUT [23]	✗	✗	✓	✗
SP-LUT [23]	✓	✗	✓	✗
FLUTE [14]	✗	✗	✗	✗
FLORAM [24]	✗	✓	✓	✗
2P-DUORAM [65]	✗	✓	✓	✗
FABLE (ours)	✓	✓	✓	✓

solution. Dessouky et al. [23] proposed LUT-based circuit synthesis to enhance 2PC efficiency by treating small LUTs as gates, compromising extra computation for reduced communication. They proposed two LUT evaluation protocols: Online-LUT (OP-LUT) for optimized online communication and Setup-LUT (SP-LUT) for overall communication efficiency. FLUTE [14] further reduced communication but at a higher computational cost. These protocols are tailored for small LUTs with linear communication costs in the LUT size, limiting scalability to large LUTs. OTTT and FLUTE introduce operations that depend on the LUT's content, so they are not applicable to confidential LUT evaluations. Both OP-LUT and SP-LUT protect the table content from the client, while SP-LUT offers a lightweight client in terms of computation.

Distributed oblivious RAM (DORAM) is a technique that enables data-dependent memory accesses on memory secret shared between two servers, without revealing the secret index. This is similar to the confidential LUT setting if we consider the secret shared memory as the confidential LUT and the secret index as the query. FLORAM [24] is a practical and scalable secure DORAM protocol that surpasses previous schemes with distributed point functions (DPF) [12]. 2P-DUORAM [65] further reduces communication with PIR, outperforming FLORAM in low-bandwidth conditions. The communication of both protocols is sublinear to the memory size, but it distributes balanced and heavy workloads for both parties and thus leads to significant client-side computation, which is linear to the memory size.

None of the above schemes is optimized for batch workloads. In Table 2 we summarize and compare to previous schemes in terms of the design goals. FABLE is the only protocol that achieves all four goals.

3.2 Strawman Construction

PIR is a natural first step towards designing a 2PC LUT protocol with sublinear communication, as it provides similar functionality under the same efficiency constraints. Inspired by the read-only version of 2P-DUORAM [65], we describe a simplified strawman design called 2P-DUORAM+. 2P-DUORAM+ is based on computational symmetric PIR (CSPIR), which is implemented by augmenting Spiral [50] with a general reduction [52] from CSPIR to PIR.

Let the input $[\![x]\!]$ be shared as $[\![x]\!] = [\![x]\!]_0 \oplus [\![x]\!]_1$ where $[\![x]\!]_0 = R \xleftarrow{\$} \{0,1\}^\delta$. In 2P-DUORAM+, the server rotates its table T by R to create T^R. As the i-th item in T^R is the $(i \oplus R)$-th item in T, it suffices for the client to retrieve the $[\![x]\!]_1$-th item in T^R, which is the $([\![x]\!]_1 \oplus R) = [\![x]\!]$-th item in T, i.e. the target. Sending this information back in plaintext leaks the result to the client, but the leakage can be easily prevented by applying a freshly sampled random mask r on T^R for the client to retrieve. The response revealed to the client as well as r held by the server form a Boolean sharing of the result $T(x)$. In DORAM, T is secret shared between the two parties, different from our setting where the server holds T entirely. Thus, 2P-DUORAM repeats the above procedure *twice* to obtain each of the table shares, leading to double the cost.

This design meets the first three goals, but the optimized communication also leads to significant server-side computation: To perform 128 reads on a database containing 2^{26} 64-bit words, 2P-DUORAM is 6× slower than FLORAM with 4 cores and 2× slower with 32 cores, even with 128× less communication [65, Figure 10]. This tradeoff between communication and computation makes it less appealing with high network bandwidth. Thus, we aim to combine the best of both worlds by taking advantage of the superior communication of PIR while reducing the concrete computation cost.

We observe that for workloads that make multiple read queries to the same LUT, the computation cost can be concretely amortized by techniques from batch PIR as introduced in Section 2.5. FABLE leverages this insight to optimize batch workloads in the 2PC setting, and we will demonstrate how this can improve applications (secure embedding lookup and query execution) in Section 6.

4 FABLE Protocol

In this section, we present the FABLE protocol, starting with an overview of our design (Section 4.1), followed by a formal protocol description (Section 4.2), and concluding with a discussion on how FABLE provides LUT confidentiality (Section 4.3).

4.1 Overview of FABLE Protocol

This section discusses several challenges in adapting the batching technique to our setting, along with an overview of our proposed solutions as shown in Figure 2. Recall that the protocol starts with a server holding a LUT where each row consists of key-value pairs $(x, T(x))$, where x is a δ-bit index, and $T(x)$ is the corresponding table value at x. The client and the server jointly hold a batch of secret shared queries $[\![x_1]\!], \ldots, [\![x_b]\!]$. By executing FABLE, the client and the server obtain a list of secret shared output $[\![T(x_1)]\!], \ldots, [\![T(x_b)]\!]$.

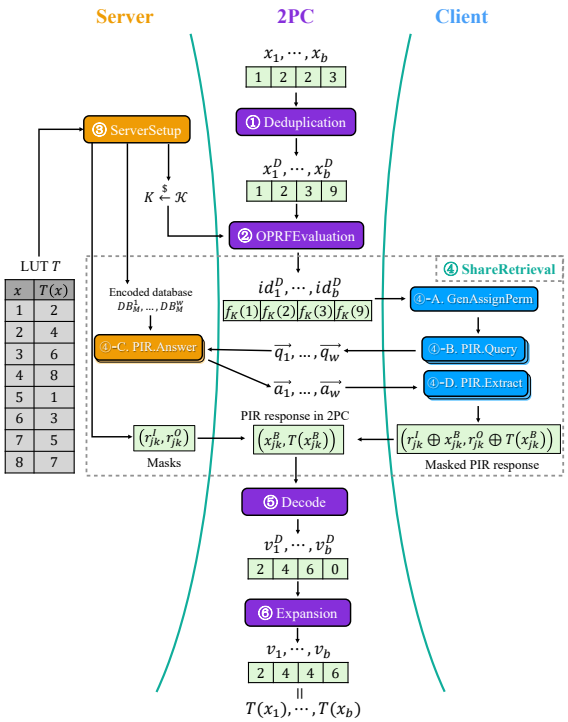

Figure 2: An Overview of FABLE with an example where LUT size $N = 8$ and batch size $b = 4$. $f_K(x)$ denotes the OPRF. FABLE starts with Deduplication that removes the duplicates in the input, followed by OPRFEvaluation that generates the IDs for the inputs and reveals them to the client. The subsequent ShareRetrieval and Decode perform a modified batch PIR to realize the lookup functionality. The results are finally post-processed by Expansion to recover the duplicates.

Efficient query translation. In batch PIR, the server encodes the database into an alternative form, organizing it into a set of buckets. The client then translates a batch of queries to the original database into a new set of queries, one per bucket. We call this process *query translation*. Although query translation can be directly executed by the client in standard batch PIR, this process is much harder in FABLE since the queries are in 2PC (thus unknown to the client).

Query translation involves two steps: (1) *query-to-bucket assignment*, where each query is mapped to a distinct bucket containing its response; and (2) *bucket index translation*, where each query is converted to an index within the assigned bucket to retrieve the response. The first step, query-to-bucket assignment, uses cuckoo hashing to ensure that different queries are assigned to different buckets. To our knowledge, no practical 2PC protocol exists for cuckoo hashing. Although an oblivious algorithm [32] can be adapted, it incurs a large constant overhead [17] and is impractical in practice. In contrast, FABLE offloads this step from 2PC to local computation while preserving security guarantees. This is done in ② OPRFEvaluation (see Figure 2), in which we use an OPRF $f_K(x)$ with a freshly-sampled key K *per batch* to randomize the input space, by mapping each input to a pseudorandom "ID". With the "ID"s, the client can locally assign queries to buckets in ④-A. GenAssignPerm without learning the original input queries.

The second step, bucket index translation, is also done by the client who looks up the appropriate index in a translation map. In standard batch PIR, this can either be generated locally by emulation or requested from the server [51]. In our setting, local emulation is impossible as the OPRF key is kept secret from the client, while requesting the map from the server results in a linear communication overhead because the map is as large as the encoded database. To address this challenge, FABLE adapts *PIR by keywords* within each bucket. Inspired by LPSI-PIRANA [49] and [3, Construction 1], we place the items within each bucket with cuckoo hashing during ③ ServerSetup, such that each item can only be found in w locations. Then in ④-B. PIR.Query during ShareRetrieval, it suffices for the client to retrieve w items per query, in order to find the desired item.

Input deduplication. Duplicates in the input queries are not supported in PBC since it relies on cuckoo hashing to assign queries to buckets. Thus, we need to replace duplicated input queries with dummy queries. Again, in standard batch PIR, the client can locally deduplicate since it knows the plaintext queries. FABLE designs two novel 2PC algorithms to address this issue: 1) ① Deduplication, a pre-processing step for replacing duplicates in input queries with dummy queries, and 2) ⑥ Expansion, a post-processing step to add back duplicate queries with the correct responses. As shown in Figure 2, Deduplication and Expansion are executed at the beginning and end of the protocol, respectively.

LUT confidentiality via masking and noise flooding. FABLE aims to ensure the privacy of the server's confidential LUT. As the items contained in the PIR responses are parts of the confidential LUT, they should be properly masked during retrieval. We use independent random values to mask each item the client retrieves, such that the responses are indistinguishable from true random values. In addition, given that the responses in SOTA batch PIR schemes [49, 51] are FHE ciphertexts whose noise would also leak certain information about the input LUT, we adopt the noise flooding technique [5] to prevent such leakage. More details on the protection of LUT confidentiality can be found in Section 4.3.

4.2 FABLE Protocol Workflow

In this section, we describe the sub-protocols in Figure 2 in detail. We provide detailed illustrations for ServerSetup, ShareRetrieval, and Decode in Figure 3. For the cuckoo hashing in our protocol, we assume that the parties agree on two lists of hash functions $(h_1^B, \ldots, h_w^B)$ and $(h_1^P, \ldots, h_w^P)$. All hash functions have domain O; $(h_1^B, \ldots, h_w^B)$'s ranges are $[B]$ and $(h_1^P, \ldots, h_w^P)$'s ranges are $[N_B]$. B and N_B are hyper-parameters

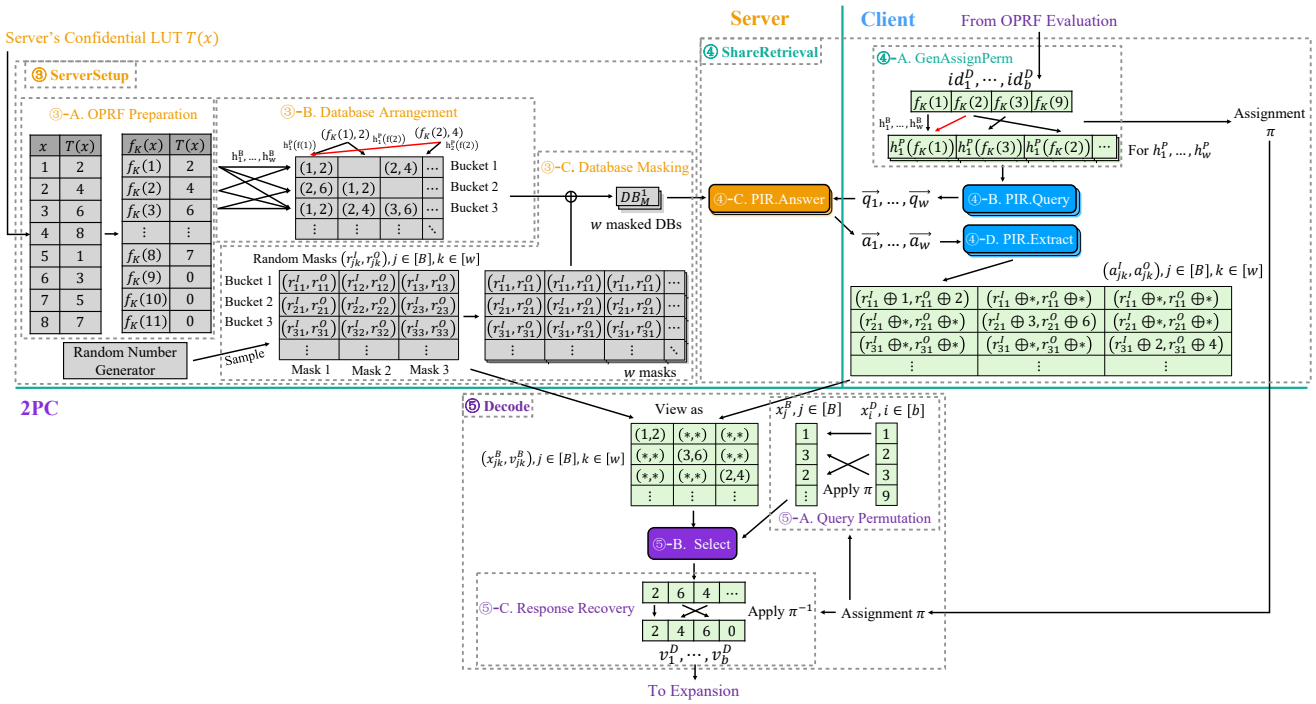

Figure 3: A detailed illustration of ServerSetup, ShareRetrieval, and Decode. This example has LUT size $N = 8$ and batch size $b = 4$. ServerSetup prepares the database to enable the client to retrieve the LUT items using OPRF-obfuscated IDs with batch PIR. During ShareRetrieval, the client calls batch PIR on masked databases so that the items it fetches form additive secret shares with the server's masks. Decode post-processes the output from ShareRetrieval in 2PC as required by batch PIR.

Algorithm 1: DEDUPLICATION

Deduplication$(\llbracket x_1 \rrbracket, \dots, \llbracket x_b \rrbracket)$:

1: *// Sort*
2: $C_S \leftarrow \mathsf{Sort}(\llbracket x_1 \rrbracket, \dots, \llbracket x_b \rrbracket)$
3: *// Placeholder Replacement*
4: **for** $i \in \{2, \dots, b\}$ **do**
5: $\quad \llbracket t_i \rrbracket \leftarrow \llbracket x_i \rrbracket \neq \llbracket x_{i-1} \rrbracket$
6: $\quad \llbracket x_i^D \rrbracket \leftarrow \mathsf{MUX}(\llbracket t_i \rrbracket, \llbracket N + i \rrbracket, \llbracket x_i^D \rrbracket)$
7: **return** $((\llbracket x_1^D \rrbracket, \dots, \llbracket x_b^D \rrbracket); (\llbracket t_2 \rrbracket, \dots, \llbracket t_b \rrbracket); C_S)$

whose selection is detailed in Section B.1. FABLE uses the cuckoo table building procedure CuckooBuild, which sequentially inserts $x_1, \dots, x_p$ into a size-q array ($q > p$) with hash functions $h_1, \dots, h_w$ using cuckoo hashing. Specifically, for each item x_i, we check the slots $h_1(x_i), \dots, h_w(x_i)$: x_i is inserted into the first available slot. If no slots are available, x_i will be placed into a random slot from them, and the original item in that slot is re-inserted using the same procedure.

Deduplication. As discussed in the previous section, input queries in FABLE need to be deduplicated before query translation because the cuckoo hashing used in the batch PIR protocol does not support duplicates in the input. Specifically, having more than $w+1$ copies of an item can violate correctness due to the Pigeonhole Principle. In addition, duplicates reveal input frequency patterns, as identical inputs yield the same OPRF output. Hence, we perform Deduplication at the beginning of FABLE to remove the duplicates in secret inputs $x_1, \dots, x_b$ to obtain deduplicated inputs $x_1^D, \dots, x_b^D$.

The oblivious algorithm for deduplication has been proposed in Snoopy [22], an oblivious storage system based on oblivious sort and compaction [31]. However, when directly adapted to 2PC settings, the algorithm leads to significant communication overhead. Therefore, we introduce multiple optimizations to reduce communication cost by 14.0% (see Appendix E for detailed estimation). To deduplicate an array of length b, Snoopy's algorithm follows an extend-then-compact approach: The input array is obliviously sorted to cluster identical inputs and extended with b placeholders. All duplicates in input and some placeholders are removed by an oblivious compaction, such that the result is still a size-b array. The bottleneck is the oblivious sort and compaction, which rely heavily on comparisons and conditional swaps. While sorting is necessary, we avoid compaction in Deduplication by replacing duplicates with placeholders *in place*. The formal algorithm description is in Alg. 1.

We also observe that the intermediate sorting results can be cached to improve the performance of later protocols. As the comparison order of Bitonic sort on a size-p array only

depends on p, we denote the comparison order generator as $\mathsf{SortOrder}(p)$ whose formal definition is in Appendix A. Hence, Bitonic Sort can be viewed as a loop that swaps x_i and x_j if $x_i > x_j$ for every index-pair $i, j \in [p]$ in $\mathsf{SortOrder}(p)$. During this process, the comparison results C can be cached. By running another Sort with comparison results from C, we can reproduce the ordering on another size-p array without any comparison gate. During Deduplication, we record the context C_S for later reuse in Expansion.

OPRFEvaluation. After Deduplication filters out all input duplicates, we proceed to *query translation*. Since the deduplicated input queries $x_1^D, \ldots, x_b^D$ are in 2PC, the client cannot directly translate these queries to be used in batch PIR. FABLE solves this using OPRFEvaluation, in which we encode the input queries into obfuscated inputs using OPRF f_K that is jointly executed by the server and the client. The server samples and provides the OPRF key K, but it does not learn anything about the input queries or the translated queries. The client only learns a set of pseudorandom IDs $\mathsf{id}_1^D, \ldots, \mathsf{id}_b^D$, which are the OPRF encodings of $x_1^D, \ldots, x_b^D$. Once the client receives these IDs, it can proceed to run local processing to translate queries. Note that the OPRF key should be resampled for each batch. Otherwise, the client can learn whether an incoming batch contains a query that was included in a previous batch by comparing the IDs with previous ones.

ServerSetup. OPRFEvaluation obfuscates the input so that the client can prepare the query and invoke batch PIR. As the OPRF key is freshly resampled each time, the server has to prepare the database with the new OPRF key for every batch so that the client can retrieve from the database using its IDs. As illustrated in the upper left corner of Figure 3, ServerSetup consists of three steps: ③-A OPRF Preparation, ③-B Database Arrangement, and ③-C Database Masking.

During ③-A. OPRF Preparation, the server pads the LUT with $b-1$ zeros to ensure that PIR can properly handle the dummy inputs added during Deduplication. Then the server encodes all LUT keys from δ-bit numbers to OPRF-encoded indices: the i-th item originally indexed by $i \in [N]$ becomes indexed by $f_K(x)$ (f_K is the OPRF), so that the client can use those IDs to look up the LUT. This step is parallelizable, and our evaluation shows it is efficient in practice.

After this step, the server arranges the database (③-B. Database Arrangement) following batch PIR's approach: Each LUT item with key $f_K(x)$ is assigned to w buckets with indices $(h_1^B(f_K(x)), \ldots, h_w^B(f_K(x)))$. Inside each bucket, each item with key $f_K(x)$ is assigned to a position out of w possible positions $(h_1^P(f_K(x)), \ldots, h_w^P(f_K(x)))$ using cuckoo hashing. For instance, the first item $(f_K(1), 2)$ is routed to buckets $h_1^B(f_K(1)), \ldots, h_w^B(f_K(1))$ and the same for the rest items. Inside each bucket, each item with ID $f_K(x)$ is then put at one of the positions $h_1^P(f_K(x)), \ldots, h_w^P(f_K(x))$ with cuckoo hashing. Since the client needs to retrieve w items from each bucket during PIR by keywords, the server stores the key x together with the value $T(x)$ so that the client can query correctly.

Algorithm 2: SERVER SETUP

$\mathsf{ServerSetup}(T)$:

1: *// OPRF Preparation.*
2: $k_{oprf} \xleftarrow{\$} \mathcal{K}$
3: **for** $x \in [N]$ **do**
4: $\quad \mathsf{id}_x \leftarrow \mathsf{OPRF.EvaluateLocal}(x; k_{oprf})$
5: *// Database Arrangement.*
6: $C_1, \ldots, C_B \leftarrow \mathsf{Encode}((\mathsf{id}_1, \ldots, \mathsf{id}_N); (h_1^B, \ldots, h_w^B))$
7: **for** $j \in [B]$ **do**
8: $\quad$ *// $C_j = (\mathsf{id}_{j_1}, \ldots, \mathsf{id}_{j_s})$ where s is the bucket size*
9: $\quad (\mathsf{id}_{j_1^B}, \ldots, \mathsf{id}_{j_{N_B}^B}) \leftarrow \mathsf{CuckooBuild}(C_j; (h_1^P, \ldots, h_w^P))$
10: $\quad C_j \leftarrow (j_1^B \| T(j_1^B), \ldots, j_{N_B}^B \| T(j_{N_B}^B))$
11: *// Database Masking.*
12: **for** $k \in [w]$ **do**
13: $\quad$ **for** $j \in [B]$ **do**
14: $\quad\quad r_{j,k}^I \leftarrow \{0,1\}^\delta$
15: $\quad\quad r_{j,k}^O \leftarrow \{0,1\}^\sigma$
16: **for** $k \in [w]$ **do**
17: $\quad$ **for** $j \in [B]$ **do**
18: $\quad\quad C_j^k \leftarrow ((j_l^B \oplus r_{j,k}^I) \| (T(j_l^B) \oplus r_{j,k}^O), \forall j_l^B \| T(j_l^B) \in C_j)$
19: $\quad\quad$ *// For empty slots in C_j, we fill $r_{j,k}^I \| r_{j,k}^O$ in C_j^k.*
20: $\quad \mathsf{db}_M^k \leftarrow (C_1^k, \ldots, C_B^k)$
21: **return** $(k_{oprf}; \{r_{j,k}^I, r_{j,k}^O : k \in [w], j \in [B]\}; \{\mathsf{db}_M^k : k \in [w]\})$

To this end, the server has prepared the database for batch PIR. However, the client should not fetch the items in plaintext as that would compromise LUT confidentiality. In our PIR by keywords construction, the client fetches w items from each bucket, and all wB items it fetches should be obfuscated by different masks. The client does so by making w separate batch PIR calls, where each call retrieves B items. Therefore, for each PIR call, the server generates a different random mask for each bucket, with a total of wB unique random masks $(r_{jk}^I, r_{jk}^O), \forall j \in [B], \forall k \in [w]$. As illustrated in ③-C. Database Masking, the wB random masks are arranged in a matrix of size $B \times w$. Each batch PIR call uses B masks, and each bucket uses a unique mask. Therefore, the wB masked items the client fetches form additive secret shares of the LUT items with the wB random masks held by the server, and the client does not learn the lookup value in plaintext.

ShareRetrieval. At this point, the server has properly prepared the database in ServerSetup, and the client has properly translated queries in OPRFEvaluation. The client can now use batch PIR to fetch items from the server's database. This step is the core protocol for producing LUT lookup results.

As illustrated in the top right portion in Figure 3, ShareRetrieval consists of four steps, which are subprotocols of batch PIR with minor modifications for the 2PC setting. The first step ④-A. GenAssignPerm takes b input IDs $\mathsf{id}_1^D, \ldots, \mathsf{id}_b^D$ from OPRFEvaluation and assigns each query

Algorithm 3: DECODE

$\mathsf{Decode}(\{[\![x^B_{j,k}]\!], [\![v^B_{j,k}]\!] : k \in [w], j \in [B]\}; \pi, C_S)$:

1: $C_\pi \leftarrow \mathsf{GenContext}(\pi(1), \ldots, \pi(B))$
2: $[\![x^B_1]\!], \ldots, [\![x^B_B]\!] \leftarrow \mathsf{Permute}((\![x^D_1]\!], \ldots, [\![x^D_b]\!], 0, \ldots, 0); C_\pi)$
3: **for** $j \in [B]$ **do**
4: $\quad [\![v^B_j]\!] \leftarrow \bigoplus_{k\in[w]} \mathsf{MUX}([\![x^B_{j,k}]\!] = [\![x^B_j]\!], 1, 0)\, [\![v^B_{j,k}]\!]$
5: $[\![v^D_1]\!], \ldots, [\![v^D_B]\!] \leftarrow \mathsf{InvPermute}((\![v^B_1]\!], \ldots, [\![v^B_B]\!]); C_\pi)$
6: **return** $([\![v^D_1]\!], \ldots, [\![v^D_b]\!])$

to a bucket. The client also records bucket assignment π – a map from batch index to bucket index – which is later used for Decode. For every $k \in [w]$, the client constructs and sends a PIR query $\vec{q}_k$ to the server to retrieve items located at positions $h^P_1(\mathsf{id}^D_1), \ldots, h^P_w(\mathsf{id}^D_b)$ from the corresponding buckets. For $\vec{q}_k$, the server generates a PIR response $\vec{a}_k$ with db^k_M as the database, from which the client extracts the masked items (a^I_{jk}, a^O_{jk}) it requests. The masks (r^I_{jk}, r^O_{jk}) held by the server and masked items (a^I_{jk}, a^O_{jk}) extracted by the client are then viewed as *additive 2PC shares of the lookup results*. This step is formally described in Alg. 6 in Appendix D.

Decode. After ShareRetrieval, the client and the server obtain secret shared values, but these values do not directly correspond to results for the original input queries. There are two reasons for this. First, since we use cuckoo hashing-based PIR by keywords, the client retrieves *extra unnecessary* items (denoted using $(*,*)$ in Figure 3), which need to be filtered out. Second, after a filter is applied, there are still B responses, but we need to further remap these back to b responses. FABLE uses Decode to solve these two problems. Since the outputs from ShareRetrieval are secret shared, we need efficient 2PC protocols for filtering and remapping.

Recall that during ShareRetrieval, the client obtains the query assignment map π from batch index to bucket index. We derive three primitives from sorting: GenContext, which generates the sorting context by recording the comparison result during a local Bitonic sort; Permute and InvPermute, which apply a map or its inverse over a 2PC array given the sorting context. They are formally defined in Appendix A. To remove unnecessary items in the responses, the client first generates the context of π with GenContext and shares the context in 2PC. Given the deduplicated inputs $x^D_1, \ldots, x^D_b$ in 2PC, the server and the client simulate GenAssignPerm in 2PC (⑤-A. Query Permutation) to reconstruct queries for each bucket $x^B_1, \ldots, x^B_B$, by applying π over $x^D_1, \ldots, x^D_b$ with Permute. Then, picking the correct response from each bucket becomes trivial (⑤-B. Select): for each bucket j, out of w responses $(x^B_{j1}, v^B_{j1}), \ldots, (x^B_{jw}, v^B_{jw})$, there is only one item (x^B_{jk}, v^B_{jk}) satisfying $x^B_{jk} = x^B_j$, and the corresponding value v^B_{jk} is the desired response v^B_j. Formally, we evaluate $v^B_j = \sum_{k\in[w]} \mathsf{MUX}(x^B_{jk} = x^B_j, v^B_{jk})$ in 2PC.

To remap the B responses for each bucket to b responses for each input (⑤-C. Response Recovery), the server and the client apply π^{-1} on $v^B_1, \ldots, v^B_B$ in 2PC by InvPermute to route the response from each bucket to the corresponding input index $x^D_1, \ldots, x^D_b$. The results are $v^D_i = T(x^D_i)$ for $i \in [b]$. The formal algorithm description of Decode is in Alg. 3.

Expansion. The output from Decode is the LUT values corresponding to the deduplicated inputs. However, since the original input queries may have duplicates, we need to run Expansion, a *reverse* of Deduplication. Expansion reconstructs the secret-shared output by removing dummy items and expanding the table values of the "deduplicated" input.

Same as Deduplication, Snoopy [22] presents an oblivious algorithm for Expansion, which, however, is expensive when translated directly into 2PC settings due to network overhead. Snoopy only uses the original inputs as auxiliary data, but we observe that the cost can be greatly reduced using cached context during Deduplication, because accessing previously computed secret shares is almost free in 2PC while re-comparison is expensive. In particular, the placeholder replacement is inverted by reusing the tag array $[\![t_2]\!], \ldots, [\![t_b]\!]$ to populate all "duplicate" slots with a replica of the preceding value. The sorting is inverted by InvPermute with the recorded context C_S in Deduplication. The formal description can be found in Alg. 7 in Appendix D. These optimizations result in 84.1% fewer gates.

4.3 Discussion of LUT Confidentiality

In this section, we discuss how LUT confidentiality is ensured in FABLE. In all procedures involving the client, ShareRetrieval is the only one that takes LUT as input, with the PIR response being the sole message sent to the client. Thus, our goal is to ensure that the PIR response does not leak information about the LUT's content. In state-of-the-art batch PIR protocols, the PIR response is a list of (R)LWE homomorphic encryption ciphertexts. After decryption, the client obtains the encrypted values and the ciphertext noise.

The encrypted values in the PIR response are the LUT items masked by the server-generated masks. On each masked database DB^k_M, the client retrieves one item per bucket, while each bucket j uses a different random mask (r^I_{jk}, r^O_{jk}). Therefore, each item requested by the client is hidden by a different mask. Thus, from the client's perspective, all items are independent random values. However, the magnitude of the ciphertext noise can also leak information about the database, because the HE circuit (PIR.Answer) has dependency on the input LUT, whose content affect the encryption noise. To avoid this leakage, we adopt a standard noise flooding technique [5], which adds an encryption of zero with a noise larger than the existing noise by an exponential factor to the ciphertext to smudge out the original noise. Specifically, to get s-bit of statistical security, the added noise should be $n_f = s + \log_2 n + \log_2 \alpha$ bits larger than the upper bound of

the original noise, where α is the number of ciphertexts and n is the polynomial modulus degree [19].

5 Implementation

We use PIRANA [49] for our batch PIR scheme as it is currently state-of-the-art and makes the best tradeoff between communication and computation. As we are unable to find an open-source implementation of PIRANA, we implement the protocol according to their paper using the BFV scheme in Microsoft SEAL (version 4.1.1) [60], with polynomial modulus degree $n = 8192$, plaintext modulus bits $\log p = 18$, and coefficient modulus bits $\log q \approx 218$. We set the hamming weight for the constant weight code as 2. We use modulus switching at the end of PIR.Answer to reduce communication costs and add multi-threading support for the server with OpenMP [55]. We select LowMC [2] as the underlying OPRF with 64-bit blocks, 2^{64} allowed data complexity, 128-bit keys, eight S-boxes per layer, and 24 rounds. The data complexity refers to the number of plaintext-ciphertext pairs required to perform a successful differential cryptanalysis attack. To implement LowMC and 2PC primitives, we use the Garbled Circuits from the Secure and Correct Inference (SCI) library of EzPC [18]. We set the correctness parameter (in bits) $\lambda = 40$, $B = n = 8192$, and $w = 3$. To estimate the BFV noise growth, we adopt a general worst-case analysis [39] and a tighter but more constrained (requires the independent input) average-case analysis [9] when applicable.

6 Evaluation

In this section, we demonstrate FABLE's significant performance gains over existing 2PC LUT protocols for batched queries, both theoretically and empirically. We implement FABLE in 6.5K lines of C++ code and evaluate FABLE over various network settings, LUT sizes, and batch sizes.

Experimental Setup. We consider three choices of the input bits δ: 20, 24, and 28 with LUT sizes $N = 2^{\delta}$. For baselines, we consider two state-of-the-art LUT protocols in MPC (SP-LUT+ [14], FLUTE [14]) and two in DORAM (FLORAM [24], 2P-DUORAM [65]); all have been introduced in Section 3.1. Note that we do not split a large LUT into smaller LUTs, which is not friendly for normal users and will leak information about the confidential function represented by the LUT as discussed in Section 1. Since the performance of FABLE is independent of the table content, evaluating a single LUT is sufficient for performance evaluation. Additionally, all baselines' running time is independent of the LUT content except FLUTE, and FLUTE's running time is not reported because it takes too long to run in our configurations (e.g., evaluating a single query with a 24-bit LUT takes over 2 days without any network constraint). We fill the LUT content with the Gamma function Γ over $[1,4]$. We conduct our experiments on two `c6i.16xlarge` AWS EC2 instances with 32 physical cores. We simulate the following network settings using Linux Traffic Control (tc): 1) LAN: 3Gbps rate with 0.8ms one-way latency; and 2) WAN: 100Mbps rate with 80ms one-way latency. We repeat each experiment *five* times and report the average values. Experiments under additional network settings are included in Appendix F.

Table 3: Comparison of asymptotic complexity.

Metric	Computation		Communication	Round
Party	Client	Server	-	-
FABLE	$O(\sqrt{N})$	$O(N)$	$O(\sqrt{N})$	4
SP-LUT+	$O(\log N)$	$O(\sigma N)$	$O(\sigma N)$	4
FLUTE	$O(N^2)$	$O(N^2)$	$O(N)$	6
FLORAM-CPRG	$O(N)$	$O(N)$	$O(\sigma \log N)$	$O(\log N)$
2P-DUORAM	$O(N)$	$O(N)$	$O(\sigma \log N)$	4

6.1 Asymptotic Complexity Analysis

We compare the asymptotic computation, communication, and round complexity of FABLE with baselines in Table 3. For simplicity, we report the complexity with respect to LUT size N as it is the most significant variable. Among all procedures in FABLE, Deduplication, OPRFEvaluation, Decode, and Expansion work on a batch of size b or B, so their complexity is not related to N. ServerSetup is a local procedure on the server side with a linear scan of the table, which is $O(N)$; ShareRetrieval is a variant of PIRANA, which is $O(N)$ for the server and $O(\sqrt{N})$ for the client. Therefore, FABLE is linear to the table size N on the server side and sublinear on the client side. A detailed asymptotic analysis including all variables can be found in Appendix C.

With respect to table size, SP-LUT+ achieves sublinear client computation but requires linear communication; FLUTE has linear communication with lower constant but superlinear computation; FLORAM and 2P-DUORAM have sublinear communication, but the client's computation cost is linear; FABLE is the only one that achieves *both* sublinear client computation and communication.

6.2 Microbenchmarks

In this section, we evaluate each component of FABLE, investigate how lightweight the client is, and the scalability with respect to table sizes. We also benchmark FABLE with various thread numbers to evaluate its parallelizability.

FABLE performance breakdown. Table 4 reports the end-to-end breakdown of FABLE running on a batch of 4096 queries over LUTs of various sizes with 32 threads. We report ShareRetrieval in terms of 4 sub-procedures: Query, Answer, Extract, and share conversion, which converts the Boolean sharing into GC sharing for later 2PC algorithms. For Query and Answer, we report both total and local computation time.

Table 4: End-to-end performance breakdown. The performance of FABLE with AES as the OPRF is listed in parentheses. The table has $\delta = \sigma$ and the batch size is $b = 4096$. The server uses 32 threads. The components are listed following the execution order, with the parties involved marked on the left. For Query and Answer, we report the computation time and the communication time separately. We mark a 2PC procedure with *(both)*, while *(server)* and *(client)* indicate the procedure is executed locally by the server and client respectively. The client time and the server time mean the total time of procedures involving the client or the server respectively, including local computation and the 2PC procedure.

LUT Size		20-bit			24-bit			28-bit		
Metric		Time (sec)		Comm. (MB)	Time (sec)		Comm. (MB)	Time (sec)		Comm. (MB)
Network		LAN	WAN	-	LAN	WAN	-	LAN	WAN	-
(both)	Deduplication	0.66	18.73	219.75	0.77	22.11	259.75	0.91	25.48	299.75
(both)	OPRFEvaluation	0.49 (5.53)	6.13 (162.49)	72.25 (1921.54)	0.47 (5.47)	6.10 (162.48)	72.25 (1921.54)	0.48 (5.47)	6.14 (162.50)	72.25 (1921.54)
(server)	ServerSetup	0.26 (0.17)	0.26 (0.19)	-	5.05 (3.96)	5.02 (3.93)	-	99.56 (79.85)	99.69 (79.32)	-
(client)	Query-Compute	0.78	0.78	-	2.70	2.70	-	16.61	16.60	-
(both)	Query-Comm.	0.14	5.21	51.69	0.49	15.82	177.91	2.00	61.38	714.36
(server)	Answer-Compute	2.26	2.31	-	27.62	27.60	-	532.31	532.05	-
(both)	Answer-Comm.	0.00	0.14	0.96	0.00	0.12	0.96	0.00	0.11	0.82
(client)	Extract	0.02	0.02	-	0.02	0.02	-	0.02	0.02	-
(both)	Share conversion	0.17	4.41	30.88	0.19	4.90	36.88	0.22	5.39	42.88
(both)	Decode	1.54	43.62	508.56	1.89	51.91	605.56	2.17	60.24	702.56
(both)	Expansion	0.30	8.47	100.00	0.38	10.14	120.00	0.43	11.84	140.00
	Client Time	3.95 (8.95)	82.16 (238.45)	-	6.41 (11.24)	97.88 (254.08)	-	20.84 (25.81)	125.70 (282.30)	-
	Server Time	5.67 (10.63)	83.93 (240.11)	-	36.37 (40.43)	127.78 (283.24)	-	636.08 (633.86)	740.83 (888.96)	-
	End-to-End	6.61 (11.70)	90.07 (246.13)	984.08 (2833.37)	39.58 (45.85)	146.44 (303.92)	1273.30 (3122.59)	654.71 (699.02)	818.94 (1014.08)	1972.62 (3821.91)

The running time bottleneck of FABLE is the server computation in ShareRetrieval, i.e., Answer and ServerSetup, especially when the LUT is large. They are the only phases with linear computation costs in the LUT size, consistent with empirical observations in Table 4. The running time of ServerSetup is about 20% of Answer, implying that the cost of additional protocols for batching compared to batch PIR calls is minimal. Consequently, given the $w = 3$ calls to batch PIR, FABLE's total cost is only slightly more than w times a single batch PIR. Also, we observe that the client's running time is significantly shorter than the server's under LAN, indicating a much lower computational cost for the client and highlighting its lightweight nature. In addition, the communication scales well with the table size N as the dominant component is the 2PC protocols with *logarithmic* communication to N. Such scalability helps maintain a decent communication cost even for a large LUT with $N = 2^{28}$.

Performance with AES as the OPRF. Prior work has shown that LowMC with small parameters is prone to chosen plaintext attacks [7, 47, 48]. We believe that applying these attacks to FABLE would be more difficult in our setting since the client does not *directly* choose inputs to the OPRF (though the client can influence these inputs), and the client does not learn the inputs either since they are secret shared.

Nevertheless, we evaluate AES-128 as an alternative and show the performance breakdown in Table 4. The higher multiplicative complexity of AES-128 leads to 27× more communication in OPRFEvaluation, but the hardware-efficient implementation of AES-128 allows for faster ServerSetup. Since OPRFEvaluation and ServerSetup are not the bottleneck of FABLE, the end-to-end slowdown of using AES-128 compared to using LowMC is 1.07-2.73×.

Running time with different thread numbers. To evaluate FABLE's scalability with more computational resources, we evaluate the end-to-end running time with varying *server-side* thread counts, shown in Figure 4. Compared to the single-threaded running time, FABLE with 32 threads can be accelerated by 20× in LAN and 6× in WAN. The improvement is less significant in WAN, as the communication time cannot benefit from multithreading.

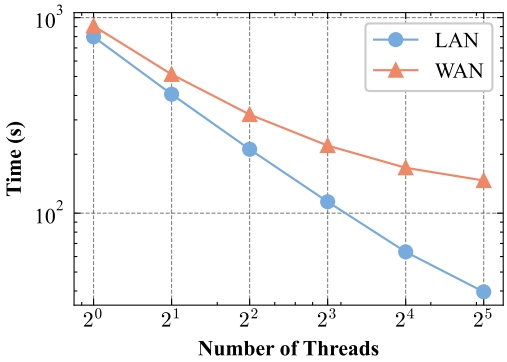

Figure 4: End-to-end running time of FABLE with different thread numbers, with $\delta = \sigma = 24$ and $b = 4096$.

6.3 End-to-End Performance Analysis

In this section, we compare the end-to-end performance of FABLE against existing protocols.

Setup for LUT protocols in MPC. The state-of-the-art 2PC LUT protocols [14, 23, 40] are all communication bounded. We choose SP-LUT+ and FLUTE as baselines because their overall communication is the best among the prior works. SP-LUT+ is SP-LUT instantiated with the silent OT exten-

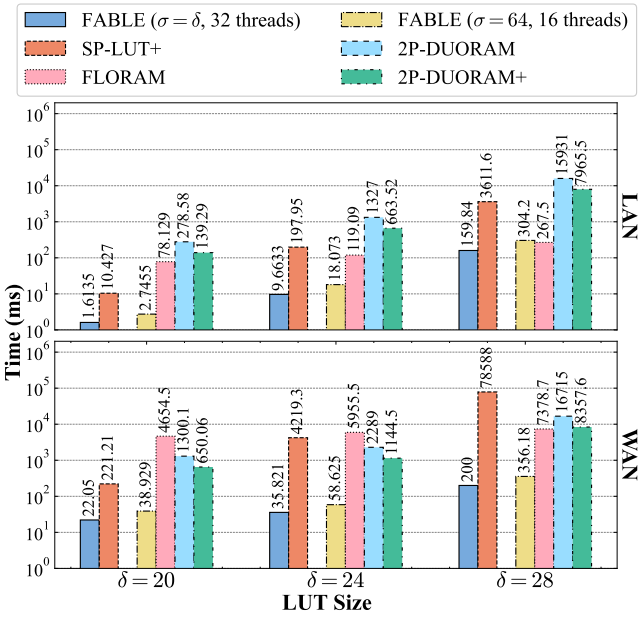

Figure 5: Amortized running time with respect to various table sizes and network settings.

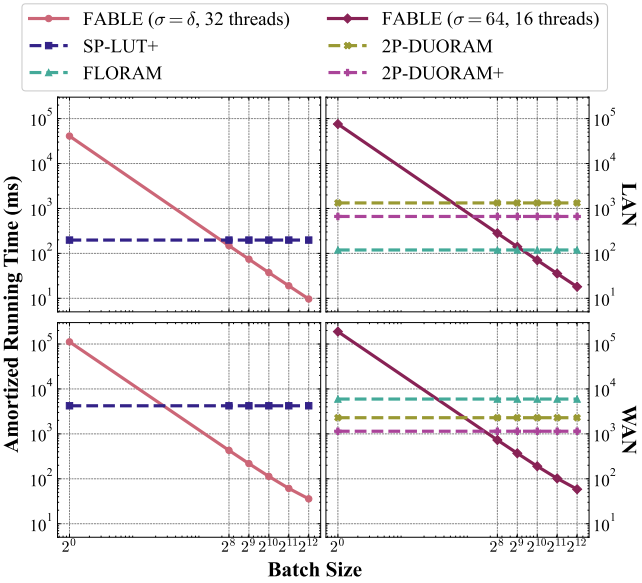

Figure 6: Amortized running time with respect to various batch sizes and network settings. The table size is 2^{24}.

sion [11], which we implement based on libOTe [57]. SP-LUT+ has much better asymptotic computation complexity and therefore its computation is more scalable with table sizes compared to FLUTE, while FLUTE has better communication both asymptotically and practically. While prior works [14, 23] additionally propose to split large LUTs into smaller LUTs, it involves a complex toolchain with commercial software and is therefore difficult to reproduce. Additionally, this method requires revealing the connection patterns of smaller LUTs and does not provide LUT confidentiality. Therefore, we scaled their protocol to the target size for benchmarking. We use the same LUT for the Gamma function and set 32 threads for the server. Following SP-LUT+'s settings, we set the LUT output bits to the same as the input ($\sigma = \delta$). Our experiments show that FLUTE takes almost two days to finish with a 24-bit LUT even in the LAN network setting. Thus, we do not report the running time of FLUTE.

Setup for DORAM. For DORAMs, we compare FABLE with FLORAM, 2P-DUORAM, and 2P-DUORAM+. We benchmark the baselines' performance using the scripts provided by DUORAM and estimate 2P-DUORAM+'s cost from 2P-DUORAM's cost. As the scripts are run on a single machine, we use 16 threads (half the cores) for FABLE for fair comparison. We set the LUT output bits $\sigma = 64$ conforming to the fixed word size in DUORAM and report the average performance over 128 independent reads for each scheme.

Results. We report the amortized performance over a batch of inputs for both running time and communication in Figure 5, Figure 6, and Figure 7.

Figure 5 shows the amortized running time under various table sizes and a fixed batch size of 4096. FABLE significantly surpasses SP-LUT+ in all configurations, with a speedup of 6-53× in LAN and 10-393× in WAN. The improvement is more significant in slow networks and larger LUT sizes because of the asymptotic and concrete communication advantage of FABLE (see Table 3). FABLE surpasses 2P-DUORAM in all settings with an improvement of 51.73-101× in LAN settings and 33-50× in WAN. Similarly, FABLE surpasses 2P-DUORAM+ in all settings with an improvement of 26-51× in LAN and 17-25× in WAN. The advantage is from the baselines' heavy server-side computation, which scales linearly with batch sizes. FABLE scales better since it is designed with batching workloads in mind. Compared with FLORAM, FABLE achieves a 0.9-28× speedup under LAN and a 11-120× speedup under WAN. FABLE is slightly slower than FLORAM in the LAN setting with a 28-bit LUT, because FLORAM is designed to have lightweight server-side computation at the cost of more communication and heavier client-side computation. Recall that FABLE with AES is 2.7× slower than with LowMC, but this gap is insignificant relative to its advantage over baselines. Thus, FABLE remains highly competitive even with AES as the OPRF.

Figure 6 shows the amortized running time under various batch sizes. FABLE is the only protocol whose running time can be amortized with larger batches. Although FABLE performs worse than existing protocols with a single query, its advantage is pronounced when multiple queries are batched. It beats SP-LUT+ when the batch size goes beyond 2^8 and surpasses DORAM protocols with a batch larger than 2^{10}.

Figure 7 shows the amortized communication with various batch sizes and table sizes. FABLE's communication scales

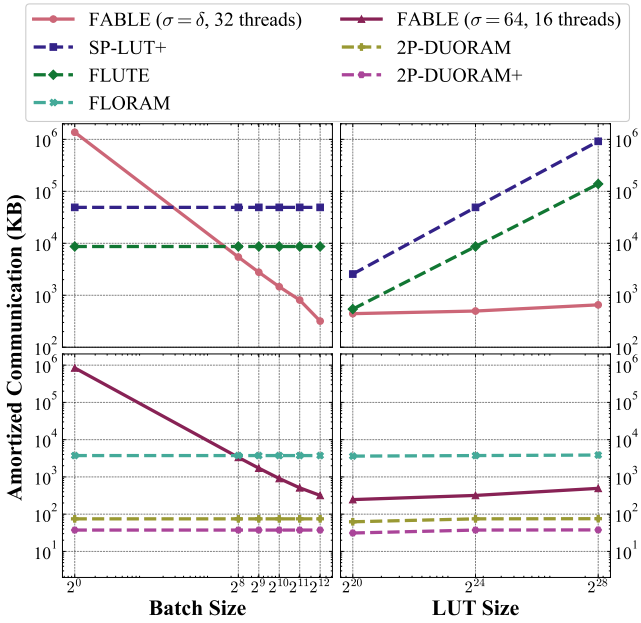

Figure 7: Amortized communication. The LUT size is fixed at 2^{24} when the batch size varies. The batch size is fixed at $b = 4096$ when varying the table sizes.

similarly to the runtime: it falls behind with small batches but becomes advantageous with larger batches. 2P-DUORAM and 2P-DUORAM+ are the only protocols with better communication than FABLE even with batching, but their heavy computation still leads to worse amortized running times. FABLE's communication scales well with LUT sizes, conforming with the communication-efficient DORAM schemes.

6.4 Applications

To demonstrate how FABLE can be used in 2PC applications, we evaluate two use cases introduced in Section 1.

Secure embedding lookup. As introduced in Section 1, an end-to-end secure inference workload usually consists of multiple steps. In our example, a client's input tokens are first sanitized (e.g., using another ML model), then converted to embeddings, and finally the model inference is executed. All procedures are in 2PC, so the input and output for the embedding lookup are both secret shared. To model a real use case for word embedding lookup, we select the text-sentiment classification benchmark in Meta's Crypten framework [43], which computes the sum of embeddings of all words in a sentence. The input sentence is first converted to a word-count vector and multiplied by the embedding matrix. Following Crypten's convention, we feed 32 sentences in a batch, with 16 words per sentence. The word embedding has 32-dimensional embeddings of 519,820 words with 16-bit fixed-point elements. The official Crypten implementation generates multiplication triples using a trusted third party, so we estimate the multiplication triples generation time by MOTION [13]. In total, the baseline takes 12,868s (109,774s) with the LAN (WAN) network. Appendix G.1 shows a detailed breakdown. The same functionality implemented by a confidential LUT evaluation with FABLE plus a summation takes only 28.22s (617.59s), 456× (178×) faster than Crypten's solution. The superior performance is attributed to FABLE's usage of the efficient batched PIR with sublinear communication for table lookup. Unlike FABLE, Crypten must traverse the LUT with communication linear in the embedding size.

Secure query execution. For the secure SQL application, we base our benchmark on the TPC-H benchmark [61]. TPC-H is a business warehouse workload that defines both data schema as well as a set of queries. The tables consist of many tables, where some tables have millions of rows (e.g., `orders`, `lineitem`) and other tables (e.g., `customer`, `region`) are much smaller. While TPC-H was not designed as a 2PC benchmark, its workload can still be adapted to a 2PC setting. Suppose a financial institution holds the `customers` and their geographic data `region`, while a payment company (e.g., Stripe) holds the tables `orders` and `lineitem` but not detailed customer information for privacy reasons. Using 2PC, these parties can securely collaborate on queries to gain business insights. In this setup, the financial institution acts as the client and the payment company as the server.

FABLE can accelerate equi-joins where one table is the output of a subquery executed jointly by the two parties with selective filters, and the other table is a large table owned by the server. This pattern is fairly common and can be found in 18% queries in TPC-H. We use a query simplified from TPC-H query 8 (see Appendix G.2). It is a multi-way join across many tables that span both parties, with selective filters on `region` and `orders`. As `lineitem` has millions of rows and there are filters on both `region` (and thus `customers`) and `orders`, the best execution plan in 2PC is for the client to first locally join `customers` with `region` and `nation`, and then join it with a filtered `orders` (executed jointly by both parties), and finally join with the largest table `lineitem`.

Concretely, we assume that the filters are selective and that the join results between `customers` and the filtered `orders` is a 2PC table with 4096 rows, and `lineitems` has 1 million rows. Senate [58] adopts a sort-compare-shuffle circuit to execute equi-join. As Senate is built upon garbled circuits with malicious security, we reimplement the circuit in the garbled circuits provided in SCI with semi-honest security. Our reimplementation of Senate takes 46.87GB communication to execute this join, leading to 149s (4228s) total running time under the LAN (WAN) network.

With FABLE, the join with `lineitem` can be accelerated by viewing the table in 2PC as a batched query. The total time with FABLE is 10s (155s), 15× (27×) faster than the baseline. FABLE also has better scalability. With respect to the table size, Senate's SCS circuit with a Bitonic merge has superlinear computation and communication complexities, while FABLE

has linear computation and sublinear communication. Given that `lineitems` can contain hundreds of millions of rows in practice, FABLE's advantage becomes even more evident.

7 Related Work

Secure two-party computation. Secure two-party computation techniques have been widely studied and are useful in many data-sensitive cryptographic applications [14, 23, 24, 26, 56, 59]. The specific functionality we focus on is LUT evaluation over a batch of inputs X, defined in Section 2. Private Set Intersection (PSI) [26] is a similar technique that aims to find the intersection of two sets X_1 and X_2 separately held by the two parties. It differs from our setting in that the client knows the plaintext contents of their respective sets. Note that PSI can also be realized from batch PIR [34].

Secure Lookup Table Evaluation. The use of multi-input/output gates (a.k.a. "lookup tables") to accelerate the communication-bounded MPC was formally explored by Ishai et al. [40], who introduced the one-time truth table (OTTT) protocol to privately evaluate entire circuits as LUTs. At a high level, their idea involves both parties holding secret shares of a random rotation value and the underlying LUT rotated by this rotation value during preprocessing. Extending this idea, Dessouky et al. [23] proposed LUT-based circuit synthesis that uses LUT as gates in circuits, and derived OP-LUT and SP-LUT, two protocols for LUT evaluation in the semi-honest 2PC setting. Further constructions and applications [21, 42] are extended into general secret-sharing-based MPC (SS-MPC). Thereafter, FLUTE [14] progressed the state of the art in the semi-honest setting by improving online and overall communication. These schemes are important baselines for FABLE. In addition to the SS-MPC approaches, another orthogonal line of work [35, 36] focuses on LUT evaluation in a single garbled circuit. Compared to SS-MPC approaches, they are non-interactive, but they lack practical evaluation and, theoretically, have higher overall communication costs than FLUTE.

Private Information Retrieval (PIR). PIR was introduced in the multi-server setting [20], where a client interacts with a set of non-colluding servers holding replicas of the queried database. Soon after, [45] presented a computationally secure single-server construction using additively homomorphic encryption. With subsequent works yielding lightweight constructions in the multi-server setting [12, 28], significant effort has been made towards making PIR concretely efficient in the single-server setting [4, 38, 49, 50]. Though initial works focused on stateless PIR, where clients do not store any information before sending a query, a rich class of stateful PIR constructions [38, 50] has offered sublinear improvements on the server-side processing cost by allowing clients to download a large database-dependent hint. However, stateful PIR is challenging to adapt to our setting as our database is freshly masked for every new input batch, so a new hint has to be downloaded before each query. Another area of optimization came in the batched setting, introduced in [41] with batch codes and further improved by probabilistic variants with reverse hashing [4] and homomorphic encryption [49, 51]. They are a building block of FABLE.

Oblivious RAM. Rather than providing access-pattern privacy to many users accessing a public database, as in PIR, oblivious RAM (ORAM) [30]—a related primitive—prioritizes a single client accessing a private database. ORAM schemes usually incur sublinear server-side work and bandwidth [67], leading to asymptotic improvements over PIR, though failing to offer the same multi-client capability. Extending ORAM to the PIR setting [16] has shown promising results. However, it introduces latency that usually depends on the number of clients, such as total computation scaling with the number of clients or requiring additional communication between the clients themselves.

Generic MPC compilers can be used to adapt ORAM to 2PC settings [67]. Distributed ORAM (DORAM) [24, 65] is a variant of ORAM, where multiple non-colluding servers achieve ORAM functionality in MPC. Several DORAM works, such as FLORAM [24], are inspired by multi-server PIR [28], where the parties can jointly and securely evaluate the ORAM. Subsequent literature has further optimized DORAM in the 2PC setting [65].

8 Conclusion

We propose FABLE, a novel protocol for confidential LUT evaluation in 2PC, optimized for large-scale LUTs with batch workloads. FABLE incorporates recent advances in batch PIR and tailors them to the stringent security requirements of 2PC with specially designed 2PC protocols. Our approach delivers exceptional efficiency, with amortized performance that is an order of magnitude faster than state-of-the-art solutions.

9 Ethical Considerations

To the best of our knowledge, this project complies with the ethics guidelines. The research project focuses on secure multi-party computation, a cryptographic technique for providing strong privacy guarantees while maintaining application functionalities. Our goal is to design efficient protocols and enable MPC/2PC to be deployed in real-world applications, thus enhancing user privacy. The authors have read the ethics considerations discussions, detailed submission instructions, and guidelines for the ethics document. The authors believe the research was done ethically, and that the team's next-step plans (after publication) are ethical.

10 Open Science

We fully support the principles of the Open Science Policy. We open-sourced the research artifacts for this project on https://doi.org/10.5281/zenodo.15586635 under the Berkeley Software Distribution (BSD) license. The artifact includes the source code of FABLE and instructions to reproduce all evaluation results reported in this paper. We encourage the scientific community's unrestricted access to review, validate, and expand upon our work.

Acknowledgment

We thank the anonymous shepherd and reviewers for their valuable feedback. This work was supported in part by a National Science Foundation grant CNS232631 and a CyLab Presidential Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption standard. Cryptology ePrint Archive, Paper 2019/939, 2019.

[2] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for mpc and fhe. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*, pages 430–454. Springer, 2015.

[3] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. {Communication–Computation} trade-offs in {PIR}. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828, 2021.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.

[5] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*, pages 483–501. Springer, 2012.

[6] Shahla Atapoor, Nigel P Smart, and Younes Talibi Alaoui. Private liquidity matching using mpc. In *Cryptographers' Track at the RSA Conference*, pages 96–119. Springer, 2022.

[7] Subhadeep Banik, Khashayar Barooti, Serge Vaudenay, and Hailun Yan. New attacks on lowmc instances with a single plaintext/ciphertext pair. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part I 27*, pages 303–331. Springer, 2021.

[8] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[9] Beatrice Biasioli, Chiara Marcolla, Marco Calderini, and Johannes Mono. Improving and automating bfv parameters selection: An average-case approach. Cryptology ePrint Archive, Paper 2023/600, 2023.

[10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings 9*, pages 178–189. Springer, 2010.

[11] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Annual International Cryptology Conference*, pages 489–518, 2019.

[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[13] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion–a framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security*, 25(2):1–35, 2022.

[14] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Flute: fast and secure lookup table evaluations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 515–533. IEEE, 2023.

[15] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: oblivious pseudorandom functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 625–646. IEEE, 2022.

[16] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client oram. In *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS 2019. Internet Society, 2019.

[17] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 660–690. Springer, 2017.

[18] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[19] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255, 2017.

[20] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, SFCS-95. IEEE Comput. Soc. Press, 1995.

[21] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *Annual International Cryptology Conference*, pages 167–187. Springer, 2017.

[22] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 655–671, 2021.

[23] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS*, 2017.

[24] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

[25] Brett Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. Gigadoram: breaking the billion address barrier. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, pages 3871–3888, 2023.

[26] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.

[27] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 151–160, 1998.

[28] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 640–658. Springer, 2014.

[29] Eric Goldman. An introduction to the california consumer privacy act (ccpa). *Santa Clara Univ. Legal Studies Research Paper*, 2020.

[30] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*, STOC '87. ACM Press, 1987.

[31] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 379–388, 2011.

[32] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[33] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[34] Meng Hao, Weiran Liu, Liqiang Peng, Hongwei Li, Cong Zhang, Hanxiao Chen, and Tianwei Zhang. Unbalanced {Circuit-PSI} from oblivious {Key-Value} retrieval. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6435–6451, 2024.

[35] David Heath, Vladimir Kolesnikov, and Lucien KL Ng. Garbled circuit lookup tables with logarithmic number of ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 185–215. Springer, 2024.

[36] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Epigram: Practical garbled ram. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 3–33, Cham, 2022. Springer International Publishing.

[37] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private set intersection with linear communication from general assumptions. In *Proceedings of the 18th ACM Workshop on Privacy in the misc Society*, pages 14–25, 2019.

[38] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.

[39] Ilia Iliashenko. *Optimisations of fully homomorphic encryption*. PhD thesis, KU Leuven, 2019.

[40] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *Theory of Cryptography: 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 600–620. Springer, 2013.

[41] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 262–271, 2004.

[42] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. *Faster Secure Multi-party Computation of AES and DES Using Lookup Tables*, page 229–249. Springer International Publishing, 2017.

[43] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 34:4961–4973, 2021.

[44] Matt Kosinski. How to prevent prompt injection attacks. https://www.ibm.com/blog/prevent-prompt-injection, 2024.

[45] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, SFCS-97. IEEE Comput. Soc, 2017.

[46] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[47] Fukang Liu, Willi Meier, Santanu Sarkar, and Takanori Isobe. New low-memory algebraic attacks on lowmc in the picnic setting. *IACR Transactions on Symmetric Cryptology*, pages 102–122, 2022.

[48] Fukang Liu, Santanu Sarkar, Gaoli Wang, Willi Meier, and Takanori Isobe. Algebraic meet-in-the-middle attack on lowmc. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 225–255. Springer, 2022.

[49] J. Liu, J. Li, D. Wu, and K. Ren. Pirana: Faster multi-query pir via constant-weight codes. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 39–39, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[50] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947. IEEE, 2022.

[51] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 437–452. IEEE, 2023.

[52] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 245–254, 1999.

[53] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, USA, 1st edition, 2009.

[54] OpenAI. Vector embeddings - openai api. https://platform.openai.com/docs/guides/embeddings.

[55] OpenMP Architecture Review Board. OpenMP application program interface version 5.1. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, November 2020.

[56] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 130–130. IEEE Computer Society, 2024.

[57] Lance Roy Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[58] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. Senate: a {Maliciously-Secure}{MPC} platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2129–2146, 2021.

[59] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. Sirnn: A math library for secure rnn inference. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020. IEEE, 2021.

[60] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[61] Omri Serlin. Tpc-h benchmark. https://www.tpc.org/tpch.

[62] Amazon Web Services. Prompt injection security. https://docs.aws.amazon.com/bedrock/latest/userguide/prompt-injection.html.

[63] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[64] Haris Smajlović, Ariya Shajii, Bonnie Berger, Hyunghoon Cho, and Ibrahim Numanagić. Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. *Genome Biology*, 24(1):5, 2023.

[65] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A {Bandwidth-Efficient} distributed {ORAM} for 2-and 3-party computation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3907–3924, 2023.

[66] Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

[67] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.

Appendices

A Formal Description of Building Blocks

In this section, we introduce the formal definition of the building blocks in FABLE.

A.1 PBC and batch PIR

We use the following PBC procedures, which take hash functions as an extra argument:

1. $(C_1,\ldots,C_B) \leftarrow \mathsf{Encode}(\mathsf{db};(\mathsf{id}_1,\ldots,\mathsf{id}_N);(h_1,\ldots,h_w))$: Given an N-element array db and N IDs associated with the array elements, distributes the items to the buckets such that the i-th item is replicated into buckets $(h_1(\mathsf{id}_i),\ldots,h_w(\mathsf{id}_i)))$.

2. $\{\pi,\perp\} \leftarrow \mathsf{GenAssignPerm}((\mathsf{id}_1,\ldots,\mathsf{id}_b);(h_1,\ldots,h_w))$: Given a set of b indices $\mathsf{id}_1,\ldots,\mathsf{id}_b$, output an assignment permutation $\pi:[B]\to[B]$, such that

$$\forall i\in[b],\exists!k\in[w],h_k(x_i)=\pi(i).$$

   $\pi(b+1),\ldots,\pi(B)$ are placeholders to ensure π to be a permutation. The output is $\perp$ if a valid query-to-bucket assignment satisfying the requirements doesn't exist, which occurs with negligible probability.

We formalize PBC-based batch PIR as three procedures:

1. $\vec{q} \leftarrow \mathsf{PIR.Query}((i_1,\ldots,i_p);\pi)$: PIR's query construction excluding $\mathsf{GenAssignPerm}$. This procedure constructs a query for the input batch $(i_1,\ldots,i_p)$ based on a query-to-bucket assignment permutation π.

2. $\vec{a} \leftarrow \mathsf{PIR.Answer}(\vec{q};\mathsf{db})$: Generate an answer to a specific query using encoded database db.

3. $\vec{a} \leftarrow \mathsf{PIR.Extract}(\vec{a})$: Extract plain response $\vec{a}=(a_1,\ldots,a_p)$ from the response $\vec{a}$ such that

$$a_k=\mathsf{db}[i_k],\forall k\in[p].$$

The cuckoo table building procedure is formally defined as $(x_1^B,\ldots,x_q^B) \leftarrow \mathsf{CuckooBuild}((x_1,\ldots,x_p);(h_1,\ldots,h_w))$: Build a cuckoo hash table by cuckoo inserting $x_1,\ldots,x_p$ into an array of size $q>p$ with hash functions $(h_1,\ldots,h_w)$, such that

$$\forall i\in[p],\exists!k\in[w]\text{ s.t. }x^B_{h_k(x_i)}=x_i.$$

A.2 OPRF

We use a slightly modified version of OPRF that assumes the input and output are secret-shared. Formally, denote the key space as $\mathcal{K}$, for a PRF family $f_K(x)$, the protocol realizes the following functionality in the 2PC semi-honest model:

- Client and Server holds secret-shared input $[\![x]\!]$ in 2PC; Server has key $k \in \mathcal{K}$;
- Client and Server outputs secret-shared outputs $[\![o]\!] = [\![f_K(x)]\!]$.

B Security Proofs

B.1 Preliminaries

We term FABLE as π_{FABLE} and define the ideal functionality as $\mathcal{F}_{LUT}$ in Alg. 4. Let π_{PIR} be a batch PIR protocol based on homomorphic encryption such that the response ciphertexts contain only the requested value, i.e. there exists a procedure pack taking $(a_1,\ldots,a_b)$ and returning $(p_1,\ldots,p_{n_{ans}})$ such that the response is encryptions of the output. The query and response consist of n_{qu} and n_{ans} ciphertexts respectively. The view View_p of party p includes their inputs, randomness, and all messages. Let the input space be $\mathsf{Dom} = (\{0,1\}^\delta)^b$.

Our parameter choices rely on the following bounds.

1) $\mathcal{U}_B(N,B,w,\lambda)$: an upper bound of the server-side bucket size B after the server Encode the database with at most $2^{-\lambda}$ failure probability. According to [19, Equation (1)],

$$\begin{aligned}&\mathcal{U}_B(N,B,w,\lambda)\\ =&\operatorname*{argmin}_{B_0}\Pr[\text{At least one bucket has load} > B_0] \le 2^{-\lambda}\\ =&\operatorname*{argmin}_{B_0} B\sum_{i=B_0+1}^{wN}\binom{wN}{i}\left(\frac{1}{B}\right)^i\left(1-\frac{1}{B}\right)^{wN-i} \le 2^{-\lambda}.\end{aligned}$$

   For $B = 8192$ and $\lambda = 40$, $U_B(N,B,w,\lambda)$ is 556 for $N = 2^{20}$, 6798 for $N = 2^{24}$, and 100890 for $N = 2^{28}$.

2) $\mathcal{U}_b(\lambda)$: an upper bound of the batch size b such that the OPRF of deduplicated inputs collides with at most $2^{-\lambda}$ probability. This is similar to the birthday paradox so the same analysis applies. Namely,

$$\mathcal{U}_b(\lambda) = \arg\max_b \prod_{i=1}^{b-1}(1-\frac{i}{2^{64}}) \le 2^{-\lambda},$$

   which is 5793 for $\lambda = 40$.

3) $\mathcal{L}_r(\lambda)$: a lower bound of ε to satisfy the probability of failure of $2^{-\lambda}$, where ε is defined as the ratio of the number of bins to the number of items to be inserted during cuckoo hashing. $\mathcal{L}_r(\lambda)$ is approximately 1.5 for $\lambda = 40$ and 1.6 for $\lambda = 53$ [19].

The ideal functionality is defined in Alg. 4. The input and output for P_p ($p \in \{0,1\}$) are x_p and $\mathsf{Output}_p^\pi(x_0,x_1,\kappa)$ respectively. The joint output is $\mathsf{Output}^\pi(x_0,x_1,\kappa) = (\mathsf{Output}_0^\pi(x_0,x_1,\kappa), \mathsf{Output}_1^\pi(x_0,x_1,\kappa))$. A functionality $f = (f_0,f_1)$ takes x_0,x_1 as input and return $f_p(x_0,x_1)$ to P_p ($p \in \{0,1\}$). The semi-honest security is formally defined in Definition 1.

Algorithm 4: IDEAL FUNCTIONALITY

$\mathcal{F}_{LUT}(T, [\![\mathbf{x}]\!]_0 = ([\![x_1]\!]_0,\ldots,[\![x_b]\!]_0), [\![\mathbf{x}]\!]_1 = ([\![x_1]\!]_1,\ldots,[\![x_b]\!]_1))$:
1: **return** $(T(x_1),\ldots,T(x_b))$

$\mathcal{F}_{LUTp}(T, [\![\mathbf{x}]\!]_0 = ([\![x_1]\!]_0,\ldots,[\![x_b]\!]_0), [\![\mathbf{x}]\!]_1 = ([\![x_1]\!]_1,\ldots,[\![x_b]\!]_1)$:
1: $(v_1,\ldots,v_b) \leftarrow \mathcal{F}_{LUT}(T, [\![\mathbf{x}]\!]_0, [\![\mathbf{x}]\!]_1)$
2: **return** $([\![v_1]\!]_p,\ldots,[\![v_b]\!]_p)$ *// Generate secret shares.*

Definition 1 (Secure two-party Computation)**.** *Let $f = (f_0,f_1)$ be a functionality. We say that π securely computes f in the presence of static semi-honest adversaries if for all $p \in \{0,1\}$, there exist probabilistic polynomial-time algorithms $\mathcal{S}_p$ such that*

$$\begin{aligned}&\{(\mathcal{S}_p(1^n,x_0,f_p(x_0,x_1)), f(x_0,x_1)\}_{x_0,x_1,\kappa}\\ \overset{c}{\equiv}&\{(\mathsf{View}_p^\pi(x_0,x_1,\kappa), \mathsf{Output}^\pi(x_0,x_1,n))\}_{x_0,x_1,\kappa}\end{aligned}$$

The secret-sharing scheme we use in this paper is formally a $(2,2)$-threshold scheme [63] that divides a secret value x into $[\![x]\!]_0$ and $[\![x]\!]_1$, such that

- knowledge of both $[\![x]\!]_0$ and $[\![x]\!]_1$ makes x easily computable; and
- knowledge of either $[\![x]\!]_0$ or $[\![x]\!]_1$ leaves x completely undetermined (in the sense that all its possible values are equally likely).

B.2 Security Theorem

Theorem 2 is a formal version of Theorem 1.

Theorem 2 (Security)**.** *Consider the noise estimator adopted has at most $2^{-\lambda}$ failure probability, and the batch PIR is based on homomorphic encryption. Given $b \le \mathcal{U}_b(\lambda)$, $B \ge \mathcal{L}_r(\lambda)b$, and $N_B \ge \mathcal{L}_r(\lambda+\log_2 B)\ \mathcal{U}_B(N,B,w,\lambda)$, then for any LUT $T : \{0,1\}^\delta \to \{0,1\}^\sigma$, π_{FABLE} securely computes $(\mathcal{F}_{LUT0}^T, \mathcal{F}_{LUT1}^T)$ in the presence of static semi-honest adversaries.*

Proof. **Correctness.** By definition, the secure multi-party computation protocols are correct. The protocol of FABLE gives correct outputs if none of the following happens:

1) The OPRF output of deduplicated inputs collides, which introduces unexpected duplicates. The probability of this event is $\Pr_1 = 2^{-\lambda}$ as $b \le \mathcal{U}_b(\lambda)$.
2) Cuckoo hashing in GenAssignPerm fails with probability $\Pr_2 = 2^{-\lambda}$ from our definition of $\mathcal{L}_r$.
3) Cuckoo hashing in CuckooBuild (Line 9 in Alg. 2) fails. The probability of any bucket size exceeding the upper bound $\mathcal{U}_B(N,B,w,\lambda)$ is bounded by $\Pr_3 = 2^{-\lambda}$. The probability of cuckoo hashing failure given the upper bound is not exceeded is $\Pr_4 = B2^{-\lambda-\log_2 B} = 2^{-\lambda}$.

— Algorithm 5: SIMULATOR FOR SUB-PROCEDURE —

$\mathcal{S}_0^s(1^\kappa, \mathsf{Input}_0^s, \mathsf{Output}_0^s)$:

1: $(\llbracket x_1^D \rrbracket_0, \ldots, \llbracket x_b^D \rrbracket_0) \leftarrow \mathsf{Input}_0^s$.
2: $(id_1, \ldots, id_b) \xleftarrow{\$} O$.
3: $\mathsf{Output}_{sim}^{oprf} \leftarrow (\llbracket id_1 \rrbracket_0, \ldots, \llbracket id_b \rrbracket_0)$.
4: $\mathsf{View}_0^{oprf} \leftarrow \mathcal{S}_0^{oprf}(1^\kappa, \mathsf{Input}_0^s, \mathsf{Output}_{sim}^{oprf})$.
5: $\{a_{i,k} : i \in [b], k \in [w]\} \leftarrow \mathsf{Output}_0^s$.
6: $\mathsf{qu} \leftarrow \mathsf{Query}(id_1, \ldots, id_b)$.
7: **return** $(\mathsf{Input}_0^s, r^0, (\mathsf{View}_0^{oprf}, (id_1, \ldots, id_b), \mathsf{qu}))$.

$\mathcal{S}_1^s(1^\kappa, \mathsf{Input}_1^s, \mathsf{Output}_1^s)$:

1: $(\llbracket x_1^D \rrbracket_1, \ldots, \llbracket x_b^D \rrbracket_1) \leftarrow \mathsf{Input}_1^s$.
2: $(id_1, \ldots, id_b) \xleftarrow{\$} O$.
3: $\mathsf{Output}_{sim}^{oprf} \leftarrow (\llbracket id_1 \rrbracket_1, \ldots, \llbracket id_b \rrbracket_1)$.
4: $\mathsf{View}_1^{oprf} \leftarrow \mathcal{S}_1^{oprf}(1^\kappa, \mathsf{Input}_1^s, \mathsf{Output}_{sim}^{oprf})$.
5: $\mathsf{qu} \leftarrow \overbrace{\mathsf{HEEnc}(p_0), \ldots, \mathsf{HEEnc}(p_0)}^{n_{qu}}$.
6: $e_1, \ldots, e_{n_{ans}} \xleftarrow{\$} \chi_f$.
7: $\mathsf{ans} \leftarrow (\mathsf{HEEnc}(p_0) + e_1, \ldots, \mathsf{HEEnc}(p_0) + e_{n_{ans}})$.
8: **return** $(\mathsf{Input}_1^s, r^1, (\mathsf{View}_1^{oprf}, \mathsf{qu}, \mathsf{ans})))$.

4) Decryption failure in Extract, which happens only when our noise estimation is correct. So the failure probability is $\Pr_5 = 2^{-\lambda}$.

Thus, $\Pr[\mathsf{Output}^{\pi_{FABLE}}(\mathbf{x}; T) = \mathcal{F}_{LUT}(\mathbf{x}; T)] = \prod_{i=1}^{5}(1 - \Pr_i) = (1 - 2^{-\lambda})^5 \geq 1 - 5 \times 2^{-\lambda}$.

Security. FABLE has multiple stages in sequential, most of which are realized with general secure multi-party computation. Therefore, the security of FABLE follows from the sequential composition theorem for semi-honest adversaries [53, Theorem 7.3.3] if the sub-procedure including ServerSetup, OPRFEvaluation, and ShareRetrieval (excluding share conversion) can be securely computed in the presence of static semi-honest adversaries. We refer to the input and output to this sub-protocol as Input^s and Output^s, with party index as a subscript. Specifically,

$$\mathsf{Input}_0^s = (\llbracket x_1^D \rrbracket_0, \ldots, \llbracket x_b^D \rrbracket_0); \mathsf{Input}_1^s = (\llbracket x_1^D \rrbracket_1, \ldots, \llbracket x_b^D \rrbracket_1)$$
$$\mathsf{Output}_0^s = \perp; \qquad \mathsf{Output}_1^s = \{a_{i,k} : i \in [b], k \in [w]\}$$

We use $\mathcal{S}_0^{oprf}$ and $\mathcal{S}_1^{oprf}$ to denote the simulator for the 2PC OPRFEvaluation. Let p_0 be the zero polynomial and χ_f be the flooding noise distribution. The simulators are outlined in Alg. 5, where HEEnc is the HE encryption operation. As the functionality is deterministic, we only need to show that the simulator's output is computationally indistinguishable from the real-world view. The input and randomness in the view are just repeated by the simulator in the output, so only the indistinguishability of messages needs to be proved.

Proof of indistinguishability with corrupted server. We show that the real-world distribution is computationally indistinguishable from the simulated distribution via the hybrid argument. In the final simulated distribution, the simulator $\mathcal{S}_0^s$ does not use the client's share of the query index, so a corrupted server learns nothing in the real world. We ignore the answer message in the server's view as it is generated from the queries without any extra information from the client.

- Hyb_0. This corresponds to the real-world distribution, where the server uses its actual input Input_0^s.
- Hyb_1. We replace the view during 2PC OPRF evaluation with the simulator's output with real output shares, i.e. $\mathsf{View}_0^{oprf} \leftarrow \mathcal{S}_0^{oprf}(1^\kappa, \mathsf{Input}_0^s, \mathsf{Output}_0^{oprf})$. $\mathsf{Hyb}_0 \overset{c}{\equiv} \mathsf{Hyb}_1$ following the security of the 2PC protocol.
- Hyb_2. We change the OPRF IDs to randomly sampled numbers. It is computationally indistinguishable from Hyb_1 because of the properties of OPRF.
- Hyb_3. We change the shares of OPRF IDs accordingly. The resulting view from OPRFEvaluation is equal to $\mathcal{S}_0^{oprf}(1^\kappa, \mathsf{Input}_0^s, \mathsf{Output}_{sim}^{oprf})$. $\mathsf{Hyb}_2 \overset{c}{\equiv} \mathsf{Hyb}_3$ because of the security definition of secret shares.
- Hyb_4. We construct the queries from the randomly sampled ID instead of using the original queries. It is computationally indistinguishable from Hyb_3 due to the IND-CPA security of HE [1]. Finally, we note that Hyb_4 is identically distributed to the simulator's output $\mathcal{S}_0^s(1^\kappa, \mathsf{Input}_0^s, \mathsf{Output}_0^s)$, completing the proof.

Proof of indistinguishability with corrupted client. We show that the real-world distribution is computationally indistinguishable from the simulated distribution via the hybrid argument. In the final simulated distribution, the simulator $\mathcal{S}_1^s$ does not use the confidential LUT held by the server, so a corrupted client learns nothing except the output in the real world. Note that the server's messages contain only 2PC views and ciphertexts from homomorphic encryptions.

- Hyb_0. This corresponds to the real-world distribution, where the client uses its actual input Input_1^s.
- Hyb_1. We replace the view during 2PC OPRF evaluation with the simulator's output with real output shares, i.e. $\mathsf{View}_1^{oprf} \leftarrow \mathcal{S}_1^{oprf}(1^\kappa, \mathsf{Input}_1^s, \mathsf{Output}_1^{oprf})$. $\mathsf{Hyb}_0 \overset{c}{\equiv} \mathsf{Hyb}_1$ following the security of the 2PC protocol.
- Hyb_2. We change the output shares Output_1^{oprf} to shares of randomly sampled IDs $\mathsf{Output}_{sim}^{oprf}$. $\mathsf{Hyb}_1 \overset{c}{\equiv} \mathsf{Hyb}_2$ because of the security definition of secret shares.
- Hyb_3. We replace all query ciphertexts with encryption of zero, i.e. $\mathsf{HEEnc}(p_0)$. To show that it is computationally indistinguishable from Hyb_2, we define a series of sub hybrids $\mathsf{Hyb}_{3,i}$ for $i \in [n_{qu}]$, where we replace

the first i query ciphertexts with $\mathsf{HEEnc}(p_0)$. $\mathsf{Hyb}_{3,i} \overset{c}{\equiv} \mathsf{Hyb}_{3,i+1} (i \in [n_{qu}-1])$ because of the IND-CPA security of HE [1], so $\mathsf{Hyb}_2 = \mathsf{Hyb}_{3,0} \overset{c}{\equiv} \mathsf{Hyb}_{3,n_{qu}} = \mathsf{Hyb}_3$.

- Hyb_4. We replace the response ciphertexts with fresh encryptions of the real response values plus freshly sampled errors from $e_i \overset{\$}{\leftarrow} \chi_f$. It is statistically indistinguishable from Hyb_3 because of the smudging Lemma [5]. Namely, the noise from the computation is completely masked by the added noise, and thus removing it will not affect the noise distribution.
- Hyb_5. We further zero out the data stored in the response ciphertexts, i.e. changing the i-th response ciphertexts to $\mathsf{HEEnc}(p_0)+e_i$. The computational indistinguishability between Hyb_5 and Hyb_4 can be shown by a similar way we prove $\mathsf{Hyb}_2 \overset{c}{\equiv} \mathsf{Hyb}_3$. Finally, we note that Hyb_5 is identically distributed to the simulator's output $\mathcal{S}_1^s(1^\kappa, \mathsf{Input}_1^s, \mathsf{Output}_1^s)$, completing the proof.

□

C Details of complexity analysis

C.1 Baselines

The complexities of FLORAM and 2P-DUORAM are from [65, Table 2]. For SP-LUT+ and FLUTE, we use the communication complexities from [14, Table 2]. SP-LUT+'s computation contains a 1-out-of-N random OT with σ-bit messages (of complexity $O(\sigma \log N)$ [52, Protocol 2.1]) and a masking operation applied to each table entry (of complexity $O(\sigma N)$), hence the total complexity is $O(\sigma N)$. The bottleneck of FLUTE's computation is the second step of the online phase, which computes N inner products between two vectors of length N, resulting in a computation complexity of $O(N^2)$.

C.2 FABLE

We provide the detailed complexity of each component in FABLE, where $\delta \leq 64$, w is a small constant satisfying $w \ll \log^2 B$ and $N_B \leq \frac{2wN}{B}$ (note that $\frac{wN}{B}$ is the average bucket size for cuckoo-hashing-based PBC).

2PC algorithms. The complexity of 2PC primitives (i.e. comparison, CondSwap, and MUX) hold the same for communication and computation, linear to the inputs' bitlength. Thus the complexities of Sort and two permutations over a p-length array with λ-bit items (abbr. (p,λ)-Sort) are $O(p \log^2 p \lambda)$. Deduplication contains a (b,δ) Sort and a $O(b\delta)$ linear scan, so the complexity is bounded by the former, i.e. $O(b \log^2 b\delta)$. Similarly, Expansion's complexity is $O(b \log^2 b\sigma)$. The complexity of OPRFEvaluation is $O(b)$ to evaluate the whole batch as the per-block cost is constant. The complexity of share conversion is $O(B(\delta+\sigma))$. Decode has three sorting-derived primitives with total complexity $O(B \log^2 B(\delta+\sigma))$ and a $O(B(\delta+\sigma))$ linear scan, so the complexity is equal to the first component.

Table 5: Asymptotic complexity analysis of each component of FABLE.

Complexity	Computation	Communication
Deduplication	$O(b \log^2 b\delta)$	$O(b \log^2 b\delta)$
OPRFEvaluation	$O(b)$	$O(b)$
ServerSetup	$O(N)$	-
Query	$O(\sqrt{N})$	$O(\sqrt{N})$
Answer	$O(N)$	$O(1)$
Extract	$O(B)$	-
Share conversion	$O(B(\delta+\sigma))$	$O(B(\delta+\sigma))$
Decode	$O(B \log^2 B\sigma)$	$O(B \log^2 B\sigma)$
Expansion	$O(b \log^2 b\sigma)$	$O(b \log^2 b\sigma)$
Client-side	$O(\sqrt{N})$	$O(\sqrt{N})$
Server-side	$O(N)$	$O(\sqrt{N})$

Share Retrieval. For ShareRetrieval, we need to consider computational and communication complexity separately. As for computation, ServerSetup has complexity $O(N)$. It encodes the whole database and creates w copies. The size of each PIR query in PIRANA [49] is equal to the code length $m = \mathrm{argmin}_m \binom{m}{2} \geq N$, implying $\frac{1}{2}(m-2)^2 < \binom{m-1}{2} < N$ and thus $m < 2+\sqrt{2N} \in O(\sqrt{N})$, so the complexity of Query is $O(\sqrt{N})$. Answer is $O(N)$. Extract is $O(B)$ by its description.

In terms of communication complexities, the query size is $O(\sqrt{N})$ while the response size is $O(1)$, directly following PIRANA.

Putting all together. We report all components' complexity in Table 5, where we consider N as the highest order term. All steps whose computation scale linearly with N (i.e. ServerSetup and Answer) are executed by the server, demonstrating that FABLE achieves client lightweightness.

D Formal Descriptions of Some Algorithms

D.1 The SortOrder algorithm

SortOrder is formally defined in Alg. 8. We follow the imperative implementation to save some space. The primitives derived from this abstraction are formally summarized in Alg. 9.

D.2 The ShareRetrieval and Expansion algorithms

The formal description of ShareRetrieval and Expansion is in Alg. 6 and Alg. 7 respectively. Note that for a bit string s (with starting index 1), $s_{i:j}$ represents a slice from indices i to j (inclusive).

Algorithm 6: ShareRetrieval

$\mathsf{Query}(\mathsf{id}_1,\ldots,\mathsf{id}_b)$:
1: $\pi \leftarrow \mathsf{GenAssignPerm}((\mathsf{id}_1,\ldots,\mathsf{id}_b);(h_1^B,\ldots,h_w^B))$
2: **for** $k \in [w]$ **do**
3: $\quad \vec{q}_k \leftarrow \mathsf{PIR.Query}((h_k^P(\mathsf{id}_1),\ldots,h_k^P(\mathsf{id}_b));\pi)$
4: **return** $(\vec{q}_1,\ldots,\vec{q}_w)$

$\mathsf{Answer}((\vec{q}_1,\ldots,\vec{q}_w);(\mathsf{db}_M^1,\ldots,\mathsf{db}_M^w))$:
1: **for** $k \in [w]$ **do**
2: $\quad \vec{a}_k \leftarrow \mathsf{PIR.Answer}(\vec{q}_k;\mathsf{db}_M^k)$
3: **return** $(\vec{a}_1,\ldots,\vec{a}_w)$

$\mathsf{Extract}(\vec{a}_1,\ldots,\vec{a}_w)$:
1: **for** $k \in [w]$ **do**
2: $\quad (a_{1,k},\ldots,a_{B,k}) \leftarrow \mathsf{PIR.Extract}(\vec{a}_k)$
3: $\quad$ **for** $j \in [B]$ **do**
4: $\quad\quad a^I_{j,k} \leftarrow a_{j,k1:\delta}$
5: $\quad\quad a^O_{j,k} \leftarrow a_{j,k\delta+1:\delta+\sigma}$
6: **return** $\{a^I_{j,k}, a^O_{j,k} : k \in [w], j \in [B]\}$

Algorithm 7: EXPANSION

$\mathsf{Expansion}((\llbracket v_1^D \rrbracket,\ldots,\llbracket v_b^D \rrbracket);(\llbracket t_2 \rrbracket,\ldots,\llbracket t_b \rrbracket);C_S)$:
1: *// Invert Placeholder Replacement*
2: **for** $i \in \{2,\ldots,b\}$ **do**
3: $\quad \llbracket v_i \rrbracket \leftarrow \mathsf{MUX}(\llbracket t_i \rrbracket, \llbracket v_{i-1}^D \rrbracket, \llbracket v_i^D \rrbracket)$
4: *// Invert Sort*
5: $\mathsf{InvPermute}((\llbracket v_1 \rrbracket,\ldots,\llbracket v_b \rrbracket);C_S)$
6: **return** $(\llbracket v_1 \rrbracket,\ldots,\llbracket v_b \rrbracket)$

Algorithm 8: THE SORTORDER PRIMITIVE

$\mathsf{SortOrder}(p)$:
1: $ord \leftarrow [\,]$.
2: **for** $k = 2^0, 2^1, \ldots, p$ **do**
3: $\quad$ **for** $j = k/2, k/4, \ldots, 1$ **do**
4: $\quad\quad$ **for** $i = 1,\ldots,p$ **do**
5: $\quad\quad\quad l = i \wedge j$;
6: $\quad\quad\quad$ **if** $l > i$ **then**
7: $\quad\quad\quad\quad$ **if** $(i\&k) == 0$ **then**
8: $\quad\quad\quad\quad\quad ord.append((i,l))$.
9: $\quad\quad\quad\quad$ **else**
10: $\quad\quad\quad\quad\quad ord.append((l,i))$.
11: **return** ord.

Algorithm 9: SORT AND DERIVED PRIMITIVES

$\mathsf{Sort}((\llbracket x_1 \rrbracket,\ldots,\llbracket x_p \rrbracket))$:
1: $ord \leftarrow \mathsf{SortOrder}(p)$.
2: **for** $k = 1,\ldots,n_s(p)$ **do**
3: $\quad (i,j) \leftarrow ord[k]$.
4: $\quad \llbracket c_k \rrbracket \leftarrow \llbracket x_i \rrbracket > \llbracket x_j \rrbracket$.
5: $\quad \mathsf{CondSwap}(\llbracket c_k \rrbracket, \llbracket x_i \rrbracket, \llbracket x_j \rrbracket)$.
6: **return** $C = (\llbracket c_1 \rrbracket,\ldots,\llbracket c_{n_s(p)} \rrbracket)$.

$\mathsf{Permute}((\llbracket x_1 \rrbracket,\ldots,\llbracket x_p \rrbracket);C = (\llbracket c_1 \rrbracket,\ldots,\llbracket c_{n_s(p)} \rrbracket))$:
1: $ord \leftarrow \mathsf{SortOrder}(p)$.
2: **for** $k = 1,\ldots,n_s(p)$ **do**
3: $\quad (i,j) \leftarrow ord[k]$.
4: $\quad \mathsf{CondSwap}(\llbracket c_k \rrbracket, x_i, x_j)$.

$\mathsf{InvPermute}((\llbracket x_1 \rrbracket,\ldots,\llbracket x_p \rrbracket);C = (\llbracket c_1 \rrbracket,\ldots,\llbracket c_{n_s(p)} \rrbracket))$:
1: $ord \leftarrow \mathsf{SortOrder}(p)$.
2: **for** $k = n_s(p),\ldots,1$ **do**
3: $\quad (i,j) \leftarrow ord[k]$.
4: $\quad \mathsf{CondSwap}(\llbracket c_k \rrbracket, x_i, x_j)$.

$\mathsf{GenContext}((x_1,\ldots,x_p))$:
1: $ord \leftarrow \mathsf{SortOrder}(p)$.
2: **for** $k = 1,\ldots,n_s(p)$ **do**
3: $\quad (i,j) \leftarrow ord[k]$.
4: $\quad c_k \leftarrow x_i > x_j$, then swap x_i and x_j if c_k.
5: **return** $C = (\llbracket c_1 \rrbracket,\ldots,\llbracket c_{n_s(p)} \rrbracket)$.

E Improvement estimation in Deduplication

To demonstrate the improvement of our optimizations of deduplication, we count non-linear gates on data (but not indices), including comparison, multiplexer, and conditional swap. These gates have similar communication costs and are *all* gates that incur communication in Deduplication and Expansion protocol of FABLE. Certain non-linear gates on secret indices in Snoopy are ignored. Assuming a batch size of 4096, Snoopy's deduplication contains 352,256 such gates and expansion has 950,272 such gates. On the contrary, our Deduplication involves 303,104 such gates, 86.0% of Snoopy, while for Expansion, it is only 151,552, 15.9% of Snoopy. The substantial improvement in Expansion is attributed to our effective caching techniques.

Table 6: Crypten's performance breakdown.

Network condition	LAN	WAN
Multiplication triples generation	12,614s	106,962s
Word-count vector generation	250.55s	2782.8s
Embedding matrix multiplication	2.73s	28.84s
Total	12,868s	109,774s

F Extended Evaluation of FABLE

We include an extended version of Figure 5 in Figure 8, which contains four network configurations.

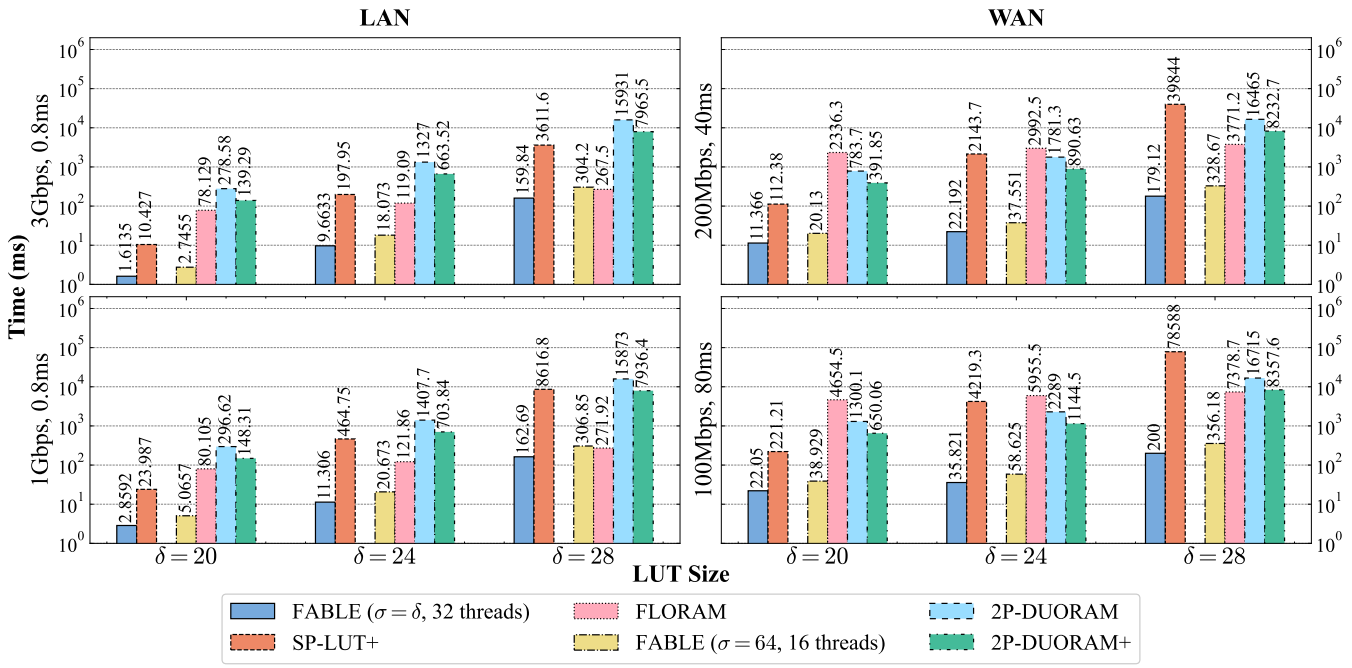

Figure 8: Amortized running time with respect to various table sizes and network settings.

G Details of FABLE's Application

G.1 A performance breakdown for the Crypten's baseline

Table 6 shows a breakdown of each component in Crypten's baseline of the secure embedding lookup application. The bottleneck is the multiplication triple generation, which is mandatory for matrix multiplication.

G.2 The example query used in the secure query application

We use the following query which we simplify from TPC-H query 8 is a multi-way join across many tables that span both parties, with selective filters on `region`, `nation`, and `orders`:

```
select
    extract(year from o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount) as volume,
    n.n_name as nation
from
    lineitem,
    orders,
    customer,
    nation n,
    region
where
    and l_orderkey = o_orderkey
    and o_custkey = c_custkey
    and c_nationkey = n.n_nationkey
    and n.n_regionkey = r_regionkey
    and r_name = ':2'
    and o_orderpriority = '1-URGENT'
    and o_orderdate = '1995-01-01'
```

The filters on `orders` are selective enough that the filtered table only has 2060 rows.