Hybrid Trust Multi-party Computation with Trusted Execution Environment

Pengfei Wu[†], Jianting Ning[‡,¶,✉], Jiamin Shen[†], Hongbing Wang[†], Ee-Chien Chang[†]
[†]School of Computing, National University of Singapore
[‡]College of Computer and Cyber Security, Fujian Normal University
[¶]Institute of Information Engineering, Chinese Academy of Sciences
Email: {wupf,shen_jiamin,changec}@comp.nus.edu.sg, {jtning88,florawang.2011}@gmail.com

***Abstract*—Trusted execution environment (TEE) such as Intel SGX relies on hardware protection and can perform secure multi-party computation (MPC) much more efficiently than pure software solutions. However, multiple side-channel attacks have been discovered in current implementations, leading to various levels of trust among different parties. For instance, a party might assume that an adversary is unable to compromise TEE, while another might only have a partial trust in TEE or even does not trust it at all. In an MPC scenario consisting of parties with different levels of trust, one could fall back to pure software solutions. While satisfying the security concerns of all parties, those who accept TEE would not be able to enjoy the benefit brought by it.**

In this paper, we study the above-mentioned scenario by proposing HYBRTC, a generic framework for evaluating a function in the *hybrid trust* manner. We give a security formalization in universal composability (UC) and introduce a new cryptographic model for the TEEs-like hardware, named *multifaceted trusted hardware* $\mathcal{F}_{TH}$, that captures various levels of trust perceived by different parties. To demonstrate the relevance of the hybrid setting, we give a distributed database scenario where a user completely or partially trusts different TEEs in protecting her distributed query, whereas multiple servers refuse to use TEE in protecting their sensitive databases. We propose a maliciously-secure protocol for a typical select-and-join query in the multi-party setting. Experimental result has shown that on two servers with 2^{20} records in datasets, and with a quarter of records being selected, only 165.82s is incurred which achieves more than $18,752.58\times$ speedups compared to cryptographic solutions.

I. INTRODUCTION

Secure multi-party computation (MPC) allows a set of parties to jointly compute a function on their sensitive inputs, so that nothing will be revealed beyond the output of the function. Since the appearance of Yao's garbled circuit [1] and the protocol of Goldreich-Micali-Wigderson (GMW) [2] showing its feasibility, MPC has been extensively researched and well improved in both security and efficiency [3], [4], [5], [6]. Nowadays, it has been suggested in a wide range of applications including private information retrieval (PIR) [7], contact tracing discovery [8], privacy-preserving machine learning [9], and secure DNA matching [10].

Despite extensive efforts in improving efficiency, MPC is still not sufficiently practical due to a large number of cryptographic operations incurred. For example, in the two-party computation using Yao's garbled circuit [1], it is manifested by representing the function as a boolean circuit and then garbling all input and output wires for each bit. Then, both parties engage in the oblivious transfer (OT) and allow one party to evaluate each gate in the circuit with two input wires one by one. This process, however, may incur heavy computational and communication overhead because the circuit complexity is linear to the input size [11]. It has become one of the major obstacles in deploying MPC to some real-world applications, especially when a large dataset is taken as the input, e.g., PIR and binary search.

Trusted execution environment (TEE) such as commoditized hardware, Intel Software Guard Extensions (Intel SGX) [12], offers a pragmatic solution to reduce these costs. It has become an attractive alternative to highly expensive cryptographic MPC protocols. TEE provides a hardware-level isolated region, commonly referred to as *enclave*, wherein code and data cannot be accessed directly by privileged software, including operating system (OS), Hypervisor, and system BIOS. This technique offers a user the capability to thwart a powerful adversary without relying on complicated cryptographic constructions. However, the security of TEE relies on additional assumptions that the hardware is tamper-proof and the hardware vendor is trusted, which some users do not accept. Additionally, some side-channel vulnerabilities in an application, e.g., control flow and branch prediction, can be exploited by an adversary to compromise the security of the enclave. A series of such attacks have been discovered in recent years by the research community, e.g., [13], [14], [15], [16], [17], leading to various levels of trust in TEE.

- Firstly, consider that the data used is highly sensitive, and any leakage would lead to a critical interest risk. Some users *do not trust* TEE at all.
- Secondly, the application codebase could be large and contributed by various developers. It could include some libraries that are well-tested and developed by trusted developers, e.g., Crypto API Toolkit for Intel SGX [18] and Intel SGX SSL library [19]. Compared to the large code developed by less well-known third parties, a user might have higher confidence in these libraries. Therefore,

[✉]Corresponding author

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24173
www.ndss-symposium.org

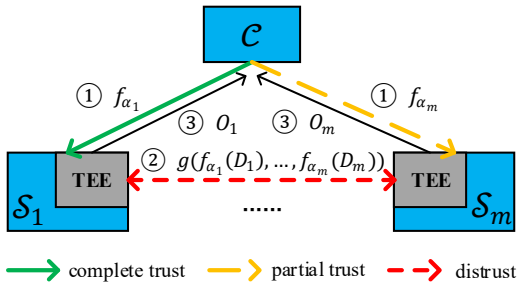


Fig. 1: An overview of hybrid trust computing setting.

some users would like to *partially trust* TEE that they accept some routines in the libraries can be kept confidential, while other parts of the application potentially might be leaked by the side-channel attack. In this paper, we consider a partial trust where a user trusts the implementation of secure channel and cryptographic primitives in libraries. She believes that the session key used in her communication with the enclave, as well as secret keys would not be leaked via side-channel in TEE execution.

- Lastly, some users believe the application provided by the vendor is robust enough to prevent side-channel attacks. So they can choose to *completely trust* TEE for the sake of performance benefits in private computation.

Hybrid Trust Computing. Motivated by a mixture of trusts perceived by different parties, we consider a multi-party computation of a function f composed with a function g that performs on a client $\mathcal{C}$ and multiple TEE-enabled servers $\mathcal{S}_1, \ldots, \mathcal{S}_m$, see Fig. 1. Each server $\mathcal{S}_i, i \in \{1, \ldots, m\}$, first performs f on a local database D_i after receiving a parameter α_i from $\mathcal{C}$. Using the results of $f_{\alpha_i}(D_i)$, they jointly work on g and return the partial outputs O_i to $\mathcal{C}$. Consider that databases contain the utmost sensitive data, so all servers distrust TEE from each other. Although the parameter α_i used is also sensitive, it induces lesser consequences if compromised. Therefore, $\mathcal{C}$ would like to have higher trusts in TEEs of all servers. For those applications that are less likely to be compromised by side-channel attacks in function evaluation, $\mathcal{C}$ completely trusts their TEEs. For others, she chooses to have partial trust in their TEEs.

We would like to remark that this computing model can be instantiated to support a wide range of privacy-preserving applications. For instance, the government makes a distributed query to multiple distinct financial institutions for gathering the fiscal spending in the last year. All institutions first filter out ineligible records and then securely make an aggregation on the remaining. However, the banking statistics are sensitive for individuals, and all institutions will take a great credit risk if some data is leaked during the query execution. Although the trusted hardware is deployed, e.g., Intel SGX, they still prefer some standard MPC protocol with a provable security guarantee. While from the perspective of the government, the query parameter ("`year=2020`") used is not so sensitive as the databases from institutions. The government can completely or partially trust an institution's TEE depending on the side-channel vulnerability analysis of its program executing a database query. In this scenario, to satisfy the security concerns of all parties, one may choose to evaluate both filter and aggregation using MPC protocols. However, this solution may introduce expensive overhead without showing the benefits of TEE. Potentially, a combination of TEE and MPC could reduce this overhead while satisfying the secrecy needs of each party.

The use of TEEs such as Intel SGX for secure computation has been investigated in a line of research works, e.g., [26], [27], [28], [29], [30]. We note that these previous works mainly focus on the theoretical analysis of TEE-based protocol. It is still unclear how the security guarantees and how much efficiency gain we can obtain for privacy-preserving applications in the hybrid trust setting.

A. Challenges and Technical Overview

In this paper, we propose HYBRTC, a generic framework for performing a hybrid trust multi-party computation wherein different parties have different trusts in TEEs. It can provably guarantee the confidentiality of input from all parties by taking advantage of standard MPC protocol and TEEs.

There are several subtleties arising when designing our framework. Firstly, how to partition a protocol in hybrid trust computing should be carefully studied to prevent potential privacy leakage. A naive way is to put TEE and MPC together and allow some parts that parties can trust to be performed inside the enclave, while others use MPC. However, some intermediate results can be leaked when switching between protocol partitions, allowing inferences about private inputs. Choi et al. [27] propose a hybrid approach for secure function evaluation (SFE) by performing even-round protocols using Yao's garbled circuit and executing odd-round ones inside the enclave. However, their solution is not secure when one round's input depends on the previous round. One party can infer more sensitive information about another party if she can collect intermediate results. This drawback is also independently investigated by Felsen et al. [28]. In HYBRTC, we have a user share a set of secret keys with a partially trusted TEE by establishing a secure channel between them. They can process their inputs with keys locally and then feed them into MPC. If secret keys are protected, the server will learn nothing about user's input, thereby keeping confidentiality in combination of TEE and MPC. Additionally, in order to break the tie of input and output from two phases f and g, we allow g to be performed on a *randomly chosen part* of f's output. In this way, the input of g can be independent of the output of f, and thus we can remedy the leakage in Choi et al. [27].

Secondly, different from the standard MPC, performing the function g in the hybrid trust setting requests for a higher security requirement. Besides the input content, i.e., $f_{\alpha_i}(D_i)$, their sizes are also required to be hidden since they mostly relate to the sensitive parameter α_i. Additionally, we only allow $\mathcal{C}$ rather than servers to learn the final output of g, which poses a stronger security demand than the standard MPC. To address this issue, we perform g multiple times, each on a different set of inputs separately. After the protocol terminates, either party can only learn the size of partial output from the ciphertext but has no idea about the plaintext and total size.

Last but not least, how to formalize the hybrid trust computing and provide rigorous security analysis remains a challenge. In this paper, we introduce a new *multifaceted trusted hardware* model $\mathcal{F}_{TH}$ as a formalization of TEEs, the

function performing in which is parameterized as a probabilistic polynomial time (PPT) Turing machine Prog. When $\mathcal{F}_{TH}$ is not completely trusted, the adversary is able to learn all incoming messages received by $\mathcal{F}_{TH}$. In HYBRTC, we follow the *universal composability* (UC) framework [31], a simulation-based proof technique, in the security analysis. Particularly, we parameterize the security definition with two parameters $\mathcal{L}, \mathcal{T}$. The former is a set of *leakage function* capturing partial information tolerated to be leaked, and the latter is a *trust function* interpreting the trust relationship between parties.

Contributions. In summary, we make the following main contributions in this paper.

- We propose HYBRTC, a generic framework for protecting the confidentiality of all parties' inputs and outputs in hybrid trust computing.
- We instantiate our framework in a scenario of performing a distributed query on multiple databases by proposing a maliciously-secure protocol for query execution. Our protocol can support several typical SQL operators, including `SELECT`, `JOIN`, and some aggregate functions `COUNT`, addition (+), and subtraction (−).
- We introduce a new cryptographic model for the TEEs-like hardware named multifaceted trusted hardware $\mathcal{F}_{TH}$. We also provide rigorous security analysis for hybrid trust computing in the UC model.
- We implement a prototype of HYBRTC in Intel SGX and evaluate the overhead incurred by enclave and MPC protocol. We compare our privacy-preserving distributed query protocol with state-of-the-art cryptographic solutions, including a maliciously-secure data analytics system Senate [35] and three alternatives in the private set intersection (PSI) [23], [24], [25]. The experimental result shows the promising efficiency of our scheme.

B. Use Cases

One of our motivations lies in designing a privacy-preserving protocol allowing a user to submit a distributed query to multiple databases maintained by different parties. In principle, HYBRTC can support arbitrary queries because it builds on generic computing tools, i.e., TEE and MPC protocol. However, our multi-party computing paradigm is more relevant to `SELECT`, `JOIN`, and some aggregate functions. All servers first filter out those records unsatisfying the given condition in `WHERE` (i.e., the parameter α_i) and then privately join the remaining on a common attribute. This paradigm serves as one of the most vital building blocks in database [32], [33] and has been shown in more than 70% SQL queries [34]. On the basis of this computing paradigm, here, we present two use cases on the distributed query, each from a different domain, which can be instantiated by our HYBRTC framework.

Query 1. Banking statistics. The first application has mentioned in Section I, where the government queries for the total fiscal spending last year of different departments in databases $D_1, \ldots, D_m$ maintained by m financial institutions, $m \geq 2$. In this setting, the government is willing to completely trust or partially trust an institution's TEE depending on whether its program used is vulnerable to the side-channel attack. However, for taking a great credit risk, institutions cannot share their databases with each other. The government may use the following query.

```
SELECT D1.spending + ... + Dm.spending
FROM D1 AS c1 ...
JOIN Dm AS cm ON c1.depa_ID = ...
   = cm.depa_ID
WHERE c1.year = 2020 AND ...
   AND cm.year = 2020;
```

Query 2. Medical analysis. COVID-19 is a contagious disease caused by the coronavirus, which has spread worldwide, leading to an ongoing pandemic. There are m hospitals that wish to study the symptom, sequela, and the number of people infected with COVID-19 but do not want to share patient's information, $m \geq 2$. In addition, the data consumer can completely trust a hospital's TEE if its program is robust enough to thwart side-channel attacks; otherwise, she can have a partial trust in it. Under this circumstance, the data consumer may use

```
SELECT symptom, sequela, COUNT(*)
FROM D1 AS c1 ...
JOIN Dm AS cm ON c1.user_ID = ...
   = cm.user_ID
WHERE c1.COVID-19 = true ...
   AND cm.COVID-19 = true;
```

C. Related Work

TEE-based Secure Computation. A variety of MPC protocols built on TEEs have been proposed for passive and malicious adversary [26], [27], [28], [29], [30]. Gupta et al. [29] propose a protocol for the SFE problem using Intel SGX and show how to improve it in the semi-honest model. Bahmani et al. [26] refine their work by putting only sensitive programs inside the enclave, and major workloads are left to the untrusted machine. They also provide rigorous security analysis for SGX-based solutions by introducing labeled attested computation (LAC). The primary difference between LAC and $\mathcal{F}_{TH}$ lies in their intuitive design coming from some authenticated mechanisms provided by TEEs, e.g., message authentication code, digital signature, and authenticated encryption. Nevertheless, the purpose of $\mathcal{F}_{TH}$ is to offer the *simulator* in the UC framework a special capability to extract the secret inside the enclave and well emulate an adversary's behavior. Analogously, Pass et al. [36] formalize the TEE as an attested execution secure processor $\mathcal{G}_{att}$, a global trusted setup in the UC model. Their formalization is purely theoretical but bridges the gap between design and provable security of TEE-based secure computation. As a closely related work, Choi et al. [27] propose a hybrid solution for two-party SFE by allowing one party to partly perform the protocol inside the enclave and the other party evaluate in Yao's garbled circuit. However, they have claimed that their approach is vulnerable in an improper protocol partition. Felsen et al. [28] utilize Intel SGX to realize both secure and private function evaluation. They allow the boolean circuit and universal circuit to be performed inside the enclave to mitigate side-channel attacks. Some other works are devoted to adapting TEE to cloud computing and building a privacy-preserving distributed computing framework, e.g., [30], [37], [38], [39]. However, most of these previous works consider TEE as a trusted third-party and secure to perform a function in plaintext. Instead,

HYBRTC discusses in a hybrid trust model that not all parties fully accept this building block. It is a more general case of reality and thus can be applied to many real-world scenarios.

Privacy-preserving SQL Execution. CryptDB [40] is a practical database system with a provable confidentiality guarantee in the face of a malicious administrator. It leverages a SQL-aware encryption strategy, a set of lightweight symmetric-key encryption schemes to improve the efficiency of performing a query. Poddar et al. [35] propose a maliciously-secure analytic system named Senate, which is based on the garbled circuit to enable multiple parties to run SQL queries collaboratively. They also provide several optimized circuits to ensure the system's practicality. However, these works still leave much to be desired in concrete communication and computation because of the heavy cryptographic operations. HYBRTC highlights another way to design a secure SQL system by combining the advantages of TEE and MPC. If more parties are willing to choose TEE, our countermeasure can benefit from many lightweight operations and win a significant efficiency boost. Zheng et al. propose Opaque [41], an SGX-based encrypted SQL analytics platform built on a popular distributed computing framework, Spark. They re-design several typical SQL operators using column sort to hide the memory-level access pattern. Dang et al. [42] also refine secure SQL execution in SGX by proposing scramble-then-compute (STC) framework that allows each operator to be performed following the Melbourne shuffle [43]. However, different from our work, both [41] and [42] discuss a secure join on tables maintained by a single administrator, while more challenges we are facing to handle MPC among different data holders. Hafiz et al. [44] and Wang et al. [45] introduce approaches to securely fetch a record in multiple databases using PIR. The former expresses a contextual index as a matrix and achieves information-theoretic security by secretly sharing it to databases. The latter extends the functional secret sharing (FSS) to support more complex queries, including `MAX` and `Top-K`. They also develop an optimized implementation of FSS that leverages AES-NI instructions to improve performance in modern hardware. Compared to these schemes, there is only one server used in the HYBRTC to perform a `SELECT` query, thereby not requiring secret-sharing queries and reconstructing output, which mitigates the computational burden on the client-side.

II. PRELIMINARIES

Notations. We denote $[n]$ as shorthand of a set of contiguous integers $1, \ldots, n$. We use $|m|$ to denote the size of message m, $a \leftarrow_{\$} \mathbf{S}$ to denote uniformly and independently sample a number a from a set $\mathbf{S}$, and $m_1||m_2$ to denote a concatenation of two messages m_1, m_2.

A. Trusted Execution Environment (TEE)

A TEE (e.g., ARM TrustZone, Intel SGX) is capable of creating a hardware-protected area of the main processor, called enclave, which provides strong security features including confidentiality and integrity to any code and data it stores or processes. TEEs are designed to thwart a powerful adversary by enforcing a dual-world view where even compromised or malicious systems in the normal world (a.k.a., rich operating system execution environment (REE)) cannot gain access and tamper with the secure world. In this work, we leverage Intel SGX at the server-side to offer strong support for SFE.

Intel SGX [12]. It is a set of X86-64 instruction set architecture (ISA) extensions introduced by Intel Corporation, enabling the creation of about 92MB secure memory region to store the data efficiently. Intel SGX allows an enclave to demonstrate to a remote user that the code has been instantiated securely and correctly by providing *remote attestation* mechanism. A secure and authenticated communication channel is then established via Diffie-Hellman key exchange, which can be used to transmit sensitive data securely. However, Intel SGX is vulnerable to many side-channel attacks, including in page table [15], [20], dynamic random access memory (DRAM) [21], [22], [14], and cache [47], [48], [49].

B. PSI and Distributed Join

PSI is one of the typical 2PC, the functionality of which allows two parties to compute the common items with respective input sets without revealing anything about the items not in the intersection. Analogously, in the database, a distributed join ($\bowtie$, a.k.a., natural join) is a SQL operator that combines two key-value pairs $\langle K_1, V_1\rangle, \langle K_2, V_2\rangle$ stored in different tables by concatenating them in a matched key, i.e., $K_1 = K_2$, and returns $\langle K, (V_1, V_2)\rangle$, where K is a common key. It can be considered as a general way to realize PSI because this protocol has to find all candidate pairs agreeing on the same key with a given pair from the other party. For example, two key-value pairs with the same key in both tables will return at least four items in the output table. However, only one item can be returned in PSI because the duplicated items are disallowed in a set. The following two building blocks are commonly used in constructing a PSI protocol.

Bloom Filter [50]. It is a space-efficient data structure for probabilistically testing whether an element is a member of a set. A bloom filter is an array of m bits and parameterized by k independently uniform hash functions $h_1, \ldots, h_k$ such that each h_i can map an element to m indexes uniformly. To add an element x into the filter, we first need to hash x using k hash functions and set all bits at the index of hash values to 1. To check if an item y is in a set, we also need to hash y using these k hash functions. If any bit at the index of hash values is 0, y is *definitely* not in this set; otherwise, y is *probably* in the set. The probability of this false positive is approximate $(1 - e^{-kn/m})^k$, where n is a total number of items in the set.

Cuckoo Hashing [51]. To mitigate the collision in simple hashing, cuckoo hashing is proposed to allow an element not to be hashed to only one bin. In cuckoo hashing, one can use two hash functions h_0, h_1 to map n elements to two hash tables T_0, T_1, both of which have a capacity of $(1+\epsilon)n$ bins, where ϵ is a rate of redundant storage. We allow each bin to accommodate at most one element. To assign an element x into these bins, first check to see whether any of the bins at the index of $h_0(x), h_1(x)$ are empty. If so, place x in one of the empty bins and terminates. Otherwise, randomly choose $m \in \{0, 1\}$, evict the element in the bin at the index of $h_m(x)$, and find another bin for it using h_{1-m}. This procedure is repeated until no more evictions are necessary or reaching an upper bound of relocations. The last item in the latter case will be moved to another storage, named *stash*. Previous work

$\Pi_{\mathsf{Hyb}}[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_m, f, g](1^\kappa, D_1, \ldots, D_m)$

$f_{\alpha_i}[\mathcal{C}, \mathcal{S}_i](1^\kappa, D_i), i \in [m]$

- **Complete trust**:
  1) $\mathcal{C}$ agrees on a session key $k \leftarrow_\$ \{0,1\}^\kappa$ with $\mathcal{S}_i$'s enclave;
  2) $\mathcal{C}$ encrypts α_i with k, and sends the ciphertext to $\mathcal{S}_i$;
  3) $\mathcal{S}_i$ decrypts ciphertext with k and performs $R_i \leftarrow f_{\alpha_i}(D_i)$;
- **Partial trust**:
  1) $\mathcal{C}$ shares a secret key $k \leftarrow_\$ \{0,1\}^\kappa$ with $\mathcal{S}_i$'s enclave;
  2) $\mathcal{S}_i$ computes $C_i \leftarrow \mathsf{F}_1(k, D_i)$ in the enclave;
  3) $\mathcal{C}$ computes $c_i \leftarrow \mathsf{F}_1(k, \alpha_i)$ in local;
  4) $\mathcal{C}$ and $\mathcal{S}_i$ engage in a 2PC protocol $R_i \leftarrow \mathsf{F}_2(c_i, C_i)$;

$g[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_m](1^\kappa, R_1, \ldots, R_m)$

1) $\mathcal{S}_1, \ldots, \mathcal{S}_m$ engage in an MPC protocol
$$(O_1, \ldots, O_m) \leftarrow g(1^\kappa, R_1, \ldots, R_m),$$
where O_i is a part of output received by $\mathcal{S}_i$;
2) $\mathcal{S}_i$ returns O_i to $\mathcal{C}$, and $\mathcal{C}$ combines all O_i as the final output.

Fig. 2: The HYBRTC framework for hybrid trust MPC

[52] has demonstrated that the probability of stash overflows is at most $\mathcal{O}(n^{-(s+1)})$, where the stash has a constant size s.

III. HYBRTC: A FRAMEWORK FOR HYBRID TRUST COMPUTING

The main idea of HYBRTC to evaluate a function is based on the level of trust that a party has in TEEs from others. If it is complete trust, the sensitive data of the trusting party can be simply delivered into the enclave and performed in plaintext directly. If it is distrust, an MPC protocol is necessary to compute on their inputs privately. If it is partial trust, we combine TEE and MPC together by allowing some parts of the function that rely on a secret key to be processed in TEE, while others use MPC. An overview of HYBRTC framework is shown in Fig. 2.

TEE-based two-party computation. In our hybrid trust setting, if $\mathcal{C}$ completely trusts the TEE deployed in a server $\mathcal{S}_i$, she first agrees on a session key k with $\mathcal{S}_i$'s enclave for securely delivering her input, e.g., by Diffie-Hellman key exchange. The key length is dependent on the security parameter κ. Then, she encrypts sensitive parameter α_i with k and sends the ciphertext to $\mathcal{S}_i$. Finally, $\mathcal{S}_i$ decrypts the parameter in the enclave and performs the function f on its database D_i in plaintext to obtain intermediate result R_i. If $\mathcal{C}$ partially trusts the TEE from $\mathcal{S}_i$, the function f is split into two parts, one is a keyed function F_1 that relies on a secret key k, e.g., a pseudo-random function; the other is a function F_2 without using a secret key. Initially, $\mathcal{C}$ shares k with $\mathcal{S}_i$'s enclave[1]. Then, $\mathcal{S}_i$ performs F_1 in the enclave with the help of k, and $\mathcal{C}$ processes her input α_i with k as well. Finally, they conduct a 2PC to evaluate F_2 with inputs c_i, C_i and get the result R_i.

Cryptographic multi-party computation. Consider that all servers do not trust the TEE from each other. We have them engage in a traditional cryptographic multi-party computation

[1]This can be done by establishing a secure channel that is encrypted by $\mathcal{C}$'s session key.

$\mathcal{F}_{TH}[\mathsf{Prog}, \mathsf{corrupted}, \mathcal{K}, \mathcal{P}_1, \ldots, \mathcal{P}_m]$

Parameters: $(\mathsf{mpk}, \mathsf{msk})$ is a verification and signing key pair. $\mathcal{K}$ is a set of keys used in Prog. Initially, set $\mathsf{corrupted} = \mathsf{false}$.

- Upon receiving (GETPK, sid) from $\mathcal{P}_i \in \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$:
  1) Return mpk to $\mathcal{P}_i$.
- Upon receiving $(\text{CORRUPT}, sid, \mathsf{Prog}^*)$ from $\mathcal{A}$:
  1) Set $\mathsf{corrupted} := \mathsf{true}$;
  2) **If** $\mathsf{Prog}^* \neq \emptyset$, $\mathsf{Prog} := \mathsf{Prog}^*$.
- Upon receiving $(\text{RUN}, sid, \mathsf{inp}_i, \mathsf{trusted})$ from $\mathcal{P}_i, i \in [m]$:
  1) **If** $\mathsf{corrupted} = \mathsf{true}$, **then**
     If $\mathsf{trusted} = 0$, send $(\text{INPNOTIFY}, sid, \mathsf{inp}_i, \mathcal{K})$ to $\mathcal{A}$;
     If $\mathsf{trusted} = 1$, send $(\text{INPNOTIFY}, sid, \mathsf{inp}_i)$ to $\mathcal{A}$;
     If $\mathsf{trusted} = 2$, send $(\text{INPNOTIFY}, sid, |\mathsf{inp}_i|)$ to $\mathcal{A}$;
     Else, send $(\text{INPNOTIFY}, sid, |\mathsf{inp}_i|)$ to $\mathcal{A}$;
  2) Store inp_i in local, let $\varrho_{\mathsf{inp}_i} \leftarrow \mathsf{H}(\mathsf{inp}_i)$;
  3) Await $(\text{RUN}, sid, \mathsf{inp}_j, \mathsf{trusted})$ from $\mathcal{P}_j$, let $\varrho_{\mathsf{inp}_j} \leftarrow \mathsf{H}(\mathsf{inp}_j)$; assert Prog has already installed, run
  $$(\mathsf{outp}_1, \ldots, \mathsf{outp}_m) \leftarrow \mathsf{Prog}_{\mathcal{K}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_m);$$
  4) Let $\varrho_{\mathsf{outp}_i} \leftarrow \mathsf{H}(\mathsf{outp}_i)$, and
  $$\sigma_i \leftarrow \mathsf{Sig}_{\mathsf{msk}}(sid, \mathsf{Prog}, \varrho_{\mathsf{inp}_i}, \varrho_{\mathsf{outp}_i});$$
  5) **If** $\mathsf{corrupted} = \mathsf{true}$ and $\mathsf{trusted} = 0$ or 1, **then**
     return $(\mathsf{outp}_i, \sigma_i)$ to $\mathcal{P}_i$ and $\mathcal{A}$;
     Else, return $(\mathsf{outp}_i, \sigma_i)$ to $\mathcal{P}_i$ and $(|\mathsf{outp}_i|, \sigma_i)$ to $\mathcal{A}$.

Fig. 3: The functionality of $\mathcal{F}_{TH}$

with the output of f as input. At the end of the computation, each party $\mathcal{S}_i$ is allowed to receive a part of the final output O_i. HYBRTC supports the following three typical and generic ways to realize SFE.

• *Garbled circuit*. Beaver-Micali-Rogaway (BMR) protocol [53] is a multi-party version of Yao's protocol [1]. Instead of having a single garbler know all circuit secrets, this mechanism allows each entry of the truth table to be computed by multiple servers in a secret-sharing manner, and neither party knows which semantic value is encrypted. Another party can evaluate the circuit as Yao after providing the table and wire labels of input values.

• *Secret sharing*. Assuming that all servers agree on a ring $\mathbb{Z}_p$ with a prime order p. For an input number x on each server $\mathcal{S}_i$, it uniformly samples a sequence of numbers $r_1, \ldots, r_{m-1} \leftarrow_\$ \mathbb{Z}_p$. Each number r_i is sent to one of the other servers as a share, and the last share, defined as $x - \Sigma r_i$ in additive secret sharing, is kept in local. Then, with m sets of shares from all servers, they can evaluate the arithmetic circuit of g via the protocol proposed by Ben-Or, Goldwasser, and Wigderson (BGW) [55], and output a secret-shared result.

• *Homomorphic encryption*. During initialization, $\mathcal{C}$ distributes a public key pk of homomorphic encryption to all servers. A function g can be evaluated on ciphertext directly after servers encrypt their input with pk.

Finally, $\mathcal{C}$ can obtain the final result of function by combining separate outputs from parties.

A. Multifaceted Trusted Hardware ($\mathcal{F}_{TH}$) Model

Providing formal proof for the security of TEE-based protocol is not trivial work. One may first formalize the hardware model in cryptography because of the close relationship between trusted hardware and the protocol's security. In this paper, we introduce a novel notion named multifaceted trusted hardware model $\mathcal{F}_{TH}$ that can be passively or maliciously corrupted. In $\mathcal{F}_{TH}$, we formally model the root key picked by hardware manufacturer during initialization as a pair of master keys $(\mathsf{mpk}, \mathsf{msk})$. The key mpk is a public verification key that can be obtained by any parties using a request (GETPK, sid), and msk is a secret key always kept by $\mathcal{F}_{TH}$ to sign the output of function with a signature scheme $(\mathsf{Sig}, \mathsf{Ver})$. The program performed inside the enclave is parameterized as a PPT Turing machine Prog with a set of keys $\mathcal{K}$. A boolean flag $\mathsf{corrupted}$ is used to identify whether $\mathcal{F}_{TH}$ is corrupted. The functionality of $\mathcal{F}_{TH}$ is shown in Fig. 3.

In $\mathcal{F}_{TH}$, upon receiving a message $(\text{CORRUPT}, sid, \mathsf{Prog}^*)$ from the adversary $\mathcal{A}$, $\mathcal{F}_{TH}$ first flips $\mathsf{corrupted}$ flag to true. If $\mathcal{A}$ is semi-honest, it always sets $\mathsf{Prog}^* = \emptyset$. However, when $\mathcal{A}$ is malicious, it can arbitrarily define Prog^* and $\mathcal{F}_{TH}$ will replace the initial program Prog with Prog^*. $\mathcal{F}_{TH}$ also supports multi-party SFE by calling $(\text{RUN}, sid, \mathsf{inp}_i, \mathsf{trusted})$, wherein $\mathsf{trusted} \in \{0, 1, 2\}$ identifies the level of trust that input provider $\mathcal{P}_i$ has in $\mathcal{F}_{TH}$. Under the condition that $\mathsf{corrupted}$ is true, if $\mathcal{P}_i$ distrusts $\mathcal{F}_{TH}$, i.e., $\mathsf{trusted} = 0$, $\mathcal{F}_{TH}$ will reveal the input inp_i and $\mathcal{K}$ together to $\mathcal{A}$; if $\mathcal{P}_i$ has a partial trust, i.e., $\mathsf{trusted} = 1$, $\mathcal{F}_{TH}$ only tells inp_i to $\mathcal{A}$; if $\mathcal{P}_i$ completely trusts $\mathcal{F}_{TH}$, i.e., $\mathsf{trusted} = 2$, $\mathcal{A}$ can only learn the size of inp_i. Particularly, consider that a trusted hardware is fully dominated by its localhost. We can always set the $\mathsf{trusted}$ element in RUN message from the platform to the local hardware as 2.

After that, $\mathcal{F}_{TH}$ stores inp_i in local and performs a collision-resistant hash function H to make a commitment to inp_i, so does inp_j from each $\mathcal{P}_j, j \neq i$. If Prog has ever been installed, $\mathcal{F}_{TH}$ performs a function with the help of $\mathcal{K}$ and receives $\mathsf{outp}_1, \ldots, \mathsf{outp}_m$. Finally, $\mathcal{F}_{TH}$ computes another hash digest $\varrho_{\mathsf{outp}_i}$ on outp_i and creates a signature σ_i on the session id sid, Prog, as well as all hash digests regarding $\mathcal{P}_i$'s input and output with msk. Their contents can be visible to $\mathcal{A}$ when $\mathsf{corrupted} = \mathsf{true}$ and $\mathsf{trusted} = 0$ or 1; otherwise, $\mathsf{outp}_i, \sigma_i$ can be returned to $\mathcal{P}_i$, and $\mathcal{A}$ has knowledge on the output size $|\mathsf{outp}_i|$ and σ_i.

B. Threat Model and Security Definition

In this section, we first define the threat model and then formally present the security of hybrid trust computing in the UC framework.

Threat Model. In HYBRTC, we assume that all servers are compromised by *malicious* adversaries $\mathcal{A}$ in the sense that they can arbitrarily deviate from the protocol specification to learn the private input of the other party [56]. Such an adversary can be a database administrator or an insider who has full access to the server infrastructure from misusing privileges or exploiting vulnerabilities in the operating system. It is able to see all messages sent or received by localhost. We allow $\mathcal{A}$ to perform several passive and active attacks, such as extracting the plaintext by cipher analysis and tampering with the intermediate results to learn the privacy according to the view of protocol performing [56]. Nevertheless, $\mathcal{A}$ is unable to break into the hardware physically to learn the root key. A client $\mathcal{C}$ in HYBRTC is only responsible for submitting a distributed query, verifying several signatures, and collecting the final output. We assume that she is always honest and does not collude with any server. Some other methods can be taken to thwart a corrupted client, such as access control [57] and query sensitivity detection [46], which can be orthogonal to our work, and we refer the interested reader to these literature.

$\mathcal{F}_{\mathsf{mpc}}^{f,g}[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_m, \mathcal{L}]$

Parameters: The functionality maintains a set of corruption $\mathcal{S}_c$. Initially, it is set as $\emptyset$.

- Upon receiving $(\text{CORRUPT}, sid, \mathcal{S}_i)$ from Sim, $i \in [m]$:
  1) **If** $\mathcal{S}_i \notin \mathcal{S}_c$, set $\mathcal{S}_c \leftarrow \mathcal{S}_c \cup \mathcal{S}_i$;
  2) Send $(\text{INPUT}, sid, \mathcal{S}_i, D_i)$ to Sim **if** D_i has already received.
- Upon receiving $(\text{COMPUTE}, sid, D_i)$ from $\mathcal{S}_i, i \in [m]$:
  1) **If** $\mathcal{S}_i \in \mathcal{S}_c$, notify Sim of $(\text{INPUT}, sid, \mathcal{S}_i, D_i)$; **Else**, notify Sim of $(\text{INPUT}, sid, \mathcal{S}_i, |D_i|)$;
  2) **If** having received all α_i from $\mathcal{C}$, compute $R_i \leftarrow f_{\alpha_i}(D_i)$, and $\mathsf{outp} \leftarrow g(R_1, \ldots, R_m)$;
  3) Provision some leakage information:
     - For f, send $(\text{LEAKAGE}, sid, \mathcal{L}_{f_{\alpha_i}}(D_i))$ to Sim;
     - For g, send $(\text{LEAKAGE}, sid, \mathcal{L}_g^{(i)}(R_1, \ldots, R_m))$ to Sim.
  4) Send $(\text{OUTPUTDELIVERY}, sid)$ to Sim. **If** receiving "ok" from Sim, return $(\text{OUTPUT}, sid, \mathsf{outp})$ to $\mathcal{C}$.

Fig. 4: The functionality of hybrid trust computation $\mathcal{F}_{\mathsf{mpc}}^{f,g}$

Trust Function. In the hybrid trust computing, we model the trust relationship between any party P and the enclave of a TEE-enabled party S as a trust function $\mathcal{T}$. Formally, the function is defined as follows.

Definition 1 (Trust Function)**.** *A trust function $\mathcal{T} : \mathcal{P} \times \mathcal{S} \rightarrow \{0, 1, 2\}$ is defined on a set of parties $\mathcal{P}$ participating in the hybrid trust computing, and a set of TEE-enabled parties $\mathcal{S} \subset \mathcal{P}$. For each party $P \in \mathcal{P}$, if it completely trusts the enclave of another party $S \in \mathcal{S}$, we have $\mathcal{T}(P, S) = 2$; if it only has partial trust, we have $\mathcal{T}(P, S) = 1$; if it distrusts S's enclave, say $\mathcal{T}(P, S) = 0$. Especially, we set $\mathcal{T}(S, S) = 2$ to denote a TEE-enabled party would like to completely trust itself.*

Universal Composability. Our security model is designed in the universal composability (UC) framework [31], which allows analyzing the security of each subroutine in a complicated protocol separately. If this holds, the protocol composed of these UC-secure subroutines can be inherently provably secure in the UC framework as well. In the UC framework, a protocol is formally modeled as a system of probabilistic Interactive Turing Machines (ITMs), wherein each represents the program to be run in a party. We call the process of performing a protocol in the presence of an adversary $\mathcal{A}$ the *real world*, and the *ideal world* defines a simulator Sim emulates the execution of $\mathcal{A}$ with an *ideal functionality* $\mathcal{F}$. This functionality plays the role of a "trusted party", who is able to communicate with each party in the ideal world. In the hybrid trust computing setting, we formally model the performing of a protocol using

the two-world paradigm as follows.

• *The real world execution*. A two-phase protocol π for the hybrid trust computing is an instance of our HYBRTC framework. For the function f, it can be used to instantiate Prog in the multifaceted trusted hardware $\mathcal{F}_{TH}$ model. The computation is performed as we have described in Fig. 3. $\mathcal{C}$ and $\mathcal{S}$ feed inputs into $\mathcal{F}_{TH}$, and the input and output will be leaked to $\mathcal{A}$ if $\mathcal{C}$ partially trusts $\mathcal{F}_{TH}$. For the function g, this should be performed as the standard cryptographic multi-party computation, e.g., using the garbled circuit, secret sharing, and homomorphic encryption, to allow the sensitive inputs of servers not to be leaked during execution.

• *The ideal world execution*. In the ideal world, multiple parties $\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_m$ communicate with the ideal functionality $\mathcal{F}_{\mathsf{mpc}}^{f,g}$. As depicted in Fig. 4, if the simulator Sim corrupts a server $\mathcal{S}_i$, this server will be added to the corruption set of $\mathcal{F}_{\mathsf{mpc}}^{f,g}$, and its corresponding input can be leaked to Sim immediately. For performing a multi-party computation, when the functionality receives a message (RUN, sid, D_i) identifying $\mathcal{S}_i$'s input D_i, it first checks whether $\mathcal{S}_i$ is in the set $\mathcal{S}_c$. If so, the functionality passes $(\text{INPUT}, sid, \mathcal{S}_i, D_i)$ to Sim; otherwise, only notifies Sim the size of D_i. When all inputs received, $\mathcal{F}_{\mathsf{mpc}}^{f,g}$ computes f first, and then g. Note that in the hybrid trust computing, all servers perform a function f_{α_i} in local, so they are able to learn *at least* the size of intermediate results. Here, to enable Sim to successfully emulate in the ideal world, we also allow it to know such information by providing a leakage function $\mathcal{L}$ associated with servers' input. For the function f, we offer Sim the leakage $\mathcal{L}_{f_{\alpha_i}}(D_i)$ pertaining to R_i that a server obtains in local. For the function g, we split the leakage function into m parts $\mathcal{L}_g^{(i)}, i \in [m]$ corresponding to the information breaches from the partial output O_i. After the evaluation terminates, the functionality returns $(\text{OUTPUT}, sid, \mathsf{outp})$ to $\mathcal{C}$ if Sim allows for the output delivery.

The security of hybrid trust computing is formally defined as follows.

Definition 2 (UC-security)**.** *Let a protocol π associated with a trust function $\mathcal{T}$ be an instance of* HYBRTC *framework. We say that it can UC-securely realize $\mathcal{F}_{\mathsf{mpc}}^{f,g}$ with a set of leakage functions $\mathcal{L}$, if for any PPT adversaries $\mathcal{A}$, there exists a PPT simulator* Sim*, such that none PPT environment $\mathcal{Z}$ exists to distinguish whether it contacts with π and $\mathcal{A}$ in the real world or* Sim *and $\mathcal{F}_{\mathsf{mpc}}^{f,g}$ in the ideal world. We have*

$$\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z},\mathcal{T}} \approx_c \mathsf{IDEAL}_{\mathcal{F}_{\mathsf{mpc}}^{f,g},\mathsf{Sim},\mathcal{Z},\mathcal{L}},$$

where $\approx_c$ denotes the computational indistinguishability.

IV. PRIVACY-PRESERVING DISTRIBUTED QUERY EXECUTION WITH HYBRTC

In this section, we show how to adapt HYBRTC framework to a distributed database query setting involving a client $\mathcal{C}$ and multiple servers $\mathcal{S}_1, \ldots, \mathcal{S}_m$. Our protocol can support both use cases in Section I-B by presenting privacy-preserving SQL operators, including `SELECT`, `JOIN`, and aggregate functions such as `COUNT`, addition, subtraction.

A. Select

The `SELECT` algorithm is for compacting the database by removing those elements unsatisfying a given condition in `WHERE`. The baseline method to implement the algorithm using TEE is to sequentially read each element into the enclave, label on those should be kicked out, and finally write back unlabeled elements after re-encryption. However, this solution can leak the data distribution of unlabeled elements. A system insider may observe whether an element is written back or not after reading to the enclave, which will further leak the sensitive parameter in `WHERE`.

The intuition of designing `SELECT` is to hide the sensitive parameter in `WHERE` by means of Melbourne shuffle $\mathbb{MS}$ [43]. This algorithm can be used to break the tie of data distribution between input and output without revealing any information about which element is removed or kept (see Appendix A). In HYBRTC, a privacy-preserving `SELECT` with complete trust in TEE consists of three steps, see Fig. 5. At the beginning, $\mathcal{C}$ and $\mathcal{S}$ submit their inputs α and D to $\mathcal{F}_{TH}$, respectively. After that, $\mathcal{F}_{TH}$ performs the program $\mathsf{Prog}^{\texttt{SELECT}}$. It scans through D, labels on a key-value pair "1" if it should be retained, and "0" if removed. Then, the labeled database C is obliviously shuffled using $\mathbb{MS}$ to obfuscate the data distribution. Next, each key-value pair is checked again. If it is labeled on "1", put it in R as the final output; otherwise, omit it and continue this process. Finally, the algorithm returns R to $\mathcal{C}$, while $\mathcal{S}$ receives nothing. To prevent $\mathcal{S}$ from maliciously tampering α and $\mathsf{Prog}^{\texttt{SELECT}}$, $\mathcal{C}$ is required to check the signature σ in the last step. She is able to accept the output only if the verification succeeds.

The main difference in the variant of partial trust is that we allow $\mathcal{C}$ to trust four secret keys in $\mathcal{F}_{TH}$, including a key k_1 for a pseudo-random function (PRF), a symmetric key k_2 for a semantically secure encryption scheme (Gen, Enc, Dec), as well as two keys used in Melbourne shuffle, one is for a pseudo-random permutation, and the other is for data re-encryption. Here, we have $\mathcal{C}$ and $\mathcal{F}_{TH}$ share keys k_1, k_2 by establishing a secure channel between them. Under the condition of partial trust, $\mathcal{S}$ can securely perform $\mathbb{MS}$ in the enclave like the variant of complete trust, as well as PRF and Enc. However, rather than delivering α to $\mathcal{F}_{TH}$ directly, we have $\mathcal{C}$ preprocess her input with PRF and check on c and C_k using 2PC instead. Additionally, the value V of each record is encrypted with k_2. If $\mathcal{S}$ has no idea about k_1, k_2, two strings C_k, C_v are computationally indistinguishable from the output of a truly random function and a random string, respectively [65]. This makes $\mathcal{S}$ infeasible to infer α. After receiving R from $\mathcal{F}_{TH}$, $\mathcal{C}$ can decrypt each C_v with k_2 to get the output.

By taking advantage of the trusted unit, both variants only require to scan the database twice, which runs in $\mathcal{O}(n)$, while some other data-oblivious alternatives can be more expensive, e.g., $\mathcal{O}(n \log n)$ in [41] using a combination of quicksort and bitonic sorting network.

B. Distributed Join

A distributed `JOIN` algorithm is designed to find out all elements agreeing on the same key in different databases. An efficient way to realize this functionality in a privacy-preserving manner can borrow from cryptographic PSI proto-

$\mathsf{Prog}^{\mathrm{SELECT}}$

- Upon receiving $(\textsc{Run}_1, sid, \alpha, \mathsf{trusted}_1)$ from $\mathcal{C}$ and $(\textsc{Run}_2, sid, D, \mathsf{trusted}_2)$ from $\mathcal{S}$:
  - If $\textsc{Run}_1 = \textsc{Run}_2 = \textsc{Select}$ and $\mathsf{trusted}_1 = 2$:
    1) **Foreach** $\langle K, V\rangle \in D$ **do**
       * **If** $K = \alpha, C \leftarrow \langle K, V\rangle || \text{“1”}$; **Else**, $C \leftarrow \langle K, V\rangle || \text{“0”}$.
    2) $\tilde{C} \leftarrow \mathbb{MS}(C)$;
    3) **Foreach** $(\langle K, V\rangle || r) \in \tilde{C}$ **do**
       * **If** $r = \text{“1”}, R \leftarrow \langle K, V\rangle \cup R$; **Else**, omit it and continue.
    4) return $\mathsf{outp}_1 \leftarrow R$ to $\mathcal{C}$ and $\mathsf{outp}_2 \leftarrow \emptyset$ to $\mathcal{S}$.
  - If $\textsc{Run}_1 = \textsc{Run}_2 = \textsc{Select}$ and $\mathsf{trusted}_1 = 1$:
    1) **Foreach** $\langle K, V\rangle \in D$ **do**
       a) $C_k \leftarrow \mathsf{PRF}(k_1, K), C_v \leftarrow \mathsf{Enc}(k_2, V)$;
       b) $C \leftarrow \langle C_k, C_v\rangle \cup C$.
    2) $\tilde{C} \leftarrow \mathbb{MS}(C)$;
    3) **Foreach** $\langle C_k, C_v\rangle \in \tilde{C}$ **do**
       * **If** $C_k = \alpha, R \leftarrow \langle C_k, C_v\rangle \cup R$; **Else**, omit it and continue.
    4) return $\mathsf{outp}_1 \leftarrow R$ to $\mathcal{C}$ and $\mathsf{outp}_2 \leftarrow \emptyset$ to $\mathcal{S}$.

$\pi^{\mathrm{SELECT}}_{\mathsf{2pc}}$

Inputs: Parameter α is a condition in `WHERE` from $\mathcal{C}$. Database D is a set of key-value pairs $\langle K, V\rangle$ from $\mathcal{S}$.

Initialization:

- $\mathcal{C}$ sends the program $\mathsf{Prog}^{\mathrm{SELECT}}$ to $\mathcal{F}_{TH}$ and generates two keys used in $\mathbb{MS}$ in $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{SELECT}}]$, one is for a pseudo-random permutation, and the other is for data re-encryption.
- (**Only for partial trust**): $\mathcal{C}$ and $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{SELECT}}]$ share another two keys, including a key $k_1 \leftarrow_{\$} \{0,1\}^{\kappa}$ for PRF, and a key $k_2 \leftarrow \mathsf{Gen}(1^{\kappa})$, by establishing a secure channel between them.

Protocol description:

- **Step-1: (Submit Client's Input)**. Upon receiving α from $\mathcal{Z}$, $\mathcal{C}$:
  - **(Complete trust)**: send $(\textsc{Select}, sid, \alpha, 2)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{SELECT}}]$;
  - **(Partial trust)**: compute $c \leftarrow \mathsf{PRF}(k_1, \alpha)$ and send $(\textsc{Select}, sid, c, 1)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{SELECT}}]$;
- **Step-2: (Submit Server's Input)**. Upon receiving D from $\mathcal{Z}$, $\mathcal{S}$:
  - send $(\textsc{Select}, sid, D, 2)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{SELECT}}]$;
- **Step-3: (Verify Results)**. Upon receiving (R, σ) from $\mathcal{F}_{TH}$, $\mathcal{C}$:
  1) send $(\textsc{getPK}, sid)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{SELECT}}]$, and await mpk;
  2) – **(Complete trust)**: let $\varrho_{\mathsf{inp}} \leftarrow \mathsf{H}(\alpha)$, $\varrho_{\mathsf{outp}} \leftarrow \mathsf{H}(R)$;
     – **(Partial trust)**: let $\varrho_{\mathsf{inp}} \leftarrow \mathsf{H}(c)$, $\varrho_{\mathsf{outp}} \leftarrow \mathsf{H}(R)$;
  3) assert $\mathsf{Ver}_{\mathsf{mpk}}((sid, \mathsf{Prog}^{\mathrm{SELECT}}, \varrho_{\mathsf{inp}}, \varrho_{\mathsf{outp}}), \sigma) = 1$;
  4) – **(Complete trust)**: return R to $\mathcal{Z}$;
     – **(Partial trust)**: **foreach** $\langle C_k, C_v\rangle \in R$, compute $V \leftarrow \mathsf{Dec}(k_2, C_v)$, return a set of all V to $\mathcal{Z}$;

Fig. 5: A TEE-based select protocol

col, such as ones based on garbled bloom filter [59], [60], [61], garbled circuit [62], [63], [24], and oblivious pseudo-random function (OPRF) [64], [23], [25].

In this paper, we present a novel secure distributed join algorithm on multiple databases that incurs a linear asymptotic computational overhead as state-of-the-art counterparts [23], [24], [25] but with higher concrete efficiency, see Fig. 6. Our protocol processes in two phases including forward transmission and backward transmission. At the beginning, $\mathcal{S}_1$ initializes the forward transmission phase by creating a bloom filter $\mathbf{B}_1$, a cuckoo hashing table $\mathbf{T}_1$ (with a stash $\mathbf{S}_1$) using D_1, and then delivers $\mathbf{B}_1$ to $\mathcal{S}_2$. Next, for $2 \leq i < m$, each server $\mathcal{S}_i$ filters out ineligible records in D_i using $\mathbf{B}_{i-1}$, creates $\mathbf{B}_i, \mathbf{T}_i, \mathbf{S}_i$ with the remaining, and forwards $\mathbf{B}_i$ to $\mathcal{S}_{i+1}$. After filtering out ineligible records using $\mathbf{B}_{m-1}$, $\mathcal{S}_m$ returns all the remaining to $\mathcal{S}_{m-1}$. Particularly, if a server has no record left after applying the bloom filter, which means that there is no record agreeing on the same key in databases, the protocol can output an empty set. In the backward transmission, for $2 \leq i < m$, each server $\mathcal{S}_i$ searches keys of all records received from $\mathcal{S}_{i+1}$ in $\mathbf{T}_i, \mathbf{S}_i$ to find all joinable records and aggregates their values in homomorphic encryption. Then $\mathcal{S}_i$ delivers these aggregated records backward to $\mathcal{S}_{i-1}$. After $\mathcal{S}_1$ finishes its aggregation, it returns all records to $\mathcal{C}$ as the final output. We note that all servers are possibly malicious. If a compromised $\mathcal{S}_i$ inserts only one record in $\mathbf{B}_i$, it can learn the number of aggregated records in subsequent servers that agree on the same key with this target. Therefore, another two steps **Verify Signature** are necessary to check whether Prog and some intermediate results have tampered in this process. For some aggregate functions, e.g., addition and subtraction, allowing multiple parties to privately compute over elements agreeing on the common key, we implement them utilizing Paillier cryptosystem [54], an efficient encryption scheme supporting additive homomorphic property (see Appendix B). To enable more operators such as `MIN`, `MAX`, we leave them as an avenue for future work by replacing Paillier cryptosystem with a fully homomorphic encryption scheme, e.g., Brakerski-Gentry-Vaikuntanathan (BGV) cryptosystem [58].

In detail, $\mathcal{C}$ initializes the protocol by creating two enclaves in all servers. One is for the PRF and the other is for `JOIN`. Then she produces a key k for PRF and a pair of keys (pk, sk) for Paillier cryptosystem. Two keys k and pk are delivered to PRF and `JOIN` enclave, respectively. Next, all servers preprocess each K in their databases with PRF and engage in a `JOIN` protocol that processes as follows. In **Step-1**, $\mathcal{S}_1$ constructs a bloom filter $\mathbf{B}_1$ in local by sending $(\textsc{CreateBF}, sid, D_1, 2)$ to $\mathcal{F}_{TH}$. $\mathcal{F}_{TH}$ inserts each C_k in $\mathbf{B}_1$ with ℓ hash functions $h_1, \ldots, h_\ell$. Meanwhile, $\mathcal{S}_1$ sends $(\textsc{CreateCCH}, sid, D_1, 2)$ to $\mathcal{F}_{TH}$ to construct an ℓ-ary cuckoo-hashing table $\mathbf{T}_1$ with these ℓ hash functions. To prevent the data loss, all overflowed keys are stored in the stash $\mathbf{S}_1$. Then, $\mathcal{S}_1$ sends signatures with hash digests of input and output to $\mathcal{C}$ and makes a call $(\textsc{SearchBF}, sid, \mathbf{B}_1, 0)$ to deliver $\mathbf{B}_1$ to $\mathcal{S}_2$. In **Step-2**, $\mathcal{C}$ verifies the input for constructing $\mathbf{B}_1$ and $\mathbf{T}_1$ are identical, i.e., $\varrho^{(1,1)}_{\mathsf{inp}} = \varrho^{(2,1)}_{\mathsf{inp}}$, after requesting for mpk using $(\textsc{getPK}, sid)$. In addition, she asserts that both $\sigma^{(1,1)}, \sigma^{(2,1)}$ are valid. If so, $\mathcal{C}$ allows $\mathcal{S}_2$ to filter out key-value pairs not agreeing on the same key using $\mathbf{B}_1$ by invoking $(\textsc{SearchBF}, sid, D_2, 2)$ in **Step-3**. All remaining records in a set $R_{b,2}$ are used to create a new bloom filter $\mathbf{B}_2$ and cuckoo hashing table $\mathbf{T}_2$. For subsequent servers $\mathcal{S}_i$, where $2 < i < m$, they process their databases as in $\mathcal{S}_2$, which filter out ineligible records in D_i and forward $\mathbf{B}_i$ to $\mathcal{S}_{i+1}$. In **Step-4**, after $\mathcal{S}_m$ returns $R_{b,m}$ to $\mathcal{S}_{m-1}$, $\mathcal{C}$ ensures that $\mathcal{S}_m$ does not tamper with $\mathsf{Prog}^{\mathrm{JOIN}}$ and intermediate results by verifying $\sigma^{(3,m)}$. If it succeeds, in **Step-5**, $\mathcal{S}_{m-1}$ searches in $\mathbf{T}_{m-1}$ and $\mathbf{S}_{m-1}$ individually to find out all records with matched keys in D_{m-1} by calling

$\mathsf{Prog}^{\mathrm{JOIN}}$

Upon receiving $(\textsc{Run}, sid, D, \mathsf{trusted})$ from $\mathcal{S}$:

- If $\textsc{Run} = \textsc{CreateBF}$: First, initialize a bloom filter $\mathbf{B}$ with all zeros. Then, **foreach** $\langle C_k, V\rangle \in D$, insert C_k in $\mathbf{B}$ with ℓ independent hash functions $h_1, \ldots, h_\ell$. Finally, return $\mathsf{outp} \leftarrow \mathbf{B}$ to $\mathcal{S}$;
- If $\textsc{Run} = \textsc{CreateCCH}$: First, initialize a bloom filter $\mathbf{T}$ with all zeros and a stash $\mathbf{S} = \emptyset$. Then, **foreach** $\langle C_k, V\rangle \in D$, insert C_k in $\mathbf{T}$ with ℓ independent hash functions $h_1, \ldots, h_\ell$. All overflowed C_k are stored in $\mathbf{S}$. Finally, return $\mathsf{outp} \leftarrow \mathbf{T}, \mathbf{S}$ to $\mathcal{S}$;

Upon receiving $(\textsc{Run}_1, sid, \mathbf{D}, \mathsf{trusted}_1)$ from $\mathcal{S}_1$ and $(\textsc{Run}_2, sid, D, \mathsf{trusted}_2)$ from $\mathcal{S}_2$:

- If $\textsc{Run}_1 = \textsc{Run}_2 = \textsc{SearchBF}$: First, parse $\mathbf{D} = \mathbf{B}$ and initialize a result set $R_b = \emptyset$. Then, **foreach** $\langle C_k, V\rangle \in D$, search C_k in $\mathbf{B}$ with ℓ independent hash functions $h_1, \ldots, h_\ell$. If found, let $R_b \leftarrow R_b \cup \langle C_k, \mathsf{HE.Enc}(pk, V)\rangle$; Finally, return $\mathsf{outp}_1 \leftarrow \emptyset$ to $\mathcal{S}_1$ and $\mathsf{outp}_2 \leftarrow R_b$ to $\mathcal{S}_2$.
- If $\textsc{Run}_1 = \textsc{Run}_2 = \textsc{SearchCCH}$: First, parse $\mathbf{D} = (\mathbf{T}, \mathbf{S})$ and initialize a result set $R_c = \emptyset$. Then, **foreach** $\langle C_k, \mathsf{HE.Enc}(pk, V)\rangle \in D$, search C_k in $\mathbf{T}, \mathbf{S}$ with ℓ hash functions $h_1, \ldots, h_\ell$. Let V' be the corresponding value of a found key, evaluate $C_v \leftarrow \mathsf{HE.Eval}(\mathsf{HE.Enc}(pk, V), \mathsf{HE.Enc}(pk, V'))$, $R_c \leftarrow R_c \cup \langle C_k, C_v\rangle$; Finally, return $\mathsf{outp}_1 \leftarrow R_c$ to $\mathcal{S}_1$ and $\mathsf{outp}_2 \leftarrow \emptyset$ to $\mathcal{S}_2$.

$\pi_{\mathsf{mpc}}^{\mathrm{JOIN}}$

Inputs: Databases $D_1, \ldots, D_m$ are from $\mathcal{S}_1, \ldots, \mathcal{S}_m$, respectively. Each of them is a set of key-value pairs $\langle K, V\rangle$.

Initialization:

- $\mathcal{C}$ sends a program $\mathsf{Prog}^{\mathsf{PRF}}$ of PRF to $\mathcal{F}_{TH}$. Then she generates a key $k \leftarrow_{\$} \{0,1\}^\kappa$ for PRF, and delivers it to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathsf{PRF}}]$ by establishing a secure channel between them. In $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathsf{PRF}}]$, $\mathcal{S}_i$ preprocesses each K in D_i by computing $C_k \leftarrow \mathsf{PRF}(k, K)$, $i \in [m]$.
- $\mathcal{C}$ sends $\mathsf{Prog}^{\mathrm{JOIN}}$ to $\mathcal{F}_{TH}$. Then she generates $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\kappa)$ for Paillier cryptosystem, and delivers pk to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$.

Forward transmission phase:

- **Step-1: (Create Bloom Filter and Cuckoo Hashing Table)**. Upon receiving D_1 from $\mathcal{Z}$, $\mathcal{S}_1$:
  1) send $(\textsc{CreateBF}, sid, D_1, 2), (\textsc{CreateCCH}, sid, D_1, 2)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$, and await $(\mathbf{B}_1, \sigma^{(1,1)}), (\mathbf{T}_1, \mathbf{S}_1, \sigma^{(2,1)})$;
  2) let $\varrho_{\mathsf{inp}}^{(1,1)} \leftarrow \mathsf{H}(D_1), \varrho_{\mathsf{outp}}^{(1,1)} \leftarrow \mathsf{H}(\mathbf{B}_1), \varrho_{\mathsf{inp}}^{(2,1)} \leftarrow \mathsf{H}(D_1), \varrho_{\mathsf{outp}}^{(2,1)} \leftarrow \mathsf{H}(\mathbf{T}_1, \mathbf{S}_1)$;
  3) send $(\textsc{SearchBF}, sid, \mathbf{B}_1, 0)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$ and $(\sigma^{(1,1)}, \varrho_{\mathsf{inp}}^{(1,1)}, \varrho_{\mathsf{outp}}^{(1,1)}), (\sigma^{(2,1)}, \varrho_{\mathsf{inp}}^{(2,1)}, \varrho_{\mathsf{outp}}^{(2,1)})$ to $\mathcal{C}$;
- **Step-2: (Verify Signature I)**. Upon receiving $(\sigma^{(1,1)}, \varrho_{\mathsf{inp}}^{(1,1)}, \varrho_{\mathsf{outp}}^{(1,1)}), (\sigma^{(2,1)}, \varrho_{\mathsf{inp}}^{(2,1)}, \varrho_{\mathsf{outp}}^{(2,1)})$ from $\mathcal{S}_1$, $\mathcal{C}$:
  1) send $(\textsc{GetPK}, sid)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$, and await mpk;
  2) assert $\varrho_{\mathsf{inp}}^{(1,1)} = \varrho_{\mathsf{inp}}^{(2,1)}$, $\mathsf{Ver}_{\mathsf{mpk}}((sid, \mathsf{Prog}^{\mathrm{JOIN}}, \varrho_{\mathsf{inp}}^{(1,1)}, \varrho_{\mathsf{outp}}^{(1,1)}), \sigma^{(1,1)}) = 1$, and $\mathsf{Ver}_{\mathsf{mpk}}((sid, \mathsf{Prog}^{\mathrm{JOIN}}, \varrho_{\mathsf{inp}}^{(2,1)}, \varrho_{\mathsf{outp}}^{(2,1)}), \sigma^{(2,1)}) = 1$. Abort if any of them is not valid; otherwise, send "$\textsc{SigCheckedI}$" to $\mathcal{S}_2$;
- **Step-3: (Filter out Ineligible Records using Bloom Filter)**. Upon receiving D_i from $\mathcal{Z}$ and "$\textsc{SigCheckedI}$" from $\mathcal{C}$, $\mathcal{S}_i, 2 \le i \le m$:
  1) send $(\textsc{SearchBF}, sid, D_i, 2)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$, and await $(R_{b,i}, \sigma^{(3,i)})$;
  2) – **if** $2 \le i < m$, abort and return an empty set to $\mathcal{C}$ if there is no record in $R_{b,i}$; otherwise, create $\mathbf{B}_i, \mathbf{T}_i, \mathbf{S}_i$ with $R_{b,i}$ as in **Step-1**, send corresponding signatures and hash digests to $\mathcal{C}$ as in **Step-2** but let $\mathcal{S}_{i+1}$ be the "$\textsc{SigCheckedI}$" receiver;
     – **if** $i = m$, let $\varrho_{\mathsf{inp}}^{(3,i)} \leftarrow \mathsf{H}(D_i), \varrho_{\mathsf{outp}}^{(3,i)} \leftarrow \mathsf{H}(R_{b,i})$, send $(\textsc{SearchCCH}, sid, R_{b,i}, 0)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$ and $(\sigma^{(3,i)}, \varrho_{\mathsf{inp}}^{(3,i)}, \varrho_{\mathsf{outp}}^{(3,i)})$ to $\mathcal{C}$;
- **Step-4: (Verify Signature II)**. Upon receiving $(\sigma^{(3,m)}, \varrho_{\mathsf{inp}}^{(3,m)}, \varrho_{\mathsf{outp}}^{(3,m)})$ from $\mathcal{S}_m$, $\mathcal{C}$ asserts $\mathsf{Ver}_{\mathsf{mpk}}((sid, \mathsf{Prog}^{\mathrm{JOIN}}, \varrho_{\mathsf{inp}}^{(3,m)}, \varrho_{\mathsf{outp}}^{(3,m)}), \sigma^{(3,m)}) = 1$, and sends "$\textsc{SigCheckedII}$" to $\mathcal{S}_{m-1}$ if it is valid; otherwise, aborts execution.

Backward transmission phase:

- **Step-5: (Find All Joinable Records using Cuckoo Hashing)**. Upon receiving "$\textsc{SigCheckedII}$" or "$\textsc{SigCheckedIII}$" from $\mathcal{C}$, $\mathcal{S}_i, i \in [m-1]$:
  1) send $(\textsc{SearchCCH}, sid, \mathbf{T}_i, \mathbf{S}_i, 2)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$, await $(R_{c,i}, \sigma^{(4,i)})$, and let $\varrho_{\mathsf{inp}}^{(4,i)} \leftarrow \mathsf{H}(\mathbf{T}_i, \mathbf{S}_i), \varrho_{\mathsf{outp}}^{(4,i)} \leftarrow \mathsf{H}(R_{c,i})$;
  2) – **if** $2 \le i \le m-1$, send $(\sigma^{(4,i)}, \varrho_{\mathsf{inp}}^{(4,i)}, \varrho_{\mathsf{outp}}^{(4,i)})$ to $\mathcal{C}$ and $(\textsc{SearchCCH}, sid, R_{c,i}, 0)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathrm{JOIN}}]$;
     – **if** $i = 1$, send $(R_{c,i}, \sigma^{(4,i)}, \varrho_{\mathsf{inp}}^{(4,i)}, \varrho_{\mathsf{outp}}^{(4,i)})$ to $\mathcal{C}$;
- **Step-6: (Verify Results)**. Upon receiving $(\sigma^{(4,i)}, \varrho_{\mathsf{inp}}^{(4,i)}, \varrho_{\mathsf{outp}}^{(4,i)})$ from $\mathcal{S}_i, 2 \le i \le m-1$, or $(R_{c,i}, \sigma^{(4,i)}, \varrho_{\mathsf{inp}}^{(4,i)}, \varrho_{\mathsf{outp}}^{(4,i)})$ from $\mathcal{S}_i, i = 1$, $\mathcal{C}$:
  1) assert $\varrho_{\mathsf{inp}}^{(4,i)} = \varrho_{\mathsf{outp}}^{(2,i)}$ and $\mathsf{Ver}_{\mathsf{mpk}}((sid, \mathsf{Prog}^{\mathrm{JOIN}}, \varrho_{\mathsf{inp}}^{(4,i)}, \varrho_{\mathsf{outp}}^{(4,i)}), \sigma^{(4,i)}) = 1$. Abort if any of them is not valid;
  2) – **if** $2 \le i \le m-1$, send "$\textsc{SigCheckedIII}$" to $\mathcal{S}_{i-1}$;
     – **if** $i = 1$, **foreach** $\langle C_k, C_v\rangle \in R_{c,i}$, decrypt C_v to obtain $V \leftarrow \mathsf{HE.Dec}(sk, C_v)$, and return a set of all V to $\mathcal{Z}$.

Fig. 6: A multi-party distributed join protocol

$(\textsc{SearchCCH}, sid, \mathbf{T}_{m-1}, \mathbf{S}_{m-1}, 2)$. If found, $\mathcal{F}_{TH}$ encrypts the corresponding value and performs an aggregate function in the Paillier cryptosystem, e.g., addition and subtraction, the result of which is stored in a set $R_{c,m-1}$. Subsequently, all

servers $\mathcal{S}_i$, where $2 \leq i < m-1$, process as in $\mathcal{S}_{m-1}$ by aggregating records in $R_{c,i+1}$ with ones in $\mathbf{T}_i, \mathbf{S}_i$ to produce $R_{c,i}$, and passes it backward to $\mathcal{S}_{i-1}$. In each move, $\mathcal{C}$ confirms $\mathbf{T}_i, \mathbf{S}_i$ are not tampered in **Step-6** by determining $\varrho_{\mathsf{inp}}^{(4,i)} = \varrho_{\mathsf{outp}}^{(2,i)}$ and ensures $\sigma^{(4,i)}$ is legitimate. After $\mathcal{C}$ validates the signature from $\mathcal{S}_1$, it decrypts each value in $R_{c,1}$ with sk and returns output to $\mathcal{Z}$.

Compared to some existing solutions, our scheme mostly benefits from TEE in both communication and computation aspects. On the one hand, we allow $\mathcal{S}_i$ to preprocess each K in the database by performing PRF in initialization. If the secret key k from $\mathcal{C}$ is kept confidential under the complete or partial trust, a server can generate a random string for each key-value pair directly without performing a two-party OPRF. This can reduce much network traffic, especially when the database has a large size. On the other hand, our scheme only involves some lightweight operations, e.g., create and search in a bloom filter or cuckoo hashing table, rather than OT instances as in [23], [25]. This advantage leads to the elimination of some asymmetric operations used in OT extension, e.g., encrypt and decrypt messages in OT-bases. Let the size of all databases be n, our JOIN protocol only introduces $\mathcal{O}(pn + \Sigma_{i\in[m-1]}|\mathbf{B}_i|)$ network traffic, where $p = \frac{m(m-1)}{2}$. For each $i \in [m-1]$, $\mathcal{S}_i$ constructs a bloom filter $\mathbf{B}_i$ and transfers it to $\mathcal{S}_{i+1}$. In return, $\mathcal{S}_{i+1}$ sends $\mathcal{S}_i$ a set of pairs $R_{c,i+1}$, which is linear to the total size of databases that have made an aggregation. For the computational overhead, all servers require performing a total of $\mathcal{O}(pn(C + |\mathbf{S}_i|))$ comparisons to find out all matched keys, where C is a constant. For each key-value pair returned from $\mathcal{S}_{i+1}$, $\mathcal{S}_i$ first looks up the cuckoo-hashing table $\mathbf{T}_i$ and compares C_k with all items in bins $h_1(C_k), \ldots, h_\ell(C_k)$, which processes in a constant time. And then, $\mathcal{S}_i$ scans $\mathbf{S}_i$ and compares C_k with each item within, which suffers from a linear overhead to the size of $\mathbf{S}_i$. Previous work has shown that a stash of size $\mathcal{O}(\log n)$ ensures a negligible failure probability [66]. To mitigate this overhead, we present three optimization strategies to relieve the use of stash on each server and reduce the comparison overhead to only $\mathcal{O}(pn)$.

Optimizations. Our first optimization utilizes permutation-based hashing introduced by Pinkas et al. [67] allowing each bin to store $b - \log|\mathbf{T}|$ bits, where b is the number of bits per key and $|\mathbf{T}| = (1+\epsilon)n$ is the size of a cuckoo hashing table. Then, we show how to reduce the size of the stash by extending each bin in $\mathbf{T}$ to a constant size d for accommodating multiple elements. Finally, we introduce a technique inspired by [24] to set a constant-sized stash by empirical analysis.

• *Permutation-based hashing*. Let $K = K_L||K_R$ be the bit representation of a key K, where $|K_L| = \log|\mathbf{T}|$, i.e., $|K_L|$ is equal to the number of bins in $\mathbf{T}$, $f : [K_R] \rightarrow [K_L]$ be a random function with the purpose of mapping K to a bin $K_L \oplus f(K_R)$, which stores a value K_R. Because the length of K_R is much shorter than K, this technique can dramatically reduce the comparison overhead of each item and the server storage, especially when $|K|$ is not much greater than $|\mathbf{T}|$. The function f ensures that if two keys K, K' are mapped to the same bin such that $K_L \oplus f(K_R) = K'_L \oplus f(K'_R)$, it holds that $K_L = K'_L$ because the stored values are equal, and therefore $K = K'$. For example, we assume that $|K| = 64$ bits and $\mathbf{T}$ has 2^{50} bins, then only 14 bits for each value are stored in a bin, which is much shorter than $|K|$ in the original scheme.

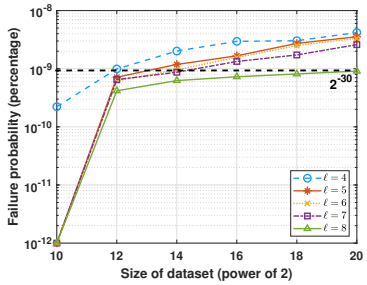


Fig. 7: Failure probability of cuckoo-hashing when mapping n elements into $1.2n$ bins using $\ell \in \{4, 5, 6, 7, 8\}$ hash functions with each bin size $d = 18$ and stash size $|\mathbf{S}| = 5$.

• *Multiple elements in a bin*. Kirsch et al. [52] have shown that the number of evictions requiring to relocate any element can be essentially bounded by the size of the largest cycle in a *cuckoo graph*. Here, it is dominated by the maximum number of key-value pairs that agree on a common key in the database. If it is much larger than the number of hash functions ℓ, the elements with an identical key will collide with a high probability. This collision will lead to a large stash size and introduce much overhead in scanning. To reduce the number of comparisons in $\mathbf{S}$, we allow each bin of $\mathbf{T}$ to accommodate d items instead of one. In this way, hash-collided keys have more opportunity to reside in $\mathbf{T}$ without being overflowed. Another advantage is that a d-size bin of cuckoo hashing does no harm to the efficiency of consulting an item. A most recent work presented by Minaud et al. [68] has demonstrated that it incurs a worst-case constant time when comparing in ℓ bins only with a probability of failure $\mathcal{O}(n^{-d-|\mathbf{S}|})$.

• *Cuckoo hashing with a constant-sized stash*. The key insight of this technique is to choose a fixed size of stash $|\mathbf{S}|$ and a fixed size of bin d, and then run an empirical evaluation for a negligible failure probability of cuckoo hashing, e.g., 2^{-30} chosen in [63], [24]. Here, the failure refers to an item cannot be accommodated in both $\mathbf{T}$ and $\mathbf{S}$. To achieve such failure probability, we choose $|\mathbf{S}| = 5, d = 18$, number of bins $|\mathbf{T}|$ as $1.2\times$ of dataset size, and make statistics in different number of hash functions ℓ ranging from four to eight, see Fig. 7. From this figure, we observe that when $|\mathbf{S}| = 5, d = 18$, the number of hash functions required for achieving 2^{-30} failure probability is drastically increased for a larger dataset — for $n = 2^{12}$, we need $\ell = 4$; for $n = 2^{14}$, we need $\ell = 6$; and for $n = 2^{20}$, we need $\ell = 8$.

C. Hybrid Trust Select-and-Join Protocol

By putting SELECT and JOIN protocols together, we propose a hybrid trust protocol for realizing select-and-join execution in m databases, see Fig. 8. In this protocol, $\mathcal{C}$ first creates two enclaves for programs $\mathsf{Prog}^{\texttt{SELECT}}, \mathsf{Prog}^{\texttt{JOIN}}$ in all servers. Then, $\mathcal{C}$ generates a pair of keys (pk, sk) for Paillier cryptosystem and a key k for PRF. Two keys pk and k are delivered to the SELECT enclave of each server. Similar to Fig. 5, another two keys for the pseudo-random permutation and data re-encryption are produced to perform Melbourne shuffle in the enclave. Next, $\mathcal{C}$ provides sensitive parameters $\alpha_1, \ldots, \alpha_m$ to servers as stated in $\pi_{\mathsf{2pc}}^{\texttt{SELECT}}$ and allows servers to perform SELECT over their databases in

$\pi_{\mathsf{mpc}}^{\mathtt{SELECT,JOIN}}$

Inputs: Parameters $\alpha_1 \ldots, \alpha_m$ are conditions in `WHERE` from $\mathcal{C}$. Each database D_i is from $\mathcal{S}_i, i \in [m]$, which is a set of key-value pairs $\langle K, V \rangle$.

Initialization:

- $\mathcal{C}$ sends programs $\mathsf{Prog}^{\mathtt{SELECT}}$ and $\mathsf{Prog}^{\mathtt{JOIN}}$ to $\mathcal{F}_{TH}$. Then $\mathcal{C}$ generates a pair of keys $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\kappa)$ for Paillier cryptosystem, a key $k \leftarrow_\$ \{0,1\}^\kappa$ for PRF, and delivers pk, k to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$ by establishing a secure channel between them. Additionally, $\mathcal{C}$ generates another two keys used in $\mathbb{MS}$ in $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$. One is for a pseudo-random permutation and the other is for the data re-encryption.
- (**Only for partial trust**): Overall, $\mathcal{C}$ requires to trust three secret keys in $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$, including k and two keys used in Melbourne shuffle.

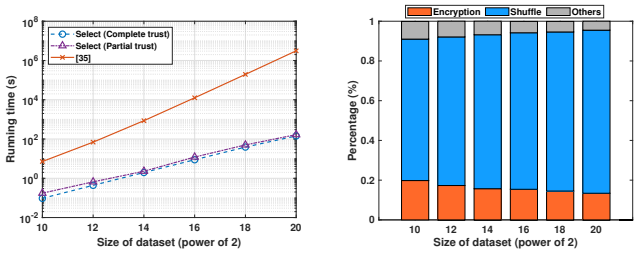
Protocol description:

- **Step-1: (Perform $\pi_{\mathsf{2pc}}^{\mathtt{SELECT}}$ on $\mathcal{C}$ and $\mathcal{S}_i, i \in [m]$)**. As shown in Fig. 5,
  1) **Step-1.1**: Upon receiving α_i from $\mathcal{Z}$, **if** $\mathcal{C}$ completely trusts $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$, she sends α_i to it; **if** $\mathcal{C}$ partially trusts $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$, she computes $c_i \leftarrow \mathsf{PRF}(k, \alpha_i)$ and sends c_i to it;
  2) **Step-1.2**: Upon receiving D_i from $\mathcal{Z}$, $\mathcal{S}_i$ sends D_i to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$.
  3) **Step-1.3**: **If** $\mathcal{S}_i$ is in complete trust, **foreach** $\langle K, V \rangle \in R_i$, $\mathcal{S}_i$ performs PRF on K by computing $C_k \leftarrow \mathsf{PRF}(k, K)$ and encrypts V by computing $C_v \leftarrow \mathsf{HE.Enc}(pk, V)$. **If** $\mathcal{S}_i$ is in partial trust, **foreach** $\langle C_k, C_v \rangle \in R_i$, $\mathcal{S}_i$ produces C_v by means of the Paillier cryptosystem.
- **Step-2: (Commit to $R_i, i \in [m]$)**. Upon receiving $(R_i, \sigma_1^{(i)})$ from $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$, $\mathcal{S}_i$ computes $\varrho_{\mathsf{inp}}^{(1,i)} \leftarrow \mathsf{H}(D_i), \varrho_{\mathsf{outp}}^{(1,i)} \leftarrow \mathsf{H}(R_i)$, and sends $(\sigma_1^{(i)}, \varrho_{\mathsf{inp}}^{(1,i)}, \varrho_{\mathsf{outp}}^{(1,i)})$ to $\mathcal{C}$.
- **Step-3: (Verify Signature)**. Upon receiving $(\sigma_1^{(i)}, \varrho_{\mathsf{inp}}^{(1,i)}, \varrho_{\mathsf{outp}}^{(1,i)})$ from $\mathcal{S}_i, i \in [m]$, $\mathcal{C}$ sends $(\textsc{getPK}, sid)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{SELECT}}]$, and awaits mpk. Then she verifies whether $\sigma_1^{(i)}$ is legitimate. If so, $\mathcal{C}$ sends "SIGCHECKEDI" to $\mathcal{S}_1$; otherwise, aborts execution.
- **Step-4: (Perform $\pi_{\mathsf{mpc}}^{\mathtt{JOIN}}$ on Servers)**. As shown in Fig. 6,
  1) **Step-4.1:** Upon receiving "SIGCHECKEDI" from $\mathcal{C}$, $\mathcal{S}_1$ creates $\mathbf{B}_1, \mathbf{T}_1, \mathbf{S}_1$ with R_1. It computes $\varrho_{\mathsf{inp}}^{(2,1)} \leftarrow \mathsf{H}(R_1), \varrho_{\mathsf{outp}}^{(2,1)} \leftarrow \mathsf{H}(\mathbf{B}_1), \varrho_{\mathsf{inp}}^{(3,1)} \leftarrow \mathsf{H}(R_1), \varrho_{\mathsf{outp}}^{(3,1)} \leftarrow \mathsf{H}(\mathbf{T}_1, \mathbf{S}_1)$, sends $\mathbf{B}_1$ to $\mathcal{S}_2$ and $(\sigma^{(2,1)}, \varrho_{\mathsf{inp}}^{(2,1)}, \varrho_{\mathsf{outp}}^{(2,1)}), (\sigma^{(3,1)}, \varrho_{\mathsf{inp}}^{(3,1)}, \varrho_{\mathsf{outp}}^{(3,1)})$ to $\mathcal{C}$;
  2) **Step-4.2:** Upon receiving $(\sigma^{(2,i)}, \varrho_{\mathsf{inp}}^{(2,i)}, \varrho_{\mathsf{outp}}^{(2,i)}), (\sigma^{(3,i)}, \varrho_{\mathsf{inp}}^{(3,i)}, \varrho_{\mathsf{outp}}^{(3,i)})$ from $\mathcal{S}_i$, $\mathcal{C}$ sends $(\textsc{getPK}, sid)$ to $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{JOIN}}]$, and awaits mpk. Then she asserts $\varrho_{\mathsf{outp}}^{(1,i)} = \varrho_{\mathsf{inp}}^{(2,i)} = \varrho_{\mathsf{inp}}^{(3,i)}$ and $\sigma^{(2,i)}, \sigma^{(3,i)}$ are valid. If so, $\mathcal{C}$ sends "SIGCHECKEDII" to $\mathcal{S}_{i+1}$; otherwise, aborts execution;
  3) **Step-4.3:** Upon receiving "SIGCHECKEDII" from $\mathcal{C}$,
     - **If** $2 \le i \le m-1$, $\mathcal{S}_i$ filters out ineligible records in R_i with $\mathbf{B}_{i-1}$, uses the remaining to create $\mathbf{B}_i, \mathbf{T}_i, \mathbf{S}_i$, and forwards $\mathbf{B}_i$ to $\mathcal{S}_{i+1}$. Then it sends corresponding signatures and hash digests to $\mathcal{C}$ as in **Step-4.2**;
     - **If** $i = m$, $\mathcal{S}_i$ randomly splits R_i into m parts $R_{i,1}, \ldots, R_{i,m}$ in $\mathcal{F}_{TH}[\mathsf{Prog}^{\mathtt{JOIN}}]$, and filters out ineligible records in $R_{i,1}$ with $\mathbf{B}_{i-1}$. Then it computes $\varrho_{\mathsf{inp}}^{(4,i)} \leftarrow \mathsf{H}(R_{i,1}), \varrho_{\mathsf{outp}}^{(4,i)} \leftarrow \mathsf{H}(R_{b,i})$, sends $R_{b,i}$ to $\mathcal{S}_{i-1}$ and $(\sigma^{(4,i)}, \varrho_{\mathsf{inp}}^{(4,i)}, \varrho_{\mathsf{outp}}^{(4,i)})$ to $\mathcal{C}$;
     - All servers conduct another $m-1$ number of $\pi_{\mathsf{mpc}}^{\mathtt{JOIN}}$ separately. Each takes $R_1, \ldots, R_{m-1}, R_{m,i}$ as inputs and lets $\mathcal{S}_i$ be the receiver of output O_i.
  4) **Step-4.4:** Upon receiving $(\sigma^{(4,m)}, \varrho_{\mathsf{inp}}^{(4,m)}, \varrho_{\mathsf{outp}}^{(4,m)})$ from $\mathcal{S}_m$, $\mathcal{C}$ verifies $\sigma^{(4,m)}$ and sends "SIGCHECKEDIII" to $\mathcal{S}_{m-1}$ if it is valid;
  5) **Step-4.5:** Upon receiving "SIGCHECKEDIII" or "SIGCHECKEDIV" from $\mathcal{C}$, $\mathcal{S}_i$ aggregates records in $R_{c,i+1}$ with ones in $\mathbf{T}_i, \mathbf{S}_i$ to get $R_{c,i}$. Then it computes $\varrho_{\mathsf{inp}}^{(5,i)} \leftarrow \mathsf{H}(\mathbf{T}_i, \mathbf{S}_i), \varrho_{\mathsf{outp}}^{(5,i)} \leftarrow \mathsf{H}(R_{c,i})$. **If** $2 \le i \le m-1$, $\mathcal{S}_i$ sends $R_{c,i}$ to $\mathcal{S}_{i-1}$ and $(\sigma^{(5,i)}, \varrho_{\mathsf{inp}}^{(5,i)}, \varrho_{\mathsf{outp}}^{(5,i)})$ to $\mathcal{C}$. **If** $i = 1$, $\mathcal{S}_i$ sends $(R_{c,i}, \sigma^{(5,i)}, \varrho_{\mathsf{inp}}^{(5,i)}, \varrho_{\mathsf{outp}}^{(5,i)})$ to $\mathcal{C}$.
- **Step-5: (Verify Results)**. Upon receiving $(\sigma^{(5,i)}, \varrho_{\mathsf{inp}}^{(5,i)}, \varrho_{\mathsf{outp}}^{(5,i)})$ from $\mathcal{S}_i, 2 \le i \le m-1$, or $(R_{c,i}, \sigma^{(5,i)}, \varrho_{\mathsf{inp}}^{(5,i)}, \varrho_{\mathsf{outp}}^{(5,i)})$ from $\mathcal{S}_i, i = 1$,
  1) $\mathcal{C}$ asserts $\varrho_{\mathsf{inp}}^{(5,i)} = \varrho_{\mathsf{outp}}^{(3,i)}$ and verifies $\sigma^{(5,i)}$. If any of them is not valid, $\mathcal{C}$ aborts execution;
  2) – **If** $2 \le i \le m-1$, $\mathcal{C}$ sends "SIGCHECKEDIV" to $\mathcal{S}_{i-1}$;
     – **If** $i = 1$, **foreach** $\langle C_k, C_v \rangle \in R_{c,i}$, $\mathcal{C}$ decrypts C_v to obtain $V \leftarrow \mathsf{HE.Dec}(sk, C_v)$. Let O_i be a set of all V. After collecting all partial outputs from other $m-1$ separate $\pi_{\mathsf{mpc}}^{\mathtt{JOIN}}$, $\mathcal{C}$ combines and returns them to $\mathcal{Z}$.

Fig. 8: A hybrid trust multi-party select-and-join protocol

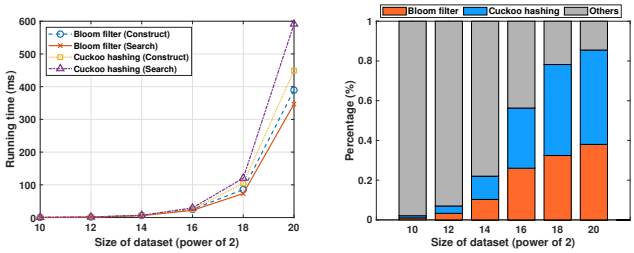
local, which outputs a set of intermediate results $R_1, \ldots, R_m$. However, consider that $R_1, \ldots, R_m$ are used as input of `JOIN`, wherein all servers do not trust each other. We have each server $\mathcal{S}_i$ in complete trust hide the real key-value pairs in R_i using PRF and Paillier cryptosystem before returning R_i from $\mathcal{F}_{TH}$. Additionally, rather than using a symmetric key encryption scheme as in `SELECT`, each server in partial trust produces C_v in Paillier cryptosystem as well. This enables two records agreeing on a common key can be aggregated in `JOIN` directly without revealing any information. Subsequently, $\mathcal{C}$ checks the hash digests and signatures of `SELECT` output to determine whether an adversary maliciously tampers with her input. If not, $\mathcal{C}$ sends SIGCHECKEDI to $\mathcal{S}_1$ and keeps hash digests of $R_1, \ldots, R_m$, which are used in integrity verification later.

The key insight to hide the final output of query $R_1 \bowtie \ldots \bowtie R_m$ from one server is to randomly split the input R_m into m parts $R_{m,1}, \ldots, R_{m,m}$ in the enclave, and perform



(a) (b)

Fig. 9: (a) Performance comparisons of three privacy-preserving select operators; (b) normalized running time of our select protocol (the variant of complete trust).



(a) (b)

Fig. 10: (a) Performance comparisons of bloom filter and cuckoo hashing; (b) normalized running time of our join protocol

$$
\begin{aligned}
&R_1 \bowtie \ldots \bowtie R_{m-1} \bowtie (R_{m,1} \cup \ldots \cup R_{m,m}) \\
&= (R_1 \bowtie \ldots \bowtie R_{m-1} \bowtie R_{m,1}) \cup \ldots \\
&\quad \cup (R_1 \bowtie \ldots \bowtie R_{m-1} \bowtie R_{m,m})
\end{aligned}
$$

instead (**Step-4.3**). We allow each $\mathcal{S}_i$ to learn about the partial output $O_i = R_1 \bowtie \ldots \bowtie R_{m-1} \bowtie R_{m,i}$ by performing JOIN on different sets of input separately. We note that there may be t records left in R_m, where t is less than m. To deal with this circumstance, $\mathcal{S}_m$ can split R_m into t parts and allow t of m servers have partial output. Alternatively, $\mathcal{C}$ can randomly choose another server that has enough records to play the role of $\mathcal{S}_m$ and reorder the message transmission among servers. After the protocol terminates, each server $\mathcal{S}_i$ returns O_i to $\mathcal{C}$, and $\mathcal{C}$ decrypts each value with the private key sk before confirming all servers' operations are legitimate.

Analogous to $\pi_{\mathsf{mpc}}^{\mathtt{JOIN}}$, to handle malicious adversary, all servers in this protocol are also required to check the integrity of input and output, as well as the correctness of the program used. For example, when the output of SELECT is used as input of JOIN, $\mathcal{C}$ will check the consistency by comparing two corresponding hash digests. The correctness can also be guaranteed by verifying the signature returned from $\mathcal{F}_{TH}$. The adversary has no idea to tamper with the program and makes a forged signature because no one knows the master key msk.

V. Security Analysis

In this section, we demonstrate the security of our scheme. We assume that once a malicious adversary corrupts a server, the underlying hardware $\mathcal{F}_{TH}$ is compromised as well, i.e., $\mathcal{F}_{TH}$ is able to collude with its localhost. We first give a formal proof that two subroutines SELECT and JOIN are able to UC-securely realize ideal functionalities of two-party SFE $\mathcal{F}_{\mathsf{2pc}}$ and multi-party SFE $\mathcal{F}_{\mathsf{mpc}}$, respectively. For the sake of coherence, we leave the definitions of $\mathcal{F}_{\mathsf{2pc}}$ and $\mathcal{F}_{\mathsf{mpc}}$ to Appendix C. And then, we show that our privacy-preserving select-and-join protocol is able to UC-securely realize $\mathcal{F}_{\mathsf{mpc}}^{f,g}$ in the $\mathcal{F}_{TH}$-hybrid model. Formally, we have

Theorem 1. *If the signature scheme used in $\mathcal{F}_{TH}$ is existentially unforgeable under chosen message attacks (EUF-CMA), the Decisional Diffie-Hellman assumption holds in the adopted algebraic group, the symmetric key encryption scheme* (Gen, Enc, Dec) *is semantically secure, the pseudo-random permutation for Melbourne shuffle and* PRF *are cryptographically secure, the* SELECT *protocol described in Fig. 5 is able to UC-securely realize $\mathcal{F}_{\mathsf{2pc}}$ in the $\mathcal{F}_{TH}$-hybrid model with the presence of a malicious adversary $\mathcal{A}$.*

Proof: The proof is given in Appendix D. ■

Theorem 2. *If the signature scheme used in $\mathcal{F}_{TH}$ is EUF-CMA,* PRF *is a cryptographically secure PRF function, $h_1, \ldots, h_\ell$,* H *are collision-resistant hash functions, and homomorphic encryption scheme* HE *is semantically secure under DCRA, the* JOIN *protocol described in Fig. 6 is able to UC-securely realize $\mathcal{F}_{\mathsf{mpc}}$ in the $\mathcal{F}_{TH}$-hybrid model with the presence of a malicious adversary $\mathcal{A}$.*

Proof: The proof is given in the full version. ■

Theorem 3. *If the signature scheme used in $\mathcal{F}_{TH}$ is EUF-CMA, the Decisional Diffie-Hellman assumption holds in the adopted algebraic group, the pseudo-random permutation for Melbourne shuffle and* PRF *are cryptographically secure, $h_1, \ldots, h_\ell$,* H *are collision-resistant hash functions, the homomorphic encryption scheme* HE *is semantically secure, the hybrid trust select-and-join protocol described in Fig. 8 is able to UC-securely realize $\mathcal{F}_{\mathsf{mpc}}^{f,g}$ in the $\mathcal{F}_{TH}$-hybrid model with the presence of a malicious adversary $\mathcal{A}$.*

Proof: The proof is given in the full version. ■

VI. Implementation

We implement a prototype of HybrTC[2] and designed protocols with the help of OpenEnclave v0.15.0, an open-sourced SDK[3] for building enclave applications in Intel SGX. We choose OpenEnclave because it provides all basic function calls to manage the life-cycle of an enclave, including data sealing and attestation, as well as some pluggable libraries for necessary language and cryptographic support. Our implementation consists of two separate programs, the client application and the enclave application in servers. Both of them are programmed in C++ language with about 4,700 LOC totally.

Client Application. The functionality of the client consists of remote attestation and system initialization for providing necessary keys. In our implementation, the SGX remote attestation relies on the Data Center Attestation Primitives (DCAP)

[2]https://github.com/HybrTC/HybrTC

[3]https://github.com/openenclave/openenclave/releases/tag/v0.15.0

TABLE I: Performance comparisons of proposed distributed join to three state-of-the-art schemes, KKRT16 [23], PSTY19 [24], and CM20 [25] (the number in the bracket is the ratio of overhead to ours).

n	Communication overhead (MB)				Computational overhead (ms)			
	KKRT16	PSTY19	CM20	**Ours**	KKRT16	PSTY19	CM20	**Ours**
2^{10}	**0.12** (0.02)	2.31 (0.37)	79.03 (12.54)	6.30	90.23 (1.10)	527.09 (6.44)	346.14 (4.23)	**81.89**
2^{12}	**0.29** (0.05)	9.22 (1.46)	79.06 (12.53)	6.31	91.17 (1.01)	1,144.19 (12.65)	376.54 (4.16)	**90.44**
2^{14}	**1.26** (0.20)	36.86 (5.83)	79.18 (12.53)	6.32	129.64 (1.13)	9,545.58 (83.06)	505.76 (4.40)	**114.92**
2^{16}	**3.90** (0.62)	147.42 (23.25)	79.65 (12.56)	6.34	242.57 (1.33)	1.39E+05 (762.76)	1,027.58 (5.65)	**181.91**
2^{18}	17.56 (2.75)	589.66 (92.28)	81.53 (12.76)	**6.39**	791.33 (1.62)	2.28E+06 (4,666.38)	3,097.40 (6.33)	**489.58**
2^{20}	60.00 (9.02)	2,358.63 (354.68)	89.04 (13.39)	**6.65**	3,279.84 (1.44)	3.59E+07 (15,795.87)	11,237.71 (4.94)	**2,273.24**

technology that enables client and enclave to agree on a 256-bit symmetric key in Elliptic-curve Diffie–Hellman (ECDH). By invoking an OpenEnclave API `oe_get_evidence()`, a server can create a quote signed in Elliptic Curve Digital Signature Algorithm (ECDSA) and a client can verify its validity by invoking `oe_verify_evidence()`. All messages between the client and enclave are encrypted in the AES-GCM algorithm with the agreed symmetric key. We also implement the key generation algorithm of Paillier cryptosystem in the client, and the security parameter we choose is 128-bit.

Enclave Application in Server. We partition the application running in servers into the *sensitive* and *insensitive* two parts. The sensitive part of the program is kept within the enclave. It consists of two handlers for executing `SELECT` and `JOIN` queries, which define Melbourne shuffle, and some algorithms for creating and searching in bloom filter and cuckoo hashing table. In our implementation, the hash function we use is the SHA512 algorithm, and the symmetric key encryption scheme in `SELECT` is the AES-GCM algorithm with a 256-bit key. They are implemented by mbedtls[4] library. For the insensitive part, we define some additional functions for I/O and message transmission.

VII. Experiments and Evaluation

In this section, we evaluate HybrTC by benchmarking `SELECT` and `JOIN` protocols and comparing them with state-of-the-art counterparts, including a maliciously-secure data analytic system Senate [35], and three alternatives in PSI, KKRT16 [23], PSTY19 [24], and CM20 [25]. In addition, we simulate and assess the efficiency of executing two SQL queries in Section I-B.

Experimental Setup. All our experiments are conducted on a machine equipped with a 3.70GHz Intel Xeon E-2176G CPU, 64GB RAM, and 440GB disk. The operating system running on the machine is Ubuntu 18.04 LTS. In our evaluation, we randomly produce a dataset for each server in the protocol, which consists of 2^{20} records, and each record is 64-bit.

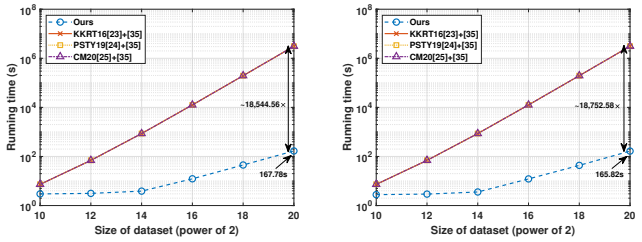
A. Performance of SQL Operators

In this experiment, we assess two privacy-preserving SQL operators `SELECT` and `JOIN`. We reproduce the filter algorithm in Senate using ABY framework[5] and Fig. 9 (a) shows the comparison result with our scheme. From this figure, we find that the proposed `SELECT` with complete trust only introduces 0.10-141.69s computational overhead, which achieves approximately 75.24-21,625.11$\times$ speedups of Senate when the dataset size ranges from 2^{10} to 2^{20} records. This efficiency gap is primarily attributable to some plaintext-based operations exploited in our approach. Under the assumption of complete trust, $\mathcal{C}$ is allowed to deliver the sensitive parameter to the enclave directly, which enables labeling on key-value pairs to be performed in plaintext. We can also find that the variant of partial trust is slightly slower, which incurs an extra 0.08-23.56s. This overhead mainly comes from the pseudo-random function and data encryption on each record, as well as network transmission for these ciphertexts. To achieve the confidentiality of `WHERE`, Senate resorts to evaluating a large garbled circuit between $\mathcal{C}$ and $\mathcal{S}$, which can be more expensive. Additionally, another PIR is required in Senate to retrieve those labeled results obliviously. Both of them incur much time cost, which is more than 3.1×10^6s in a 2^{20}-size dataset. From Fig. 9 (b), we can observe that the Melbourne shuffle dominates the performance of the `SELECT` algorithm by accounting for 71.2-82.1% of total overhead. It is due to the fact that this process requires to randomly permute the whole dataset, which suffers a lot from serialization and deserialization of intermediate array in entering and exiting an enclave (performing `ECALL` and `OCALL` functions). As the size of the dataset increases, the system demands more time to pad dummy data and re-encrypt the whole dataset, leading to a long delay for servers.

To show the practicality of our `JOIN` protocol, we choose and compare with three state-of-the-art PSI protocols as baselines [23], [24], [25]. Here, we set two servers in our protocol to have a fair comparison with these two-party PSIs. Table I reveals the comparison results in both communication and computation. From this table, we find that KKRT16 does the best in communication when the dataset size is smaller than 2^{16}, the reason for which comes from the underlying OPRF constructed by OT extension protocol [69]. It allows both servers to engage in k instances of 1-out-of-2 OT protocols on each column of two $m \times k$ matrices ($m \gg k$), a.k.a. base-OTs, and then extend the messages received to achieve m effective OTs by some symmetric operations. Our scheme requires more network bandwidth than theirs because two joint key-value pairs have to be aggregated using homomorphic encryption, which produces a much longer ciphertext. Nevertheless, as the dataset size increases, more OT instances are required, and both servers have to transfer longer messages by constructing larger matrices, which is up to 60.0MB over a 2^{20}-size dataset. Additionally, we find that both PSTY19 and CM20 are less practical than ours because the receiver in PSTY19 has to evaluate a large polynomial in programmable PRF (PPRF)

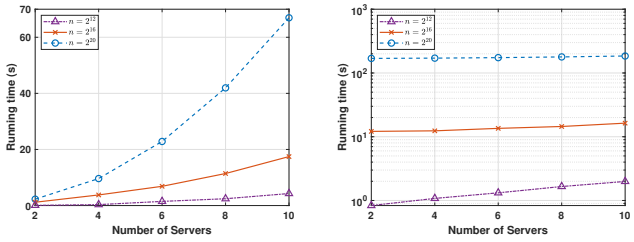
[4]https://github.com/openenclave/openenclave-mbedtls

[5]https://github.com/encryptogroup/ABY



(a) Query 1 (b) Query 2

Fig. 11: Performance comparisons of executing two SQL queries in use cases.



(a) Join (b) Select-and-join

Fig. 12: Performance comparisons of proposed schemes in a different number of servers.

constructed by the sender, the size of which is square to the dataset size. Also, despite extending to multi-point OPRF in CM20, this scheme still relies on the heavy use of OT protocol. However, our approach eliminates the use of OT and allows both servers to discard most inappropriate items locally using bloom filter, which incurs much less network communication than PSTY19 and CM20.

As shown in this table, our scheme achieves the lowest computational overhead in all sizes of the dataset. Particularly, when the size of a database is 2^{20}, the proposed JOIN only incurs 2.27s time cost that achieves four order-of-magnitude speedups compared to PSTY19. Although both PSTY19 and CM20 possess the same linear asymptotic overhead as ours, we achieve better concrete efficiency for the following two aspects. On the one hand, instead of involving in some highly compute-intensive operation, e.g., evaluate circuit in PSTY19, our JOIN protocol only leverages lightweight cryptographic primitives like hashing and PRF. On the other hand, to achieve malicious security, our scheme only needs to check a signature produced by the trusted hardware without introducing extra overhead. Hence, from the analysis above, we may believe that the proposed distributed join protocol is a good choice for the privacy-preserving SQL execution, especially for some cases with a large database.

To show the performance impact of the bloom filter and cuckoo hashing, we test the running time of constructing and searching in them with a different size of datasets, and report their take-ups in the JOIN protocol, see Fig. 10. From Fig. 10(a), we can find that constructing and searching in a cuckoo hashing table are slightly slower than in a bloom filter. For example, when the dataset size is up to 2^{20}, constructing and searching in a cuckoo hashing table approximately introduce 449.09ms and 591.42ms, respectively, while only 389.52ms and 346.93ms in a bloom filter. It is well understood because cuckoo hashing requires comparing in bins item-by-item and handling relocation in construction, which incur more time cost than operations in bits of the bloom filter. As depicted in Fig. 10(b), we observe that bloom filter and cuckoo hashing increasingly dominate the performance of JOIN when a larger dataset is taken as input. If we set the dataset size as 2^{20}, they approximately account for 38.01% and 47.57% of total overhead, respectively. It is mainly because for the purpose of accuracy, our protocol relies on filtering out ineligible records in the bloom filter and finding out all joinable records in the cuckoo hashing. The larger dataset that is taken as input, the longer time this process lasts. For some other components, such as homomorphic encryption, they have a less significant effect on the performance, which only account for 14.42% of total overhead when dataset size is 2^{20}.

B. Performance of SQL query execution

When we put SELECT and JOIN algorithms together, Fig. 11 shows the performance comparisons of two SQL queries in HYBRTC with a combination of Senate and three alternatives in PSI. Here, we set two servers in our protocol, wherein one's TEE is in complete trust, and the other is in partial trust. We simulate the execution by randomly filtering out three-quarters of elements and joining all remainings. From these figures, we can observe that HYBRTC achieves approximately 2.47-18,752.58× speedups of three purely MPC-based approaches and all baselines seem to incur the same overhead. It is mainly because a garbled circuit based filter algorithm dominates the performance of query execution, which accounts for more than 97.8% of total overhead. In HYBRTC, by taking advantage of the hybrid trust setting, we can get rid of this heavy cryptographic operation, which reduces much online overhead. In addition, we can find that Query 1 roughly incurs an extra 0.21-1.96s than Query 2. Consider that the SQL queries studied in this paper follow the same paradigm; the main difference between these two queries lies in the aggregate function. In Query 1, we accumulate values of elements that agree on the same key while simply counting their numbers in Query 2. Therefore, the efficiency difference is introduced from performing homomorphic addition. However, this overhead does no harm to the useability of the system, which only accounts for 0.1-1.2% of running time.

C. Scalability

In this experiment, we scale up the system from two to ten servers to assess the scalability of the proposed JOIN protocol and hybrid trust select-and-join protocol. We simulate the select-and-join execution by randomly filtering out three-quarters of records and joining all remainings. Additionally, half of the servers' TEEs in the select-and-join are chosen to be completely trusted, and the other half are partially trusted. As depicted in Fig. 12, we can find that both protocols scale well among multiple parties. Specifically, when we increase the number of servers from two to ten, the JOIN protocol incurs 64.68s more cost when $n = 2^{20}$, 16.51s more cost when $n = 2^{16}$, and 4.40s more cost when $n = 2^{12}$. In the select-and-join protocol, the SELECT part is performed in parallel on servers. Therefore, when we increase the number of servers from two to ten, the extra running time mainly comes from

JOIN part, which is about 15.28s, 4.20s, and 1.16s on the dataset size $n = 2^{20}, 2^{16}$, and 2^{12}, respectively.

VIII. Conclusion

In this paper, we proposed HybrTC framework, which provides a general way to realize hybrid trust multi-party computation by combining TEE and cryptographic MPC protocol. We first formalized the TEEs-like hardware by introducing a notion of multifaceted trust hardware $\mathcal{F}_{TH}$, that captures three levels of trust in TEE, including complete trust, partial trust, and distrust. To show the practicality of our scheme, we instantiated the framework in the privacy-preserving distributed query setting by presenting some typical secure SQL operations, including SELECT, JOIN, and some aggregate functions. Detailed security analysis in the UC model demonstrated that the proposed HybrTC achieves the goal of secure computing only with little tolerated privacy leakage to the malicious adversary. Finally, we benchmarked the framework in two use cases and compared it with Senate as well as three state-of-the-art alternatives in PSI. The experimental results showed that the presented select-and-join protocol outperforms all three baselines, indicating the promising efficiency of our scheme.

Acknowledgement

This research is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative, the National Natural Science Foundation of China under Grant 61972094, Grant 62032005, the Science Foundation of Fujian Provincial Science and Technology Agency (2020J02016), and the young talent promotion project of Fujian Science and Technology Association. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Andrew C. Yao, Protocols for Secure Computations (extended abstract). In *FOCS*, pp. 160-164, 1982.

[2] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority. In *STOC*, pp. 218-229, 1987.

[3] V. Kolesnikov. Gate Evaluation Secret Sharing and Secure One-round Two-party Computation. In *AsiaCrypt*, pp. 136-155, 2005.

[4] S. Zahur, M. Rosulek, and D. Evans. Two Halves Make a Whole. In *EuroCrypt*, pp. 220-250, 2015.

[5] B. Pinkas, T. Schneider, and N. P. Smart. Secure Two-Party Computation Is Practical. In *AsiaCrypt*, pp. 250-267, 2009.

[6] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *S&P*, pp. 843-862, 2017.

[7] S. Angel, H. Chen, K. Laine. PIR with Compressed Queries and Amortized Query Processing. In *S&P*, pp. 962-979, 2018.

[8] T. Duong, D. H. Phan, and N. Trieu. Catalic: Delegated PSI Cardinality with Applications to Contact Tracing. In *AsiaCrypt*, pp. 870-899, 2020.

[9] P. Mohassel, and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *S&P*, pp. 19-38, 2017.

[10] J. Katz, and L. Malka. Secure Text Processing with Applications to Private DNA Matching. In *CCS*, pp. 485-492, 2010.

[11] E. Boyle, N. Gilboa, and Y. Ishai. Breaking the Circuit Size Barrier for Secure Computation Under DDH. In *CRYPTO*, pp. 509-539, 2016.

[12] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU based Attestation and Sealing. In *HASP*, pp. 1-7, 2013.

[13] S. Lee, M-W. Shi, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, pp. 557-574, 2017.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, pp. 142-157, 2019.

[15] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, pp. 640-656, 2015.

[16] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, pp. 693-707, 2018.

[17] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, pp. 1-19, 2019.

[18] Crypto API Toolkit for Intel(R) SGX. [Online] Available: https://github.com/intel/crypto-api-toolkit. 2021.

[19] Intel Software Guard Extensions SSL. [Online] Available: https://github.com/intel/intel-sgx-ssl. 2021.

[20] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, pp. 2421-2434, 2017.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, et al. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, pp. 973-990, 2018.

[22] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, pp. 991-1008, 2018.

[23] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *CCS*, pp. 818-829, 2016.

[24] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient Circuit-Based PSI with Linear Communication. In *EuroCrypt*, pp. 122-153, 2019.

[25] M. Chase, and P. Miao. Private Set Intersection in the Internet Setting From Lightweight Oblivious PRF. In *CRYPTO*, pp. 34-63, 2020.

[26] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A-R. Sadeghi, G. Scerri, and B. Warinschi. Secure Multiparty Computation from SGX. In *FC*, pp. 477-497, 2017.

[27] J. I. Choi, D. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. B. Butler, and P. Traynor. A Hybrid Approach to Secure Function Evaluation using SGX. In *AsiaCCS*, pp. 100-113, 2019.

[28] S. Felsen, Á. Kiss, T. Schneider, and C. Weinert. Secure and Private Function Evaluation with Intel SGX. In *CCSW*, pp. 165-181, 2019.

[29] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor. Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation. In *FC*, pp. 302-318, 2016.

[30] P. Wu, Q. Shen, R. H. Deng, X. Liu, Y. Zhang, and Z. Wu. ObliDC: An SGX-based Oblivious Distributed Computing Framework with Formal Proof. In *AsiaCCS*, pp. 86-99, 2019.

[31] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, pp. 136-145, 2001.

[32] Y. Wang, and K. Yi. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD*, 2021.

[33] R. Li, M. Riedewald, and X. Deng. Submodularity of Distributed Join Computation. In *SIGMOD*, pp. 1237-1252, 2018.

[34] L. Rupprecht, W. Culhane, P. R. Pietzuch. SquirrelJoin: Network-Aware Distributed Join Processing with Lazy Partitioning. In *Proceedings of the VLDB Endowment*, 10(11): 1250-1261, 2017.

[35] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *USENIX Security*, 2021.

[36] R. Pass, E. Shi, and F. Tramèr. Formal Abstractions for Attested Execution Secure Processors. In *EuroCrypt*, pp. 260-289, 2017.

[37] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *S&P*, pp. 38-54, 2015.

[38] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Russinovich, and D. Sharma. Observing and Preventing Leakage in MapReduce. In *CCS*, pp. 1570-1581, 2015.

[39] T-T-A Dinh, P. Saxena, E-C Chang, B-C Ooi, and C. Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security*, pp. 447-462, 2015.

[40] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, pp. 85-100, 2011.

[41] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX NSDI*, pp. 283-298, 2017.

[42] H. Dang, T. T. A. Dinh, E-C. Chang, and B. C. Ooi. Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute. In *PoPETs*, 3(2017):21-38, 2017.

[43] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In *ICALP*, pp. 556-567, 2014.

[44] S. M. Harfiz, and R. Henry. Querying for queries: Indexes of queries for efficient and expressive IT-PIR. In *CCS*, pp. 1361-1373, 2017.

[45] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical Private Queries on Public Data. In *USENIX NSDI*, pp. 299-313, 2017.

[46] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. PrivApprox: Privacy-Preserving Stream Analytics. In *USENIX ATC*, pp. 659-672, 2017.

[47] J. Götzfried, M. Echert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *EuroSys*, pp. 1-6, 2017.

[48] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, pp. 605-622, 2015.

[49] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*, pp. 69-90, 2017.

[50] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. In *Communications of the ACM*, 13(7):422–426, 1970.

[51] R. Pagh, and F. F. Rodler. Cuckoo Hashing. In *Algorithm - ESA*, pp. 121-133, 2001.

[52] A. Kirsch, M. Mitzenmacher, and U. Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. In *SIAM Journal on Computing*, 39(4): 1543-1561, 2009.

[53] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. In *STOC*, pp. 503-513.

[54] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EuroCrypt*, pp. 223-238, 1999.

[55] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *STOC*, pp. 351-371, 1988.

[56] Y. Lindell. Secure Multiparty Computation for Privacy Preserving Data Mining. In *The Journal of Privacy and Confidentiality*, 1(1):59–98, 2005.

[57] C. Xiang, Y. Wu, B. Shen, M. Shen, H. Huang, T. Xu, Y. Zhou, C. Moore, X. Jin, and T. Sheng. Towards Continuous Access Control Validation and Forensics. In *CCS*, pp. 113-129, 2019.

[58] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully Homomorphic Encryption without Bootstrapping. In *ACM Transactions on Computation Theory*, pp. 1-36, 2014.

[59] C. Dong, L. Chen, and Z. Wen. When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol. In *CCS*, pp. 789-800, 2013.

[60] R. Inbar, E. Omri, and B. Pinkas. Efficient Scalable Multiparty Private Set-Intersection via Garbled Bloom Filters. In *SCN*, pp. 235-252, 2018.

[61] P. Rindal, and M. Rosulek. Improved Private Set Intersection Against Malicious Adversaries. In *EuroCrypt*, pp. 235-259, 2017.

[62] Y. Huang, D. Evans, and J. Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[63] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient Circuit-based PSI via Cuckoo Hashing. In *EuroCrypt*, pp. 125-157, 2018.

[64] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert. Mobile Private Contact Discovery at Scale. In *USENIX Security*, pp. 1447-1464, 2019.

[65] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions (Extended Abstract). In *FOCS*, pp. 464-479, 1984.

[66] M. T. Goodrich, and M. Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *ICALP*, pp. 576-587, 2011.

[67] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private Set Intersection using Permutation-based Hashing. In *USENIX Security*, pp. 515-530, 2015.

[68] B. Minaud, and C. Papamanthou. Note on Generalized Cuckoo Hashing with a Stash. In arXiv:2010.01890, 2020. [Online] Available: https://arxiv.org/abs/2010.01890.

[69] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, pp. 145-161, 2003.

APPENDIX A
MELBOURNE SHUFFLE

Melbourne shuffle [43] randomly permutes n elements in an input set I to an output set O. It processes in two phases, including a *distribution* phase and a *cleanup* phase. Another array T of size $\mathcal{O}(n \log n)$ is leveraged for storing intermediate results.

In the distribution phase, the algorithm randomly samples a permutation $\omega : \{0,1\}^{\sqrt{n}} \times \{0,1\}^{\kappa} \rightarrow \{0,1\}^{\sqrt{n}}$ associated with a key k and continuously reads batches of $\sqrt{n}$ elements from I. For each element x in a batch, the algorithm puts it into another bucket $\omega(k, x)$ in T after re-encryption. According to the balls-into-bins model, a set of $p \log n$ encrypted elements from each batch is written into a bucket in T, where p is a constant. If there are less than $p \log n$ elements in one set shuffling to a bucket, dummy data is necessary to pad that set to $p \log n$. Here, the padded dummy element should be the same length as the real element to make an adversary indistinguish any two of them after data encryption. On the other hand, if a set has more than $p \log n$ elements, excessive elements will be overflowed and the algorithm cannot ensure data-obliviousness. Fortunately, its probability can be negligible for some well-chosen parameters, or a user can simply try some other permutations by starting the algorithm anew. In the cleanup phase, the algorithm reads each bucket from T, filters out all dummy data, and arranges remaining elements to their final locations in O. For the design details about the Melbourne shuffle, we may refer the reader to [43].

APPENDIX B
PAILLIER CRYPTOSYSTEM

The Paillier cryptosystem [54] is a probabilistic asymmetric encryption scheme that supports additive homomorphic encryption whose security is designed on the hardness of decisional composite residuosity assumption (DCRA). Paillier cryptosystem contains a set of four PPT algorithms ($\mathsf{HE.KeyGen}, \mathsf{HE.Enc}, \mathsf{HE.Eval}, \mathsf{HE.Dec}$), which are defined as follows.

- $(pk, sk) \leftarrow \mathsf{HE.KenGen}(1^{\kappa})$ is a probabilistic algorithm to generate a pair of asymmetric keys (pk, sk). Given a security parameter κ, the algorithm randomly samples two large prime numbers p, q, and computes $n = p \cdot q$. It then

selects a random integer g from a group $\mathbb{Z}_{n^2}^*$ such that the order of g is at least n. Lastly, the algorithm returns a tuple (n, g) as the public key pk, and $\lambda = \text{lcm}(p-1, q-1)$ as the secret key sk, where $\text{lcm}(\cdot,\cdot)$ means the least common multiple of two input numbers.

- $C \leftarrow \mathsf{HE.Enc}(pk, m)$ is a probabilistic algorithm to encrypt a given message $m \in \mathbb{Z}_n$ with pk. It randomly samples a number $r \leftarrow_{\$} \mathbb{Z}_n^*$, and computes the ciphertext as $C = g^m \cdot r^n \mod n^2$.

- $C' \leftarrow \mathsf{HE.Eval}(C_1, C_2)$ is a deterministic algorithm to compute on two ciphertexts C_1, C_2 based on the additive homomorphic property. The Paillier cryptosystem supports the following addition computation on ciphertext.

$$\begin{aligned}&\mathsf{HE.Enc}(pk, m_1) \cdot \mathsf{HE.Enc}(pk, m_2)\\ &= (g^{m_1} \cdot r_1^n) \cdot (g^{m_2} \cdot r_2^n) \mod n^2\\ &= g^{m_1+m_2} \cdot (r_1 r_2)^n \mod n^2\\ &= \mathsf{HE.Enc}(pk, m_1 + m_2).\end{aligned}$$

- $m \leftarrow \mathsf{HE.Dec}(sk, C)$ is a deterministic algorithm to decrypt a ciphertext $C \in \mathbb{Z}_{n^2}$ with sk. The plaintext can be solved by computing $\frac{L(C^{sk} \mod n^2)}{L(g^{sk} \mod n^2)} \mod n$, where $L(x) = \frac{x-1}{n}$.

Definition 3 (Semantic security)**.** *The Paillier cryptosystem is semantically secure under the DCRA, if for any PPT adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that*

$$\Pr\begin{bmatrix}(pk, sk) \leftarrow \mathsf{HE.KenGen}(1^\kappa);\\ (m_0, m_1) \leftarrow \mathcal{A}(pk); b \leftarrow_{\$} \{0,1\}; \quad : b = b'\\ C \leftarrow \mathsf{HE.Enc}(pk, m_b); b' \leftarrow \mathcal{A}(C);\end{bmatrix} \leq \mathsf{negl}(\kappa)$$

APPENDIX C
IDEAL FUNCTIONALITIES $\mathcal{F}_{\mathsf{2PC}}$ AND $\mathcal{F}_{\mathsf{MPC}}$

Fig. 13 defines an ideal functionality of two-party SFE. Upon receiving a corruption message $(\text{CORRUPT}, sid, \mathcal{P}_i)$ from Sim, $\mathcal{F}_{\mathsf{2pc}}$ leaks $\mathcal{P}_i$'s input inp_i to Sim. For evaluating a function, when $\mathcal{F}_{\mathsf{2pc}}$ receives a message $(\text{RUN}, sid, \mathsf{inp}_1)$, it first checks whether $\mathcal{P}_1$ is corrupted. If not, the size of $\mathcal{P}_1$'s input will be sent to Sim. After having received another input inp_2 from $\mathcal{P}_2$, $\mathcal{F}_{\mathsf{2pc}}$ evaluates f, and notifies $\mathcal{P}_i$ of outp_i after being permitted by Sim. In the multi-party version of SFE (Fig. 14), upon receiving a corruption message, $\mathcal{F}_{\mathsf{mpc}}$ reveals input of the corrupted party to Sim. If having received all inputs, the functionality performs a function g on them. Similar to the $\mathcal{F}_{\mathsf{mpc}}^{f,g}$ in Fig. 4, we also allow Sim to learn some information regarding the size of intermediate results that can be obtained by $\mathcal{A}$ in the real world. It is defined in the leakage function $\mathcal{L}_g^{(i)}(\mathsf{inp}_1, \ldots, \mathsf{inp}_m)$. After Sim permits the output delivery, $\mathcal{F}_{\mathsf{mpc}}$ notifies outp_i to $\mathcal{P}_i$.

APPENDIX D
SECURITY PROOF

Theorem 1 Revisited. *If the signature scheme used in $\mathcal{F}_{TH}$ is existentially unforgeable under chosen message attacks (EUF-CMA), the Decisional Diffie-Hellman assumption holds in the adopted algebraic group, the symmetric key encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is semantically secure, the pseudo-random permutation for Melbourne shuffle and* PRF *are cryptographically secure, the* SELECT *protocol described in Fig. 5 is able to UC-securely realize $\mathcal{F}_{\mathsf{2pc}}$ in the $\mathcal{F}_{TH}$-hybrid model with the presence of a malicious adversary $\mathcal{A}$.*

$\mathcal{F}_{\mathsf{2pc}}[f, \mathcal{P}_1, \mathcal{P}_2]$

- **Upon** receiving $(\text{CORRUPT}, sid, \mathcal{P}_i), i \in \{1, 2\}$ from Sim:
  1) Send $(\text{INPUT}, sid, \mathcal{P}_i, \mathsf{inp}_i)$ to Sim if inp_i has already received.
- **Upon** receiving $(\text{COMPUTE}, sid, \mathsf{inp}_1)$ from $\mathcal{P}_1$:
  1) If $\mathcal{P}_1$ has not been corrupted, notify Sim of $(\text{INPUT}, sid, |\mathsf{inp}_1|)$;
  2) If having received $(\text{COMPUTE}, sid, \mathsf{inp}_2)$ from $\mathcal{P}_2$, compute $(\mathsf{outp}_1, \mathsf{outp}_2) \leftarrow f(\mathsf{inp}_1, \mathsf{inp}_2)$;
  3) Send $(\text{OUTPUTDELIVERY}, sid)$ to Sim. If receiving "ok" from Sim, send $(\text{OUTPUT}, sid, \mathsf{outp}_i)$ to $\mathcal{P}_i$.

Fig. 13: The functionality of two-party SFE $\mathcal{F}_{\mathsf{2pc}}$

$\mathcal{F}_{\mathsf{mpc}}[g, \mathcal{P}_1, \ldots, \mathcal{P}_m]$

- **Upon** receiving $(\text{CORRUPT}, sid, \mathcal{P}_i), i \in [m]$ from Sim:
  1) Send $(\text{INPUT}, sid, \mathcal{P}_i, \mathsf{inp}_i)$ to Sim if inp_i has already received.
- **Upon** receiving $(\text{COMPUTE}, sid, \mathsf{inp}_1)$ from $\mathcal{P}_1$:
  1) If $\mathcal{P}_1$ has not been corrupted, notify Sim of $(\text{INPUT}, sid, |\mathsf{inp}_1|)$;
  2) If having received $(\text{COMPUTE}, sid, \mathsf{inp}_i)$ from all $\mathcal{P}_i$, compute $(\mathsf{outp}_1, \ldots, \mathsf{outp}_m) \leftarrow g(\mathsf{inp}_1, \ldots, \mathsf{inp}_m)$;
  3) Send $(\text{LEAKAGE}, sid, \mathcal{L}_g^{(i)}(\mathsf{inp}_1, \ldots, \mathsf{inp}_m))$ to Sim;
  4) Send $(\text{OUTPUTDELIVERY}, sid)$ to Sim. If receiving "ok" from Sim, send $(\text{OUTPUT}, sid, \mathsf{outp}_i)$ to $\mathcal{P}_i$.

Fig. 14: The functionality of multi-party SFE $\mathcal{F}_{\mathsf{mpc}}$

Proof: We construct a simulator Sim in the ideal world that internally runs a copy of $\mathcal{A}$. Any messages between $\mathcal{Z}$ and $\mathcal{A}$ or between $\mathcal{A}$ and $\mathcal{F}_{TH}$ are simply forwarded to Sim. Also, Sim simulates the interface of $\mathcal{F}_{TH}$ and honest parties. In detail, according to whether $\mathcal{S}$ has been corrupted, we consider the following two cases

Case 1. *$\mathcal{S}$ is corrupted, while $\mathcal{C}$ is honest.*

- If $\mathcal{C}$ has complete trust in $\mathcal{F}_{TH}$, upon receiving $(\text{INPUT}, sid, |\alpha|)$ from $\mathcal{F}_{\mathsf{2pc}}$, Sim randomly generates a parameter α^* as canonical input of $\mathcal{C}$ such that $|\alpha^*| = |\alpha|$, and then sends $(\text{COMPUTE}, sid, \alpha^*)$ to $\mathcal{F}_{TH}$. After receiving $(\text{SELECT}, sid, D, 2)$ from $\mathcal{S}$ to $\mathcal{F}_{TH}$, due to the fact that $\mathcal{S}$ has been corrupted, Sim can extract its real input D and forwards it to $\mathcal{F}_{\mathsf{2pc}}$. Using α^* and D, Sim acts as $\mathcal{F}_{TH}$ to compute SELECT algorithm $\mathsf{outp}^* \leftarrow f_{\alpha^*}(D)$ by picking a pseudo-random permutation ω for Melbourne shuffle, labeling on D, and filtering out items unsatisfying a predicate α^*. Finally, Sim leaks $|\mathsf{outp}^*|$ to $\mathcal{A}$. Upon receiving $(\text{OUTPUTDELIVERY}, sid)$ from $\mathcal{F}_{\mathsf{2pc}}$, Sim allows it to pass outp to $\mathcal{C}$ in the ideal world.

- If $\mathcal{C}$ has partial trust in $\mathcal{F}_{TH}$, the difference in the simulation is that after generating a canonical input α^* such that $|\alpha^*| = |\alpha|$, Sim randomly picks a string c^* for it using PRF. Then Sim forwards c^* to $\mathcal{A}$. Using c^* and D, Sim acts as $\mathcal{F}_{TH}$ to compute SELECT algorithm $\mathsf{outp}^* \leftarrow f_{c^*}(D)$ by preprocessing each key-value pair in D with PRF and a

symmetric key encryption scheme, picking a pseudo-random permutation ω for Melbourne shuffle, and filtering out ineligible records. Finally, Sim reveals outp^* to both $\mathcal{C}$ and $\mathcal{A}$.

We first prove the indistinguishability of real world and ideal world for the complete trust case through a sequence of hybrid worlds.

Claim 1. *Assuming that the signature scheme used in $\mathcal{F}_{TH}$ is EUF-CMA, a malicious $\mathcal{A}$ cannot forge a valid signature except with negligible probability $\mathsf{Adv}_{\mathsf{Sig}}(\mathcal{A}, \kappa)$.*

Proof: Straightforward reduction to the security of digital signature. If $\mathcal{A}$ has no idea about the secret key msk of $\mathcal{F}_{TH}$, $\mathcal{Z}$ can distinguish between two worlds only if $\mathcal{A}$ can forge a signature attributed with its input and output. This happens at most with $\mathsf{Adv}_{\mathsf{Sig}}(\mathcal{A}, \kappa)$, the probability of breaking unforgeability assumption if $\mathcal{A}$ pre-obtains at most a polynomial number of chosen-text signatures on κ. ■

Hybrid $\mathcal{H}_0$. *This is the simulated execution as described above.*

Hybrid $\mathcal{H}_{1a}$. *Identical to $\mathcal{H}_0$, but the secret key used in the secure channel is agreed on via Diffie-Hellman key exchange.*

Claim 2. *Assuming that the DDH assumption is hard, then $\mathcal{H}_{1a}$ is computationally indistinguishable from $\mathcal{H}_0$.*

Proof: The security is straightforwardly reduced to the DDH assumption. We can show that if there exists a PPT adversary who is able to distinguish $\mathcal{H}_0$ and $\mathcal{H}_{1a}$, then we can construct another adversary who can break the hardness of DDH assumption with advantage $\mathsf{Adv}_{\mathsf{DDH}}(\mathcal{A}, \kappa)$. ■

Hybrid $\mathcal{H}_{2a}$. *Identical to $\mathcal{H}_{1a}$ except that instead of sending a ciphertext of canonical input α^* to $\mathcal{S}$'s enclave, Sim now sends the honest $\mathcal{C}$'s real input α.*

Claim 3. *Assuming that the communication channel between $\mathcal{C}$ and $\mathcal{S}$'s enclave is encrypted in a semantically secure encryption scheme, then $\mathcal{H}_{2a}$ is computationally indistinguishable from $\mathcal{H}_{1a}$.*

Proof: Straightforward reduction to the semantic security. Since $|\alpha^*| = |\alpha|$, any PPT adversaries only has a negligible advantage $\mathsf{Adv}_{\mathsf{SS}}(\mathcal{A}, \kappa)$ to distinguish ciphertexts of α^*, α that encrypted with the same secret key. Therefore, $\mathcal{A}$ only has at most probability of $\mathsf{Adv}_{\mathsf{SS}}(\mathcal{A}, \kappa)$ to distinguish $\mathcal{H}_{1a}$ and $\mathcal{H}_{2a}$ from extracting useful information in ciphertext. ■

Hybrid $\mathcal{H}_3$. *Identical to $\mathcal{H}_{2a}$ except that instead of using a pseudo-random permutation ω, Sim obliviously shuffles input database D with a truly random permutation.*

Claim 4. *Assuming that ω is cryptographically secure, then $\mathcal{H}_3$ is computationally indistinguishable from $\mathcal{H}_{2a}$.*

Proof: Straightforward reduction to the security of a pseudo-random permutation, which guarantees output of the permutation is identically distributed to a truly random one, such that any PPT adversaries can distinguish them only with a negligible advantage $\mathsf{Adv}_{\mathsf{PRP}}(\mathcal{A}, \kappa)$. Based on this, $\mathcal{Z}$ only has at most $\mathsf{Adv}_{\mathsf{PRP}}(\mathcal{A}, \kappa)$ probability to distinguish $\mathcal{H}_{2a}$ and $\mathcal{H}_3$ from learning access pattern during shuffling process. ■

Therefore, for the complete trust variant of `SELECT` protocol, $\mathcal{A}$'s view of $\mathcal{H}_3$ is identical to the simulated execution except with a negligible sum of distinguishing advantages such that $\mathsf{Adv}_{\mathsf{Sig}}(\mathcal{A}, \kappa) + \mathsf{Adv}_{\mathsf{DDH}}(\mathcal{A}, \kappa) + \mathsf{Adv}_{\mathsf{SS}}(\mathcal{A}, \kappa) + \mathsf{Adv}_{\mathsf{PRP}}(\mathcal{A}, \kappa) = \mathsf{negl}(\kappa)$.

Then we show the indistinguishability of real world and ideal world for the partial trust case. The difference in the proof is to replace the following two hybrids $\mathcal{H}_{1b}, \mathcal{H}_{2b}$ with $\mathcal{H}_{1a}, \mathcal{H}_{2a}$.

Hybrid $\mathcal{H}_{1b}$. *Identical to $\mathcal{H}_0$, but instead of performing a pseudo-random function PRF on a key K, Sim picks a truly random number for it over $\{0,1\}^{|K|}$.*

Claim 5. *If $\mathsf{PRF} : \{0,1\}^{|K|} \times \{0,1\}^{\kappa} \to \{0,1\}^{|K|}$ is a cryptographically secure PRF with adversarial distinguishing advantage $\mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \kappa)$, $\mathcal{H}_0$ and $\mathcal{H}_{1b}$ are indistinguishable with distinguishing advantage $|R| \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \kappa)$.*

Proof: The security is straightforwardly reduced to the pseudo-random function. Let RF denote a truly random function over $\{0,1\}^{|K|}$ to $\{0,1\}^{|K|}$. If having no idea about k, any PPT adversary is impractical to distinguish $\mathsf{PRF}(k, K)$ and $\mathsf{RF}(K)$ with non-negligible advantage $\mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \kappa)$. In `SELECT` protocol, there are $|R|$ key-value pairs returned to $\mathcal{C}$. Hence, $\mathcal{H}_{1b}$ is computationally indistinguishable from $\mathcal{H}_0$ except with $|R| \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \kappa)$ probability. ■

Hybrid $\mathcal{H}_{2b}$. *Identical to $\mathcal{H}_{1b}$ except that instead of encrypting a real value V in $\mathcal{S}$'s enclave, Sim now encrypts a random value V' but with the same length.*

Claim 6. *Assuming that the symmetric key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ used is semantically secure, then $\mathcal{H}_{2b}$ is computationally indistinguishable from $\mathcal{H}_{1b}$.*

Proof: Similar to the proof of Claim 3. The security is straightforwardly reduced to the semantic security. If two messages are the same length, any PPT adversaries only has a negligible advantage $\mathsf{Adv}_{\mathsf{SS}}(\mathcal{A}, \kappa)$ to distinguish ciphertexts of them encrypted with the same secret key. Consider that there are $|R|$ records in the output set. So the probability to distinguish $\mathcal{H}_{1b}$ and $\mathcal{H}_{2b}$ is at most $|R| \cdot \mathsf{Adv}_{\mathsf{SS}}(\mathcal{A}, \kappa)$ for a PPT adversary $\mathcal{A}$. ■

Therefore, for the partial trust variant of `SELECT` protocol, $\mathcal{A}$'s view of $\mathcal{H}_3$ is identical to the simulated execution except with a negligible sum of distinguishing advantages such that $\mathsf{Adv}_{\mathsf{Sig}}(\mathcal{A}, \kappa) + |R| \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \kappa) + |R| \cdot \mathsf{Adv}_{\mathsf{SS}}(\mathcal{A}, \kappa) + \mathsf{Adv}_{\mathsf{PRP}}(\mathcal{A}, \kappa) = \mathsf{negl}(\kappa)$.

Case 2. *Both $\mathcal{C}$ and $\mathcal{S}$ are honest.*

In this case, Sim simulates both $\mathcal{C}$ and $\mathcal{S}$ by randomly generating inputs for them and then forwarding to $\mathcal{F}_{\mathsf{2pc}}$. Afterwards, $\mathcal{F}_{\mathsf{2pc}}$ returns an output to $\mathcal{C}$. It is trivial to see that two worlds in this case can be computationally indistinguishable by taking advantage of secure channel in the protocol.

This concludes the proof of Theorem 1. ■