# MPCCache: Privacy-Preserving Multi-Party Cooperative Cache Sharing at the Edge

Duong Tung Nguyen* Ni Trieu*

March 10, 2021

**Abstract**

Edge computing and caching have emerged as key technologies in the future communication network to enhance the user experience, reduce backhaul traffic, and enable various Internet of Things applications. Different from conventional resources like CPU and memory that can be utilized by only one party at a time, a cached data item, which can be considered as a public good, can serve multiple parties simultaneously. Therefore, instead of independent caching, it is beneficial for the parties (e.g., Telcos) to cooperate and proactively store their common items in a shared cache that can be accessed by all the parties at the same time.

In this work, we present MPCCache, a novel privacy-preserving **M**ulti-**P**arty **C**ooperative **Cache** sharing framework, which allows multiple network operators to determine a set of common data items with the highest access frequencies to be stored in their capacity-limited shared cache while guaranteeing the privacy of their individual datasets. The technical core of our MPCCache is a new construction that allows multiple parties to compute a specific function on the intersection set of their datasets, without revealing the intersection itself to any party.

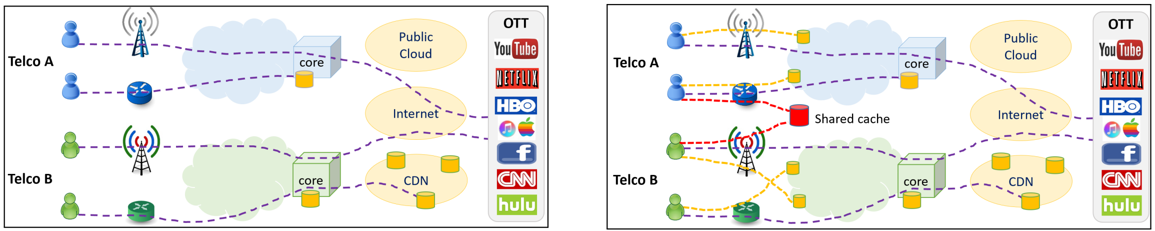
We evaluate our protocols to demonstrate their practicality and show that MPCCache scales well to large datasets and achieves a few hundred times faster compared to a baseline scheme that optimally combines existing MPC protocols.

## 1 Introduction

The explosive growth of data traffic due to the proliferation of wireless devices and bandwidth-hungry applications such as video streaming, mobile gaming, and social networking leads to an increasing capacity demand across wireless networks to enable scalable wireless access with high quality of service (QoS). This trend will likely continue for the near future due to the emergence of new applications like augmented/virtual reality, multi-view content, 4K/8K UHD video, hologram, and tactile Internet [ETS19]. Thus, it is imperative for mobile operators to develop cost-effective solutions to meet the soaring traffic demand and diverse requirements of various services in the next generation communication network.

Enabled by the drastic reduction in data storage cost, edge caching has appeared as one of the most promising technologies to tackle the aforementioned challenges in wireless networks [BBD14]. In practice, many users in the same service area may request similar content such as highly-rated Netflix movies, viral YouTube videos, breaking news, mobile operating system and application updates, maps, and weather information. Additionally, the majority of user requests are associated

*Arizona State University, {duongnt,nitrieu}@asu.edu



(a) Current Model.

(b) Edge Caching Model.

Figure 1: The current caching model and the new edge caching model. In the current model, caches are located in public clouds, CDNs, and mobile core, which are typically hundreds to thousands of miles from end-users. In edge caching, caches are placed in the network edge (e.g., at base stations, mini DCs, edge clouds) in the proximity of end-users.

with a small amount of popular content. Hence, by proactively caching popular content at the network edge (e.g., at base stations, telecom central offices, edge clouds) in advance during off-peak times, a portion of requests during peak hours can be served locally right at the edge instead of going all the way through the mobile core and the Internet to reach the origin servers. Therefore, the new edge caching paradigm can significantly reduce duplicate data transmission, alleviate the backhaul capacity requirement, mitigate backbone network congestion, increase network throughput, and improve user experience [ETS19, BBD14, 20117, YHA19].

## 1.1 Motivation

Figure 1a presents the existing caching system where caches are located in public clouds, Content Delivery Networks (CDN), and/or mobile core data centers (DCs). When a user requests a content item (e.g., a YouTube video), her request will first go to the core DC of a telecom operator (Telco), which is usually hundreds of kilometers away from the user. If the content is available in the core, the request can be processed directly there. Otherwise, a cache miss occurs, and then the request can be served from a CDN such as the Google Global CDN, and Limelight. If the content is not stored in the CDNs, the request will be routed to the origin servers of the Over-the-Top (OTT) content provider (e.g., YouTube servers, Netflix servers). Currently, cooperative caching among Telcos is not very beneficial because caches are typically placed very far from mobile subscribers. Furthermore, core DCs are normally very large and can store a huge amount of content.

To create new revenue streams and reduce network operation costs, some network operators are also deploying their own mobile CDN within the IP backbone inside their network. Indeed, edge caches can penetrate even deeper into the telecom networks in proximity of the end-users as shown in Figure 1b. However, different from the core with massive capacity, edge nodes and edge caches are resource-constrained. Furthermore, in practice, users of different Telcos (e.g., AT&T, Verizon), especially users in the same geographical area, usually have very similar content access patterns and access many common files. Thus, instead of independent deployment and management of separate edge caching systems, it can be more beneficial if the Telcos cooperatively store highly-accessed common files in a shared cache, which will drastically reduce content duplication and alleviate the edge cache deployment costs (investment in caching facilities) for the operators.

Additionally, the emerging edge computing paradigm further creates strong incentives for operators to cooperate and pool some of their compute and storage resources at the edge. They can significantly benefit from caching common content items in a shared caching facility. With edge caching, the advantages brought by cooperation becomes clear. Each operator can maintain a private cache and share a shared cache with others. It is worth noting that, besides media content such as text and video, edge caches can host various different types of files such as an iOS/Android mobile app update, mobile apps (e.g., Google Maps, Zoom), software programs (e.g., Overleaf, Jupyter), virtual desktop, and databases of numerous applications. Indeed, many of these files are commonly used by users of different operators, which reinforces the motivation of the multi-party edge caching problem.

Despite the benefits of edge caching have been studied extensively in the previous literature along with many real-world deployments [BBD14, 20117, YHA19], most of the existing works on cooperative edge caching consider cooperation among edge caches owned by a *single* operator only [BBD14, YHA19, PIT$^+$18, ZLH$^+$18]. The potential of cache cooperation among multiple operators has been largely ignored in the literature.

Furthermore, the data privacy of individuals Telcos is also important. For example, if TelcoA knows the access pattern of subscribers of TelcoB, TelcoA can learn characteristics of TelcoB's subscribers and design incentive schemes and services to attract these subscribers to switch to TelcoA (e.g., discount plans for accessing certain content/services). An adversary can also attack (e.g., DDoS) different sites hosting content accessed by TelcoB to degrade the service quality and reputation of TelcoB. Additionally, TelcoA can play strategically by declaring different values for their content items to reduce their caching cost, while potentially increasing TelcoB's cost. A privacy-preserving sharing and cooperation mechanism is a compelling option and will encourage Telcos to participate in the proposed scheme without hesitation. Therefore, it is imperative to study various mechanisms that provide the benefits of cache sharing without compromising privacy.

## 1.2 Contribution

In this paper, we aim to tackle the cooperative content caching problem at the network edge where multiple parties (i.e., network operators) can jointly cache common data items in a shared cache. The problem is to identify the set of common items with the highest access frequency to be cached in the shared cache while respecting the privacy of each individual party.

To this end, we introduce a novel model, MPCCache, for privacy-preserving cooperative cache sharing among multiple network operators at the network edge. Indeed, we are among the first to realize and formally examine the multi-party cooperative caching problem by exploiting the non-rivalry of cached data items. Furthermore, to the best of our knowledge, we are the first to tackle this problem through the lens of secure multi-party computation. In terms of technical contribution, our work presents the efficient construction that outputs only the result of a specific function computed securely on the intersection set, (i.e., find $k$ best items in the intersection set) *without revealing the intersection itself to any party*, and works for the multi-party setting with more than two parties.

We demonstrate the practicality of our proposed protocol with experimental numbers. For instance, for the setting of 8 parties each with a data-set of $2^{16}$ records, our full decentralized protocol requires 5 minutes to compute $k$-priority common items for $k = 2^8$. We also propose an optimized server-aid MPCCache construction, which is scalable in the large dataset and number of parties. With 16 parties, each has $2^{20}$ records, our optimized scheme takes only 8 minutes to compute the $k$-priority common items for $k = 2^8$. MPCCache aims at proactive caching where caches are refreshed periodically (e.g., daily) and do not need to be real-time. Therefore, the running time of MPCCache is practical in our cooperative cache sharing.

PARAMETERS: $n$ parties $P_1, \ldots, P_n$, each has a set of $m_i$ items, a threshold $k$, where $k$ is much smaller than the intersection size.

FUNCTIONALITY:

- Wait for an input $S_i = \{(x^i_1, v^i_1), \ldots, (x^i_{m_i}, v^i_{m_i})\}$ from the party $P_i$
- Let $I = \bigcap_{i \in [n]} \{x^i_1, \ldots, x^i_{m_i}\}$ to be the intersection set. For each $x^\star \in I$, compute the sum $v^\star$ of associated values, i.e., $v^\star = \sum_{i=1}^{n} v^i_{j_i}$ where $(x^\star, v^i_{j_i}) \in S_i$
- Give parties $\{x^\star_1, \ldots, x^\star_k\}$ where $v^\star_1, \ldots, v^\star_k$ are $k$ largest numbers among $v^\star_1, \ldots, v^\star_{|I|}$.

Figure 2: The MPCCache functionality

In addition to MPCCache as our main motivating application, we believe that the proposed techniques can find applications in other areas as well.

## 2 Related Work and Technical Overview of MPCCache

Consider a single party with a set of items $S$. Each item includes an identity $x$ (i.e., a file name, a content ID) and its associated value $v$. For each set $S$, an element $(x, v)$ is said to belong to a set of *k-priority* elements of $S$ if its associated value $v$ is one of $k$-largest values in $S$. Note that the value of each content item may represent the number of predicted access frequency of the content or the benefit (valuation) of the operator for the cached content. Because each cache hit can reduce the content access latency, enhance the quality of service for the user, and reduce the bandwidth consumption, each network operator has its own criteria to define the value for each content that can be stored in the shared edge cache space. How to define the value for each content is beyond the subject of this work.

Since the cache is shared among the operators, they would like to store only common content items in the cache. Here, a common item refers to an item (based on identity) that is owned by every party. The common items with the highest values will be placed in the shared cache, and can be accessed by all the parties. The value of a common item is defined as the sum of the individual values of the operators for the item. We consider the cooperative caching problem in the multi-party setting where each party has a set $S_i = \{(x^i_1, v^i_1), \ldots, (x^i_{m_i}, v^i_{m_i})\}$. For simplicity, we assume that parties have the same set size of $m$. An item $(x^\star, v^\star)$ is defined to belong to the set of the k-*priority common* elements if it satisfies the two following conditions: (1) $x^\star$ is the *common* identity of all parties; (2) $(x^\star, v^\star)$ are the *k-priority* elements of $S^\star = \{(x^\star_1, v^\star_1), \ldots, (x^\star_{|I|}, v^\star_{|I|})\}$, where $v^\star_i$ is the sum of the integer values associated with these common identities from each party, and $I = \bigcap_{i \in [n]} \{x^i_1, \ldots, x^i_{m_i}\}$ is the intersection set with its size $|I|$. In the setting, we consider the input datasets of each party contain proprietary information, thus none of the parties are willing to share its data with the other. We describe the ideal functionality of the cooperative cache sharing MPCCache in Figure 2. For simplicity, we remove under-script of the common item $x^\star$ and clarify that a pair $(x^\star, v^i_{j_i}) \in S_i$ belongs to the party $P_i$.

A closely related work to MPCCache is a private set intersection (PSI). Recall that the functionality of PSI enables $n$ parties with respective input sets $X_{i \in [n]}$ to compute the intersection itself $\bigcap_{i \in [n]} X_i$ without revealing any information about the items which are not in the intersection. However, MPCCache requires to evaluate a top-K computation on the top of the intersection $\bigcap_{i \in [n]} X_i$ while also keeping the intersection secret from parties. A naive solution is to use generic secure multi-party computation (MPC) protocols by expressing the MPCCache functionality as a circuit. It has high computational and communication complexity, which is impractical for real-world applications. The work of Pinkas *et al.* [PSWW18, PSTY19] addressed the difficulty in designing a circuit for computing on the intersection by deciding which items of the parties need to be compared. Very recently, the work [RS21, CGS21] present new constructions of circuit-PSI based on Vector-OLE or Relaxed Batch OPPRF. However, their constructions only work for the two-party setting. The protocol of [IKN$^+$19] is based on Diffie-Hellman and Homomorphic encryption which is preferable in many real-world applications due to their low communication complexity. But, their scheme only supports two-party intersection sum computation.

Beyond a rich literature on two-party PSI [PSZ14, OOS17, KKRT16, CLR17, PSZ18, PRTY19, GS19, KRS$^+$19, LRG19, GN19, PRTY20, CM20], there are several works on multi-party PSI. However, most of the existing multi-party PSI constructions [SS08, CJS12, HV17, KMP$^+$17] output the intersection itself. Only very few works [JKU11, NMW13] studied some specific functions on the intersection. While [JKU11] does not deal with the intersection set of all parties (in particular, an item in the output set in [JKU11] *is not necessarily a common item of all parties*), [NMW13] finds common items with the highest preference (rank) among all parties. [NMW13] can be extended to support MPCCache which is a general case of the rank computation. However, the extended protocol is very expensive since if an item has an associated value $v$, [NMW13] represents the item by replicating it $v$ times. For ranking, their solution is reasonable with small $v$. However, $v$ can be a very large value in MPCCache. We describe a detailed discussion in Appendix A. Recently, the work of [RJHK19] propose MPCircuits, a customized MPC circuit with state-of-the-art optimizations. One can extend MPCircuits to identify the secret share of the intersection and use generic MPC protocols to compute a top-k function on the secret-shared intersection set. However, the number of secure comparisons inside MPCircuits is large and depends on the number of parties. We explicitly compare our proposed MPCCache with the MPCircuits-based solution in Section 7.4. The concurrent work of [ENOPC21, CDG$^+$21] consider circuit-PSI in the multi-party setting. [ENOPC21] relies on garbled Bloom filters, which requires at least $\lambda\times$ bandwidth cost more than our construction. The protocol of [ENOPC21] is similar to MPCCache: the first phase outputs zero/random shares; the second phase invokes generic secure computation. However, the first phase of [ENOPC21] requires expensive steps to compute the shares of intersection (Step 6 &7, Figure 6). In addition, [ENOPC21, CDG$^+$21] are not customized for top-K.

To the best of our knowledge, this is the *first* work formally studying top-K computation on a private set intersection in the multi-party setting. Our decentralized MPCCache construction contains two main phases. The first one is to obliviously identify the common items (i.e., items in the intersection set) and aggregate their associated values of the common items in the multi-party setting. In particular, if all parties have the same $x^\star$ in their set, they obtain secret shares of the sum of the associated values $v^\star = \sum_{i=1}^{n} v^i_{j_i}$ where $(x^\star, v^i_{j_i}) \in S_i$. Otherwise, $v^\star$ equals to zero and it should not be counted as a $k$-priority element. A more detailed overview of the approach is presented in Section 5. It is worth noting that the first phase does not leak the intersection set to any party.

The second phase takes these secret shares which are either the zero value or the correct sum of the associated values of common items, and outputs $k$-priority elements. To privately compute the $k$-priority elements of a dataset that is secret shared by $n$ parties, one could study the top-k algorithms.

In MPC setting, a popular method for securely finding the top-k elements is to use an oblivious sort algorithm so that parties jointly sort the dataset in decreasing order of the associated values, and pick the $k$ largest values. There exists several oblivious sorting protocols [AKS83, Goo10, Shi19] with the complexity of $O(m\log(m))$, however, they are impractical due to a very high constant behind the big-O (see Appendix A.3 for more detail). The more practical sorting algorithm is Batcher's network [Bat68] whose computational complexity overhead is $O(m\log^2(m))$ of secure comparisons, where $m$ is the size of datasets while the communication complexity is $O(\ell m\log^2(m))$, where $\ell$ is the bit-length of the element. To output the index of the biggest values, we also need to keep track of their indexes, therefore, the communication complexity is $O((\ell+\log(m))m\log^2(m))$. Another approach to compute $k$-priority elements is to use an oblivious heap that allows to get a maximum element from the heap (ExtractMax). This solution requires to call ExtractMax $k$ times, which leads to a number of rounds of the interaction of at least $O(k\log(m))$.

In our MPCCache problem, the size of an edge cache is usually much smaller than the size of the content library that each party manages. For instance, mobile users of a Telco can access various types of contents and applications on the Internet. Obviously, an edge cache cannot store all content (e.g., videos, photos, news, webpages) on the Internet, thus $k << m$. Also, note that the size of the edge cache is usually much smaller than the caching facility at the core of the network operator. Since we are motivated by applications where $m$ is much larger than $k$, we propose a new protocol with computational and communication overhead of $O(m\log^2(k))$ of secure comparisons and $O((\ell+\log(m))m\log^2(k))$ bits, respectively. The proposed protocol requires $O(\log(m))$ rounds.

Very recently, Chen *et al.* [CCD+20] presents a new secure *approximate* top-K selection with complexity of $O(m+k^2)$ comparisons and $O((\ell+\log(m))(m+k^2)$ bits. One could integrate their algorithm in the second phase of our scheme to achieve a better performance. In some applications where *exact* top-K selection required, our $k$-priority is preferable.

Our decentralized protocol supports the full corrupted majority, which means that if any subset of parties is corrupted, they learn nothing except the protocol output (i.e., the indices of the $k$ common items with largest sum associated values). In this paper, we also present the optimization for MPCCache in the non-colluding semi-honest setting in which we assume to know two non-colluding parties. This model can be considered as the server-aided model where clients obliviously distribute (secret share) their private database to non-colluding distrusted servers. Our optimized server-aided MPCCache construction achieves almost the same cost of our two-party decentralized protocol.

# 3 Cryptographic Preliminaries

In this work, the computational and statistical security parameters are denoted by $\kappa, \lambda$, respectively. We use [.] notation to refer to a set. For example, $[m]$ implies the set $\{1,\dots,m\}$. Additionally, we use $[i,j]$ to denote the set $\{i,\dots,j\}$. The additive secret sharing of a value $x$ is defined as $x$.

## 3.1 Secret Sharing

To additively secret share $x$ an $\ell$-bit value $x$ of the party $P_i$ to other parties, he first chooses $x^i \leftarrow Z_{2^\ell}$ uniformly at random such that $x = \sum_{j=1}^{n} x^j \mod 2^\ell$, and then sends each $x^j$ to the party $P_j$. For ease of composition we omit the mod. To reconstruct a additive shared value $x$, all parties $P_j$ sends $x = x^j$ to the party $P_i$, who locally reconstructs the secret value by computing $x \leftarrow \sum_{i=1}^{n} x^j$. Given two shared values $x$ and $y$, it is easy to non-interactively add the shares by having parties to compute $x+y = x+y$.

In this work, we also use Boolean sharing in the binary field. Boolean sharing can be seen as additive sharing in the field $Z_2$. The XOR operation is replaced by the addition operation.

### 3.2 Programmable OPRF

Oblivious PRF [FIPR05] is a 2-party protocol in which the sender learns a PRF key $k$ and the receiver, with a private input $x$, learns $F(k, x)$, where $F$ is a PRF.

A programmable OPRF (OPPRF) functionality [PSTY19, KMP$^+$17] is similar to OPRF, except that it allows the sender to program the output of the PRF on his/her set of points. In OPPRF, a sender with a set of points $S = \{(x_1, v_1), \ldots, (x_m, v_m)\}$ interacts with a receiver who has an input $q$. An OPPRF protocol allows the receiver to learn the associated value $v_i$ if $q = x_i$, random otherwise.

### 3.3 Garbled Circuit

Garbled Circuit (GC) [Yao86, GMW87, BMR90] is currently the most common generic technique for practical secure computation. An ideal functionality GC is to take the inputs $x_i$ from every party $P_i$, respectively and computes any function $f$ on them without revealing the secret parties' inputs. In our protocol, we use $f$ as "less than" and "equality" where inputs are secretly shared amongst many parties. For example, a "less than" circuit computation takes the parties' secret shares $x$ and $y$ as input, and output the shares of 1 if $x < y$ and 0 otherwise. To evaluate a function $f$ on shared values, GC first reconstructs the shares, performs $f$ on the top of obtained values, and then secret shares the result $f(x, y)$ to parties. We denote this garbled circuit by $z \leftarrow \mathcal{GC}(x, y, f)$. More detail about GC is described in Appendix D.1.

### 3.4 Oblivious Sort and Merge

The main building block of the sorting algorithm is Compare-Swap operation that takes the secret shares of two values $x$ and $y$, then compares and swaps them if they are out of order. It is typical to measure the complexity of oblivious sort/merge algorithms based on the number of secure Compare-Swap operations.

**Oblivious Sort.** We denote the oblivious sorting by $\{x_{i i \in [m]}\} \leftarrow \mathcal{F}_{\textsf{obv-sort}}(\{x_{i i \in [m]}\}$ which takes the secret share of $m$ values and returns the refresh shares $\{x_1, \ldots, x_m\}$, where $\{x_1, \ldots, x_m\}$ are sorted in decreasing order. As discussed in Appendix A.3, Batcher's network for oblivious sort requires $\frac{1}{4} m \log^2(m)$ Compare-Swap operations.

**Oblivious Merge.** Given two sorted sequences, each of size $m$, we also need to merge them into a sorted array, which is part of the Batcher's oblivious merge sort. It is possible to divide the input sequences into their odd and even parts, and then combine them into an interleaved sequence. This oblivious merge requires $\frac{1}{2} m \log(m)$ Compare-Swap operations and has a depth of $\log(m)$. We denote the oblivious merge by $\{z_1, \ldots, z_{2m}\} \leftarrow \mathcal{F}_{\textsf{obv-merge}}(\{x_1, \ldots, x_m\}, \{y_1, \ldots, y_m\})$.

# 4 Threat MPCCache Model

### 4.1 Security Definitions

We consider a set of parties who want to achieve secure cooperative cache sharing function $f$ at the edge. At the end of the protocol, they learn the output of a function $f$ on the parties' inputs, and nothing else. In the real-world execution, the parties often execute the protocol in the presence of an adversary $\mathcal{A}$ who corrupts a subset of the parties. In the ideal execution, the parties interact with a trusted party that evaluates the function $f$ in the presence of a simulator $\mathsf{Sim}$ that corrupts the same subset of parties. There are two classical security models.

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

*Non-colluding model:* This is modeled by considering independent adversaries, each captures the view of each independent dishonest party. The protocol is secure if the individual distribution of each view can be simulated.

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

In this work, we propose two semi-honest MPCCache constructions. One is secure against any number of corrupt colluding parties. Another is a weaker model where we assume to know at least two parties that do not collude. Extensions to the malicious model will be the subject of our future work. In Appendix B, we formally present the security definition considered in our MPCCache framework, which follows the definition of [KMR11, Ode09] in the multi-party setting.

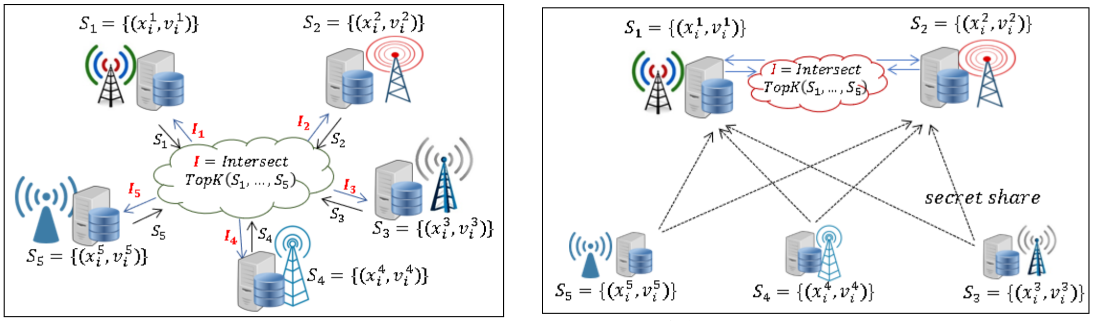
## 4.2 Threat MPCCache Model

We consider a problem of cooperative edge caching among different operators. With cooperative edge caching, each Telco can now reduce its investment on edge caching system while can cache more content at the edge. As a result, this will drastically decrease costs while significantly improving user experience. Due to the privacy concern, it is important to decide which files will be stored in the shared cache without revealing the private data of each operator.

A direct solution to this problem is to utilize a trusted cloud service provider (e.g., AWS) to whom the parties could send their data. The trusted third-party would then determine the set of common items (i.e., common identities) that all parties have. Along with this, for each common item, the third-party also computes the sum of the integer values (from all the parties) associated with the item, and then chooses the *k-priority* elements from the intersection set. Finally, this trusted third-party sends the final results as *k-priority common* elements to the parties to inform that these $k$ items will be stored and available in the shared cache. Unfortunately, there may not exist such a trusted party in real-life scenarios.

With the emergence of edge computing and 5G technology, each Telco can place their database and computing servers at different nodes (e.g., edge clouds, telecom central offices) in the network edge. Thus, for our multi-party cooperative edge caching model, the parties are typically close to each other (i.e., serving the same area like New York City) and the network latency is not a problem in our design. MPCCache can be executed in a fast network with higher latency. We build MPCCache system based on MPC from symmetric-key techniques, which is fast in terms of computation. The MPC allows each distrustful Telco to jointly determine set of common dataitems with highest access frequency to be stored in their capacity-limited shared cache while guaranteeing the privacy of their individual datasets.

The system overview of our MPCCache is described in Figure 3. We assume there is an authenticated secure channel between each pair of participants (e.g., with TLS). Each Telco interacts each other via secure computation with their private input set of items $S_i = \{(x_1^i, v_1^i), \dots, (x_m^i, v_m^i)\}$ and learns a set of $k$-priority common items. Note that each item consists of an identity $x_j^i$ and its associated value $v_j^i$. The operators can value the same item differently due to different benefits



$S_1 = \{(x_i^1, v_i^1)\}$
$S_2 = \{(x_i^2, v_i^2)\}$
$S_3 = \{(x_i^3, v_i^3)\}$
$S_4 = \{(x_i^4, v_i^4)\}$
$S_5 = \{(x_i^5, v_i^5)\}$
$I = Intersect$
$TopK(S_1, \dots, S_5)$
$I_1$
$I_2$
$I_3$
$I_4$
$I_5$
$S_1$
$S_2$
$S_3$
$S_4$
$S_5$
$S_1 = \{(x_i^1, v_i^1)\}$
$S_2 = \{(x_i^2, v_i^2)\}$
$I = Intersect$
$TopK(S_1, \dots, S_5)$
secret share
$S_5 = \{(x_i^5, v_i^5)\}$
$S_4 = \{(x_i^4, v_i^4)\}$
$S_3 = \{(x_i^3, v_i^3)\}$

(a) The decentralized model

(b) The server-aided model

Figure 3: The system overview of our MPCCache

they can obtain from the item being cached (e.g., different predicted access frequency, a different type of content has a different value to their subscribers). How to define the value for each content is beyond the subject of this work.

For multi-party cooperative edge caching, we are interested in the setting where parties serve customers in the same edge location (e.g., a metropolitan area). Thus, the datasets of the operators can be quite similar with many overlapping content items (e.g., local news), and intersection size can be much larger than cache size. Revealing only $k$ most-valued items is much more desirable for the operators compared to revealing the whole intersection set. This indeed motivates us to study the new $k$-priority common items problem.

We consider a semi-honest setting for MPCCache. In the cooperative cache sharing scenario where the computation is between large established organizations (e.g. AT&T Verizon, Sprint). A large company has a reputation to maintain and policies in place to protect against misbehavior, so assuming semi-honest security is more reasonable.

Furthermore, suppose a malicious participant wanted to determine whether certain items were in other participants' caches. An attacker can set a high value for an item $x$ to infer if $x$ were in others' databases. This attack might not be beneficial for the attacker. Specifically, the values can be understood as bids of each party for the associated files, which are proportional to access frequencies of the party to the files. If an attacker sets a fake high value for an item, he needs to pay much more (i.e., expensive to implement the attack). It also reduces the chance for truly popular files of the attacker to be cached. Because access frequencies can be verified, it is easy to determine the attacker, and his reputation can be damaged. In cooperative caching business, reputation is important. Indeed, some economic penalty schemes can be used to enforce truthfulness and prevent an attacker from inflating values associated with his top elements that he wants to be cached. Thus, semi-honest guarantee is appropriate for our application.

We propose two models for our MPCCache with different semi-honest security guarantees. The first one is decentralized which is secure against any number of corrupt, colluding, semi-honest parties. This model requires all participants involved in the end-to-end system execution. Figure 3a presents the system overview of our decentralized MPCCache. Note that a generic cryptographic and secure computation tools may not be suitable and efficient to tackle our MPCCache problem. Thus, it is highly desirable to seek new theoretical and technical refinements to improve their performance, as well as propose concrete optimizations tailored to our specific problem setting. Section 5 presents our customized MPC protocols for this decentralized MPCCache model.

In addition, we show an optimization to our decentralized variant in a weaker security model where a semi-honest adversary who can corrupt at most one of the two specific parties and any

PARAMETERS: Set size $m$, and $n$ parties
FUNCTIONALITY:
- Wait for secret share values $\{v_1, ..., v_m\}$ from the $i^{th}$ party.
- Give all parties $k$ indexes $\{i_1, \ldots, i_k\}$ such that $\{v_{i_1}, \ldots, v_{i_k}\}$ are largest values among $\{v_1, ..., v_m\}$.

Figure 4: The $k$-priority functionality ($\mathcal{F}_{\textsf{k-prior}}$)

subset of other parties. We call the considered model as server-aided MPCCache and present its system overview in Figure 3b. It is worth mentioning that our server-aided security model is weaker than colluding semi-honest setting since we assume to know two specific non-colluding parties which we called as servers. However, our model is stronger than the non-colluding semi-honest model where all parties are assumed to do not collude.

One can view our server-aided protocol as a standard two-server-aided setting where the parties (data owners) outsource the entire computation via XOR-sharing to two untrusted but non-colluding servers from the very beginning. The two-server MPC has been formalized and used in various scheme [DG16, BGI15, MZ17, RS20, CGK20]. However, a key difference from the standard server-aided setting is our proposed secret sharing scheme such that even two servers collude, they only learn the intersection items and nothing else about other uncommon items. In contrast, in the standard server-aided setting, the colluding servers learn the entire dataset of the participants.

The server-aided MPCCache is also realistic in the considered cache sharing scenario. For example, in the US, many small companies are built on a large established company (e.g., virtual mobile network operators operate on the network infrastructure of a big telco like Verizon or AT&T). It is reasonable to assume two big companies do not collude (e.g., by law or due to economic/competition reasons). The small companies and large companies can be considered as users and servers, respectively. With the established reputation of large companies, assuming non-colluding security is practical.

## 5 Our Decentralized MPCCache

We now present our main technical result, an application of MPC tailored to our MPCCache problem. We denote the $n$ parties by $P_1, \ldots, P_n$. Let $S_i$ be the input set of party $P_i$. The goal is to securely compute the $k$-priority common items. For sake of simplicity, we assume each set has $m$ items and write $S_i = \{(x^i_1, v^i_1), \ldots, (x^i_m, v^i_m)\}$. We use subscript $j$ to refer to a particular item $(x^i_j, v^i_j)$.

The construction closely follows the high-level overview presented in Section 2. Recall that our MPCCache construction contains two main parts. The first phase allows the parties to collectively and securely generate shares of the sum of the associated values under the so-called conditional secret sharing condition. If all parties have $x$ in their sets then the sum of their obtained shares is equal to the sum of the associated values for this common identity $x$ (i.e., common items). Otherwise, the sum of the shares is zero. These shares are forwarded as input to the second phase, which ignores the zero sum and returns only $k$-priority common items. For the second phase, we first present the $\mathcal{F}_{\textsf{k-prior}}$ functionality of computing *k-priority* elements in Figure 4. We will explain our protocol for $k$-priority in Section 5.3 and then use $\mathcal{F}_{\textsf{k-prior}}$ as a black box to describe our MPCCache construction.

### 5.1 A special case of our first phase

We start with a special case. Suppose that each party $P_{i\in[n]}$ has only one item $(x^i, v^i)$ in its set $S_i$. Our first phase must satisfy the following conditions:

(1) If all $x^i$ are equal, the parties obtain secret shares of the sum of the associated values $v^\star = \sum_{i=1}^{n} v^i$.

(2) Otherwise, the parties obtain secret shares of zero.

(3) The protocol is secure in the semi-honest model, against any number of corrupt, colluding, semi-honest parties.

The requirement (3) implies that all corrupt parties should learn nothing about the input of honest parties. To satisfying (3), the protocol must ensure that parties do not learn which of the cases (1) or (2) occurs.

We assume that there is a leader party (say $P_1$) who interacts with other parties to output (1). The protocol works as follows. For $(x^i, v^i)$, the party $P_{i\neq 1}$ chooses a secret $s^i$ uniformly at random, defines $w^i \overset{\mathsf{def}}{=} v^i - s^i$, and computes an encryption of $w^i$ under the key $x^i$ as $\mathsf{Enc}(x^i, w^i)$. The party $P_{i\neq 1}$ then sends the ciphertext to the leader party $P_1$. Using his item $x^1$ as a decryption key, the party $P_1$ decrypts the received ciphertext and obtains $w^i$ if $x^1 = x^i$, random otherwise. Clearly, if all parties have the same $x^1$, $P_1$ receives the correct plaintext $w^i = v^i - s^i$ from $P_{i\neq 1}$. Now, $P_1$ computes $s^1 \overset{\mathsf{def}}{=} v^1 + \sum_{i=2}^{n} w^i$. It easy to verify that $\sum_{i=1}^{n} s^i = (v^1 + \sum_{i=2}^{n} w^i) + \sum_{i=2}^{n} s^i = v^1 + \sum_{i=2}^{n}(w^i + s^i) = \sum_{i=1}^{n} v^i = v^\star$. By doing so, each party $P_i$ has an additive secret share $s^i$ of $v^\star$ as required in (1).

In case that not all $x^i$ are equal, the sum of all the shares $\sum_{i=1}^{n} s^i$ is a random value since $P_1$ receives a (random) incorrect plaintext $w^i$ from some party. To compute (2) which output the shares of zero, we use the garbled circuit to turn the random incorrect sum to zero. However, for (3), the random sum and the correct sum are indistinguishable on the view of all parties. One might build a garbled circuit on the top of the sum computation. In particular, the circuit does $n$ equality comparisons to check whether all $x^i$ is equal. If yes, the circuit gives refresh shares of the correct sum, otherwise shares of zero. This solution requires $O(n)$ equality comparisons inside MPC. We aim to minimize the number of equality tests.

We improve the above solution using zero-sharing [AFL+16, KMP+17, MR18]. An advantage of the zero-sharing is that the party can non-interactively generate a Boolean share of zero after a one-time setup. Let's denote the zero share of the party $P_i$ to be $z^i$. We have $\bigoplus_{i=1}^{n} z^i = 0$. Similar to the protocol described above to achieve (1), when the party $P_i$ uses $(x^i, z^i)$ as input, the party receives a Boolean secret share $t^i$. If all $x^i$ are equal, the XOR of all obtained shares is equal to that of all associated values. In other words, $\bigoplus_{i=1}^{n} t^i = \bigoplus_{i=1}^{n} z^i = 0$. Otherwise, $\bigoplus_{i=1}^{n} t^i$ is random. These obtained shares are used as an `if` condition to output either (1) or (2). Concretely, parties jointly execute a garbled circuit to check whether $\bigoplus_{i=1}^{n} t^i = 0$. If yes (*i.e.* parties have the same item), the circuit re-randomizes the shares of $v^\star$, otherwise, generates the shares of zero. We call the final obtained value as a conditional secret share. Since XOR is free in garbled circuit [KS08], the zero-sharing based solution requires only one equality comparison inside MPC.

In the following, we describe a detail construction to generate zero-sharing and how to compute $t^i, w^i, \forall i \in [n]$, more efficiently.

a) Zero-sharing key setup: one key is shared between every pair of parties. For example, the key $k_{ij}$ is for a pair $(P_i, P_j)$ where $i, j \in [n], i < j$. It can be done as the party $P_i$ randomly chooses $k_{i,j} \leftarrow \{0,1\}^\kappa$ and sends it to the party $P_j$. Let's denote a set of the zero-sharing keys as $K_i = \{k_{i,1}, \ldots, k_{i,(i-1)}, k_{i,(i+1)}, \ldots, k_{i,n}\}$.

b) Generating zero share: Given $K_i$ and a PRF $F : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^*$, each party $P_i$ locally computes a zero share as $z^i = \bigoplus_{j=1}^n F(k_{i,j}, 1)$. It is easy to see that each term $F(k_{i,j}, 1)$ appears exactly twice in the expression $\bigoplus_{i=1}^n z^i$. Thus, $\bigoplus_{i=1}^n z^i = 0$. We define $f^{\mathsf{z}}(K_i, b) \overset{\mathsf{def}}{=} \bigoplus_{j=1}^n F(k_{ij}, b)$ for the party $P_i$ to generate the $b$-th instance of zero share.

c) Computing $t^1$ and $w^1$: the party $P_{i\neq 1}$ chooses secret values $s^i$ and $t^i$ uniformly at random. For an input $(x^i, v^i)$ and a zero share $z^i$, he computes $w^i \overset{\mathsf{def}}{=} v^i - s^i$ and $y^i \overset{\mathsf{def}}{=} z^i \oplus t^i$ and sends the ciphertext $\mathsf{Enc}(x^i, y^i || w^i)$ to the leader party $P_1$. Using his item $x^1$ as a decryption key, the party $P_1$ obtains the correct $y^i || w^i$ if $x^1 = x^i$, random otherwise. $P_1$ computes $s^1 \overset{\mathsf{def}}{=} v^1 + \sum_{i=2}^n w^i$ and $t^1 \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^n y^i) \oplus z^1$. At this point, each party has conditional secret shares $s^i$ and $t^i$ such that $\sum_{i=1}^n s^i = v^\star$ and $\bigoplus_{i=1}^n t^i = 0$ if all $x^i$ are equal.

So far, we only consider the simple case where each party has only one item. When each party has $m$ items in his set, a naive solution costs $\mathcal{O}(m^n)$ equality checks, one for each possible combination of the sets.

## 5.2 A general case of our first phase

We show how to efficiently extend our protocol to support the general case where $m > 1$. At the high-level idea, we use hashing scheme to map the common items into the same bin, and then reply on OPPRF to compress each bin into one conditional share so that the parties can evaluate $\mathsf{MPCCache}$ bin-by-bin efficiently. By doing so, we only need to do $\mathcal{O}(m)$ secure comparisons instead of naive $\mathcal{O}(m^n)$ pair-wise comparisons.

Similar to many PSI constructions [PSSZ15, KKRT16], we use two popular hashing schemes: Cuckoo and Simple hashing. The leader party $P_1$ uses Cuckoo hashing [PR04] with $\widetilde{k} = 3$ hash functions to map his $\{x_1^1, \ldots, x_m^1\}$ into $\beta = 1.27m$ bins, each has at most one item. He then pads his bin with dummy items so that each bin contains exactly one item. This step is to hide his actual Cuckoo bin size. On the other hand, each party $P_{i\neq 1}$ use the same set of $\widetilde{k}$ Cuckoo hash functions to place its $\{x_1^i, \ldots, x_m^i\}$ into $\beta$ bins (so-called Simple hashing), each item is placed into $\widetilde{k}$ bins with high probability. The party $P_{i\neq 1}$ also pads his bin with dummy items so that each bin contains exactly $\gamma = 2\log(m)$ items. According to [PSSZ15, DRRT18], the parameters $\beta, \widetilde{k}, \gamma$ are chosen so that with high probability $1 - 2^{-\lambda}$ every Cuckoo bin contains at most one item and no Simple bin contains more than $\gamma$ items. More detail is described in Appendix D.2.

For each bin $b^{th}$, $P_1$ and $P_{i\neq 1}$ can run the protocol for our special case described in Section 5.1. In particular, let $B_i[b]$ denote the set of items in the $b^{th}$ bin of the party $P_i$. The party $P_{i\neq 1}$ chooses secret values $s_b^i$ and $t_b^i$ uniformly at random, and locally generate a zero share $z_b^i \leftarrow f^{\mathsf{z}}(K_i, b)$. Note that $\bigoplus_{i=1}^n z_b^i = 0$. For each $(x_j^i, v_j^i) \in B_i[b]$, $P_{i\neq 1}$ computes $w_j^i \overset{\mathsf{def}}{=} v_j^i - s_b^i$ and $y_j^i \overset{\mathsf{def}}{=} z_b^i \oplus t_b^i$ and sends the ciphertext $\mathsf{Enc}(x_j^i, y_j^i || w_j^i)$ to the leader party $P_1$. Using his item $x_b^1 \in B_1[b]$ as a decryption key, the party $P_1$ obtains $\hat{y}_j^i || \hat{w}_j^i$ which is a correct plaintext $y_j^i || w_j^i$ if $x_b^1 = x_j^i$, random otherwise. Since there are $\gamma$ pairs $\{\hat{y}_j^i, \hat{w}_j^i\}$ from $P_{i\neq 1}$, the leader party $P_1$ has $\gamma^{n-1}$ possible ways to choose $j_i \in [\gamma]$ and compute his conditional secret shares $s_b^1 \overset{\mathsf{def}}{=} \sum_{i=2}^n \hat{w}_{j_i}^i$ and $t_b^1 \overset{\mathsf{def}}{=} \bigoplus_{i=2}^n \hat{y}_{j_i}^i$ as before. Thus, this solution requires $\gamma^{n-1}$ equality comparisons to check all combinations whether $\bigoplus_{i=1}^n t_b^i = 0$ to determinate whether $x_b^1$ is common.

To improve the above computation, we use a technique of OPPRF to compress all $\mathsf{Enc}(x_j^i, y_j^i || w_j^i)$ of the bin into a package so that after the decryption the leader party learns from $P_{i\neq 1}$ only one pair $\{\hat{y}^i, \hat{w}^i\}$ per bin, instead of $\gamma$ pairs per bin. Given a set of points $\mathcal{T} = \{(x_1, e_1), \ldots, (x_\gamma, e_\gamma)\}$, a general key idea of OPPRF construction is to pack the set $\mathcal{T}$ into a polynomial $P$ of degree $\gamma - 1$ such that $P(x_j) = e_j, \forall j \in [\gamma]$. Therefore, if $x^\star = x_j$, $P(x^\star) = e_j$. The OPPRF construction contains another PRF step to make sure that $P(x^\star)$ is random if $x^\star$ is not equal to any $x_j$. We refer reader to [KMP+17, PSTY19] for more detail.

By integrating OPPRF as a building block into our protocol, the parties invoke its instance bin-by-bin as follows. For each bin $b$, the party $P_{i\neq 1}$ creates a set of points $\mathcal{T}_b^i = \{(x_j^i, \mathsf{Enc}(x_j^i, y_j^i||w_j^i))\}$. The leader party $P_1$ acts as OPPRF's receiver with input $x_b^1 \in B_1[b]$ while $P_{i\neq 1}$ acts as OPPRF's sender with the set of points $\mathcal{T}_b^i$. If both $P_1$ and $P_{i\neq 1}$ have the same item identity $x_b^1 = x_{j_i}^i$, $P_1$ receives the correct output $\mathsf{Enc}(x_j^i, y_{j_i}^i||w_{j_i}^i))$ and thus obtain a correct plaintext pair $\{\hat{y}_b^i, \hat{w}_b^i\} = \{y_{j_i}^i, w_{j_i}^i\}$ after the decryption. If $x_b^1$ is not in the bin $B_i[b]$ of the party $P_{i\neq 1}$, the leader party receives a random pair.

In summary, if all parties have the same item identity $x_1^b$ in the $b^{th}$ bin, $P_1$ receives the correct plaintext $\hat{w}_b^i = v_{j_i}^i - s_b^i$ and $\hat{y}_b^i = z_b^i \oplus t_b^i$ from the corresponding OPPRF execution involving $P_{i\neq 1}$. The leader sums up all the obtained values $\hat{w}_b^i$ with the associated value $v_b^1$ of the identity $x_b^1 \in B_1[b]$. *i.e.* he computes $s_1^b \overset{\mathsf{def}}{=} v_b^1 + \sum_{i=2}^n \hat{w}_b^i$. We have $\sum_{i=1}^n s_b^i = v_b^1 + \sum_{i=2}^n v_{j_i}^i$ if all parties has the common identity $x_b^1 \in B_1[b]$. In other words, $\sum_{i=1}^n s_b^i$ is equal to the sum of the associated values corresponding with the common identity $x_b^1$. Similarly, when defining a conditional secret share $t_b^1 \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^n \hat{y}_b^i) \oplus z_b^1$, we have $\bigoplus_{i=1}^n t_b^i = 0$ if all parties have $x_b^1$. Consider a case that some parties $P_{i\neq 1}$ might not hold the item $x_b^1 \in B_1[b]$ that $P_1$ has, the corresponding OPPRF with these parties gives $P_1$ random values $\hat{y}_b^i||\hat{w}_b^i$. Thus $t_b^1 \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^n \hat{y}_b^i) \oplus z_b^1$ is random for some $i$, so is $\bigoplus_{i=1}^n t_b^i$.

Similar to Section 5.1, we use garble circuit to check whether $\bigoplus_{i=1}^n t_b^i = 0$ for the bin $b$, and outputs either refresh shares of $\sum_{i=1}^n s_b^i$ or shares of zero. Since $P_1$ only has one $s_b^1$, the protocol only needs to execute one comparison circuit per bin, thus the number of equality tests needed is linear in the number of the bins.

Even though the party $P_{i\neq 1}$ uses the same offset $s_b^i, t_b^i$ to compute $w_b^i, y_b^i$, respectively, the OPPRF functionality only gives $P_1$ one pair per bin. Therefore, as long as the OPPRF used is secure, so is our first phase of MPCCache construction. We will formalize and prove secure our first phase which is presented, together with a proof of our MPCCache security in Section 5.4.

In our protocol, we choose party $P_1$ as a "leader", which acts as the receiver to interact with other parties so that he can collectively and securely generate conditional secret shares of the sum of the associated values for the common items. It suffices for him to obliviously create the shares of zero for items that some parties may not hold. We note that any party can be a leader. To avoid the computation overhead on the leader party, one can divide $n$ parties into several small groups, each has their local leader, and then these leaders create another group and pick one of them to be the leader. They build a such tree connection until one party be a global leader. For simplicity, we present our MPCCache construction based on the former network connection.

### 5.3 Our second phase: $k$-*priority* construction

We present our *k-priority* construction for the $\mathcal{F}_{\mathsf{k\text{-}prior}}$ functionality in Figure 5. At the beginning of the protocol, each party holds a secret share of $m$ values $V = \{v_1, ..., v_m\}$, the end goal is to output the indexes of $k$ largest values among $V$ without revealing any additional information. In this section, we measure the complexity of our $k$-priority protocol based on the number of secure Compare-Swap operations.

PARAMETERS:

- Number of parties $n$, set size $m$, and a $k$ value
- An ideal oblivious sort $\mathcal{F}_{\textsf{obv-sort}}$ and oblivious merge $\mathcal{F}_{\textsf{obv-merge}}$ primitives described in Section 3.4.
- A truncation function $\mathsf{trunc}$ which returns first $k$ elements in the list as $\{x_1, \ldots, x_k\} \leftarrow \mathsf{trunc}(\{x_1, \ldots, x_{2k}\})$

INPUT OF PARTY $P_i$: secret share values $S_i = \{v_1, ..., v_m\}$

PROTOCOL:

1. Parties divide the input set $S_i$ into $\frac{m}{k}$ groups, each has $k$ items.
2. For each group $i \in [\frac{m}{k}]$ consisted of $\{v_i, ..., v_{i+k-1}\}$ from party $P_j$, they jointly execute an oblivious sort $G[i] \leftarrow \mathcal{F}_{\textsf{obv-sort}}(\{v_i||i, ..., v_{i+k-1}||(i+k-1)\})$, where $G[i] \overset{\mathsf{def}}{=} \{v_{i_1}||i_1, ..., v_{i_k}||i_k\}$
3. Parties recursively invoke oblivious merges as follows. Assuming that $\frac{m}{k} = 2^d$

```
Procedure LevelMerge ($G[0, \ldots, d], d$)
    if $d = 1$ then
        return $\{v_{i_1}||i_1, ..., v_{i_k}||i_k\}$
    else
        $L = \mathsf{LevelMerge}(G[0, \ldots, \frac{d}{2} - 1], \frac{d}{2})$
        $R = \mathsf{LevelMerge}(G[\frac{d}{2}, \ldots, d - 1], \frac{d}{2})$
        $M \leftarrow \mathcal{F}_{\textsf{obv-merge}}(L, R)$
        where $M \overset{\mathsf{def}}{=} \{v_{i_1}||i_1, ..., v_{i_k}||i_{2k}$ $\{v_{i_1}||i_1, ..., v_{i_k}||i_k\} \leftarrow \mathsf{trunc}(M)$
    end
end
```

4. Parties jointly reconstruct the share $\{v_{i_1}||i_1, ..., v_{i_k}||i_k\}$, and output $\{i_1, \ldots, i_k\}$.

Figure 5: Our secure *k-priority* construction

As discussed in Section 2, one could use oblivious sorting to sort the input set and then takes the indexes of $k$ biggest values. This approach requires about $\frac{1}{4}m\log^2(m)$ Compare-Swap operations and the depth of $\log(m)$. In the following, we describe our construction which costs $\big(\frac{1}{4}\log(k) + \frac{1}{2}\big)m\log(k) - \frac{1}{2}k\log(k)$ Compare-Swap with the same depth. The proposed algorithm achieves an approximate $\frac{\log^2(m)}{\big(\log(k)+2\big)\log(k)}\times$ improvement.

The main idea of our construction is that parties divide the input set into $\lceil\frac{m}{k}\rceil$ groups, each has $k$ items except possibly the last group which may have less than $k$ items (without loss of generality, we assume that $m$ is divisible by $k$). Parties then execute an oblivious sorting invocation within each group to sort these values of this group in decreasing order. Unlike the very recent algorithm [CCD+20] for *approximate* top-K selection where it selects the maximum element within each group for a further computation, we select the top-K elements of two neighbor groups. Concretely, oblivious merger is built on the top of each two sorted neighbor groups. We select a set of the top-K elements from each merger and recursively merge two selected sets until reaching the final result.

Sorting each group requires $\frac{1}{4}k\log^2(k)$ Compare-Swap invocations, therefore, for $\frac{m}{k}$ groups the total Compare-Swap operations needed is $\frac{m}{k}\big(\frac{1}{4}k\log^2(k)\big)$. The oblivious odd-even mergers are performed in a binary tree structure. The merger of two sorted neighbor groups, each has $k$ items, is computed at each node of the tree. Unlike the sorting algorithm, we truncate this resulted array, maintain the secret shares of $k$ largest sorted numbers among these two groups, and

Parameters:

- Set size $m$, a bit-length $\theta$, security parameter $\lambda$, and $n$ parties $P_{i\in[n]}$
- A zero-sharing key setup, OPPRF, $\mathsf{GC}$, and $k$-$\mathsf{priority}$ primitives
- An encryption and decryption scheme $\mathsf{Enc}$, $\mathsf{Dec}$.
- A Cuckoo and Simple hashing with 3 hash functions, number of bins $\beta$, and max bin size $\gamma$.

Input of party $P_{i\in[n]}$: A set of key-value pairs $S_i = \{(x_1^i, v_1^i), \ldots, (x_m^i, v_m^i)\} \subset \big(\{0,1\}^*, \{0,1\}^\theta\big)^m$

Protocol:

I. **Pre-processing**.

1. Parties interact each other to set up a set of zero-sharing keys $K_{i\in[n]}$ and generate $\beta$ instances of zero shares as $z_b^i = f_i^{\mathsf{z}}(K_i, b), \forall b \in [\beta]$.
2. A leader party $P_1$ hashes items $\{x_1^1, \ldots, x_m^1\}$ into $\beta$ bins using the Cuckoo hashing scheme. Let $B_1[b]$ denote the item in is $b$th bin (or a dummy item if this bin is empty).
3. Each party $P_{i\in[2,n]}$ hashes items $\{x_1^i, \ldots, x_m^i\}$ into $\beta$ bins using Simple hashing. Let $B_i[b]$ denote the set of items in the $b^{th}$ bin of this party.

II. **Online**.

1. For each bin $b \in [\beta]$:
   a) Each party $P_{i\in[2,n]}$ chooses $t_b^i \leftarrow \{0,1\}^{\lambda+\log(n)}$ and $s_b^i \leftarrow \{0,1\}^\theta$ uniformly at random, and generates a set of points $\mathcal{T}_b^i = \big\{\big(x_j^i, \mathsf{Enc}(x_j^i, y_j^i||w_j^i)\big)\big\}$ where $x_j^i \in B_i[b]$, $y_j^i \overset{\mathsf{def}}{=} z_b^i \oplus t_b^i$, $(x_j^i, v_j^i) \in S_i$ and $w_j^i \overset{\mathsf{def}}{=} v_j^i - s_b^i$. The party then pads $\mathcal{T}_b^i$ with dummy pairs to the max bin size $\gamma$.
   b) For $i \in [2,n]$, the parties $P_i$ and $P_1$ invoke an OPPRF instance where:
      - $P_{i\in[n]}$ acts as a sender with input $\mathcal{T}_b^i$
      - $P_1$ acts as a receiver with $x_b^1 \in B_1[b]$, and obtains $c_b^i$.
   c) For $i \in [2,n]$, $P_1$ computes $\hat{y}_b^i||\hat{w}_b^i \leftarrow \mathsf{Dec}(x_b^1, c_b^i)$
      Note that $\hat{y}_b^i = z_{j_i}^i \oplus t_b^i$ and $\hat{w}_b^i = v_{j_i}^i - s_b^i$ for $x_b^1 = x_{j_2}^2 = \ldots = x_{j_n}^n$. Otherwise, $\hat{y}_b^i, \hat{w}_b^i$ are random.
   d) $P_1$ computes $t_b^1 \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^n \hat{y}_b^i) \oplus z_b^1$ and $s_1^b \overset{\mathsf{def}}{=} v_b^1 + \sum_{i=2}^n \hat{w}_b^i$ where $z_b^1$ and $v_b^1$ are zero share and the associated value corresponding to $x_b^1$, respectively.
   e) Parties jointly compute $\mathsf{GC}$:
      - Input from $P_i$ is $t_b^i$ and $s_b^i$.
      - Output to $P_i$ is an additive share $u_b$ where $u_b = \sum_{i=1}^n s_b^i$ if $\bigoplus_{i=1}^n t_b^i = 0$, otherwise $u_b = 0$.
      Note that if $x_b^1$ is common, $u_b$ is equal to the sum of its associated values of the common item identity $x_b^1$.
2. Parties invoke a $k$-$\mathsf{priority}$ functionality with input $u_b, \forall b \in [\beta]$, and obtain $k$ indexes of the $k$-$\mathsf{priority}$ common identities.

Figure 6: Our decentralized $\mathsf{MPCCache}$ construction.

throw out the rest. By doing so, instead of $2k$, only $k$ items are forwarded to the next odd-even merger. The number of $\mathsf{Compare\text{-}Swap}$ required for each merger does not blow up, and is equal to $\frac{1}{2}k\log(k)$. After $(\frac{m}{k} - 1)$ recursive oblivious merger invocations, parties obtain the secret share of the $k$ largest values among the input set. In summary, our secure $k$-priority construction requires $\big(\frac{1}{4}\log(k) + \frac{1}{2}\big)m\log(k) - \frac{1}{2}k\log(k)$ $\mathsf{Compare\text{-}Swap}$ operations.

The above discussion gives parties the secret shares of $k$ largest values. To output their indexes, before running our protocol we attach the index with its value using the concatenation $||$. Namely, we use $(\ell + \lceil\log(m)\rceil)$-bit string to represent the input. The first $\ell$ bits to store the additive share $v_i$ and the last $\lceil\log(m)\rceil$ bits to represent the index $i$. Therefore, within a group the oblivious sorting takes $\{v_i||i, ..., v_{i+k-1}||(i+k-1)\}$ as input, use the shares $v_j, \forall j \in [i, i+k-1]$ for the secure comparison. The algorithm outputs the secret shares of the indexes, re-randomizes the shares of the values and swaps them if needed. The output of the modified oblivious sorting is $\{v_{i_1}||i_1, ..., v_{i_k}||i_k\}$ where the output values $v_{i_1} \in \{v_i, \ldots, v_{i+k-1}\}$ are sorted. Similarly, we modify the oblivious merger structure to maintain the indexes. At the end of the protocol, parties obtains the secret share of the indexes of $k$ largest values, which allows them jointly reconstruct the secret indexes.

The security proof of our $k$-priority is given in Appendix C, which straightforwardly follows from the security of its building blocks.

### 5.4 Putting All Together: MPCCache system

We formally describe our semi-honest MPCCache construction is in Figure 6. In the pre-processing step, the parties jointly generate a one-time zero-sharing key setup and then hash their items into corresponding bins. The online step starts when the parties interact with each other with their private inputs.

**Correctness.** From the preceding description, the cuckoo-simple hashing scheme maps the same items into the same bin. Thus, for each bin $\#b$, if parties have the same identity $x_b^1 \in B_1[b]$, they obtain the secret share of the sum of all corresponding associated values. Otherwise, they receive the secret share of zero (In practice, the sum of all parties' associated values for items in the intersection is not equal to zero). In our protocol, the equation $\bigoplus_{i=1}^{n} t_b^i = 0$ determines whether the item $x_b^1$ is common. We choose the bit-length of zero share to be $\lambda + \log(n)$ to ensure that the probability of the false positive event for this equation is overwhelming (e.g. $1 - 2^{-\lambda}$).

The second phase of the online step takes the conditional secret shares from parties, and returns the indexes of $k$-priority common elements. Since $k$ must be less than or equal to the intersection size, the obtained results will not contain an index which its value is equal to zero. In other words, the output of our protocol satisfies the MPCCache conditions since the identity is common and the sum of the integer values associated corresponding to this identity is $k$-largest.

The security of our decentralized MPCCache construction is based on the OPPRF primitive and the encryption scheme used in the first phase, and the security of our $k$-priority (proved in Theorem 1) used in the second phase. The formal proof is given in Theorem 2 in Appendix C.

## 6 Our Server-aided MPCCache

As mentioned in Section 4.2, the server-aided model is reasonable for the setting of our cooperative edge cache sharing problem. In this section, we show an optimization to improve the efficiency of MPCCache. Let assume that parties $P_1$ and $P_2$ are two non-colluding servers, and we call other parties as users. The server-aided protocol consists of two phases. In the first one, each user interacts with the servers so that each server holds the same secret value, chosen by all users, for the common identifies that both servers and all users have. The servers also obtain the additive secret share of the sum of all the associated value corresponding to these common items. In a case that an identity $x_j^e$ of the server $P_{e\in\{1,2\}}$ is not common, this server receives random value. This phase can be considered as each user distributes a share of zero and a share of its associated value under a "common" condition (similar to our conditional secret share). Therefore, if even two servers collude they only learn the intersection items and nothing else, which provides a stronger security guarantee than the standard server-aided mentioned in Section 4.2. Our second phase involves only the servers' computation, which can be done by using our 2-party decentralized MPCCache construction described in Section 5.4.

PARAMETERS:

- Set size $m$, a bit-length $\theta$, security parameter $\lambda$, and $n$ parties $P_{i\in[n]}$.
- A OPPRF, $\mathsf{MPCCache}$ primitives

INPUT OF PARTY $P_{i\in[n]}$: A set of key-value pairs $S_i = \{(x_1^i, v_1^i), \dots, (x_m^i, v_m^i)\}$

PROTOCOL:

I. **Centralization**.

1. Each user $P_{i\in[3,n]}$ chooses $z_j^i \leftarrow \{0,1\}^{\lambda+\log(n)}$ and $s_j^i \leftarrow \{0,1\}^{\theta}$ uniformly at random, and generates two sets of points $\mathcal{T}^{e,i} = \{(x_j^i, \mathsf{Enc}(x_j^i, z_j^i||w_j^{e,i}))\}$, where $(x_j^i, v_j^i) \in S_i$, $w_j^{1,i} \overset{\mathsf{def}}{=} s_j^i$ and $w_j^{2,i} \overset{\mathsf{def}}{=} v_j^i - s_j^i$.
2. For $i \in [3,n]$ and $e \in \{1,2\}$, the user $P_i$ and the server $P_e$ invoke an OPPRF instance where:
   - $P_i$ acts as a sender with input $\mathcal{T}^{e,i}$
   - $P_e$ acts as a receiver with $x^e_{j\in[m]}$, and receives the corresponding output $c_j^e$
   - $P_e$ computes $\hat{z}_j^{e,i}||\hat{w}_j^{e,i} \leftarrow \mathsf{Dec}(x_j^e, c_j^e)$.
3. For $j \in [m]$, each server $P_{e\in\{1,2\}}$ computes $y_j^e \overset{\mathsf{def}}{=} \bigoplus_{i=3}^n \hat{z}_j^{e,i}$ and $s_j^e \overset{\mathsf{def}}{=} v_j^e + \sum_{i=3}^n \hat{z}_j^{e,i}$ for each identity $x_j^e$.

II. **Server-working**. Two servers $P_{e\in\{1,2\}}$ invoke an instance of $\mathsf{MPCCache}$ where each party's input is a set of points $\{(y_1^e, s_1^e), \dots, (y_m^e, s_m^e)\}$ and learns $k$-$\mathsf{priority}$ common items.

Figure 7: Our server-aided $\mathsf{MPCCache}$ construction.

More concretely, in the first phase, each user $P_{i\in[3,n]}$ chooses, for each item $x_j^i, \forall j \in [m]$, secret values $s_j^i \leftarrow \{0,1\}^*$ and $z_j^i \leftarrow \{0,1\}^*$ uniformly at random. The user $P_{i\in[3,n]}$ defines $w_j^{1,i} \overset{\mathsf{def}}{=} s_j^i$, and computes $w_j^{2,i} \overset{\mathsf{def}}{=} v_j^i - s_j^i$. Similar to our decentralized $\mathsf{MPCCache}$ construction described in Section 5.2, each user $P_{i\in[3,n]}$ generates two sets of points $\mathcal{T}^{e,i} = \{(x_j^i, \mathsf{Enc}(x_j^i, z_j^i||w_j^{e,i}))\}, \forall e \in \{1,2\}$, and sends each of them to the server $P_{e\in\{1,2\}}$ via OPPRF. Let's $\hat{z}_j^{e,i}||\hat{w}_j^{e,i}$ be an output of the execution with input $x_j^e$ from the server $P_{e\in\{1,2\}}$. If two servers have the same item $x_k^1 = x_{k'}^2$ which is equal to the item $x_j^i$ of the user $P_i$, we have $\hat{z}_k^1 = \hat{z}_{k'}^2 = \hat{z}_j^i$ and $\hat{w}_k^{1,i} + \hat{w}_{k'}^{2,i} = v_j^i$ (since $\hat{w}_k^{1,i} = s_j^i$ and $\hat{w}_{k'}^{2,i} = v_j^i - s_j^i$). Each server $P_{e\in\{1,2\}}$ defines $y_j^e \overset{\mathsf{def}}{=} \bigoplus_{i=3}^n \hat{z}_j^{e,i}$ as an XOR of all the obtained values $\hat{z}_j^{e,i}$ corresponding to each item $x^e_{j\in[m]}$. For two indices $k$ and $k'$, we have $y_k^1 = \bigoplus_{i=3}^n \hat{z}_j^{1,i} = \bigoplus_{i=3}^n \hat{z}_j^{2,i} = y_{k'}^2$ if all parties has $x_k^1 = x_{k'}^2$ in their set. This property allows servers obliviously determinate the common items (i.e, checking whether $y_k^1 = y_{k'}^2, \forall k, k' \in [m]$). Moreover, let $s_j^e \overset{\mathsf{def}}{=} v_j^e + \sum_{i=3}^n \hat{w}_j^{e,i}$. For two indices $k$ and $k'$, $s_k^1$ and $s_{k'}^2$ are secret shares of the sum of the associated values for the common item $x_k^1 = x_{k'}^2$ as $s_k^1 + s_{k'}^2 = (v_k^1 + \sum_{i=3}^n \hat{w}_k^{1,i}) + (v_{k'}^2 + \sum_{i=3}^n \hat{w}_{k'}^{2,i}) = v_k^1 + v_{k'}^2 + \sum_{i=3}^n(\hat{w}_k^{1,i} + \hat{w}_{k'}^{2,i})$. In summary, after this first phase, each server $P_{e\in\{1,2\}}$ has a set of points $\{(y_1^e, s_1^e), \dots, (y_m^e, s_m^e)\}$ where $y_k^1 = y_{k'}^2$ if all parties have the same identity $x_k^1 = x_{k'}^2$, and $s_k^1 + s_{k'}^2$ is equal to the sum of the associated values of the common $x_k^1$.

At this point, the n-party $\mathsf{MPCCache}$ can be considered as a two-party case where each server $P_{e\in\{1,2\}}$ has a set of points $\{(y_1^e, s_1^e), \dots, (y_m^e, s_m^e)\}$ and wants to learn the $k$-$\mathsf{priority}$ common items. We formally describe the optimized $\mathsf{MPCCache}$ protocol is in Figure 7.

**Correctness.** Recall that $y_j^e = \bigoplus_{i=3}^n \hat{z}_j^{e,i}, \forall e \in \{1,2\}, j \in [m]$. Let $i$ be the highest index of a user $P_{i\in[3,n]}$ who did not have the identity $x_k^1$ in their input set. That user does not insert a pair $\{x_k^1, \texttt{something}\}$ to his set $\mathcal{T}^{e,i}$ for the OPPRF in Step (I.1). Thus, the obtained value $\hat{z}_k^1$ is random. The protocol is correct except in the event of a false positive — i.e., $y_k^1 = y_{k'}^2$ for some $x_k^1$

not in the intersection. It is need to ensure that the probability that of a false positive involving $x_k^1$ is $2^{-\lambda}$. By setting $\ell = \lambda + 2\log_2(m)$, a union bound shows that the probability of *any* item being erroneously included in the intersection is $2^{-\lambda}$.

The security proof of our server-aided MPCCache protocol is essentially similar to that of the decentralized protocol, which is presented in Theorem 3 in Appendix C.

**Discussion.** From our two-server-aided architecture Figure 7, our protocol can extend to support a small set of servers (e.g., $t$ servers, $t < n$). More precisely, in the centralization phase, each user $P_{i\in[t+1,n]}$ secret shares their associated value $v^i_{j\in m}$ to each server $P_{e\in[t]}$ via OPPRF. Each server aggregates the share of associated value corresponding to their item. The obtained results are forwarded to the second phase (server-working) in which the servers $P_{e\in[t]}$ jointly run MPCCache to learn $k$-priority common items. The main cost of our server-aided construction is dominated by the second phase. Hence, the performance of $t$-server-aided scheme is similar to that of decentralized MPCCache performed by $t$ parties. We are interested in two-server aided architecture since we can take advantage of recent improvements on two-party secure computation for our top-k algorithm and garbled circuit. Moreover, the two-server setting is common in various cryptography schemes (e.g. private information retrieval [CGK20], distributed point function [GI14], private database query [WYG$^+$17]).

# 7 Implementation

In order to evaluate the performance of our proposed secure edge caching schemes, we implement building blocks used in our MPCCache and do a number of experiments on a single Linux machine which has Intel Core i7 1.88GHz CPU and 16GB of RAM, where each party is implemented as a separate process, and communicate over a simulated 10Gbps network with 0.2ms round-trip time. We observe that run-time on WAN can be computed with the linear cost model as the overall running time is equal to the sum of computation time and data transfer time. In real-world settings, the slow-down would likely be even higher because of network latency. However, the application of our MPCCache usually runs in the fast and low-latency edge network, especially with the upcoming 5G technologies [20117, BBD14, ETS19, YHA19]. Indeed, with the emergence of mobile edge computing, the servers of operators in our application will typically be placed closer to each other (e.g., in edge clouds in the same area such as New York City).

All evaluations were performed with an identity and its associated value input length 128 bits and $\theta = 16$ bits, respectively, a statistical security parameter $\lambda = 40$ and computational security parameter $\kappa = 128$. We use OPPRF code from [KMP$^+$17], oblivious sort and merge from [WMK16]. To understand the scalability of our scheme, we evaluate it on the range of the number parties $n \in \{4, 6, 8, 16\}$. Note that the dataset size $m$ of each party is expected to be not too large (e.g.,billions). First, the potential of MPCCache is in 5G where each shared cache is deployed for a specific region. Second, each operator chooses only frequently-accessed files as an input to MPCCache because the benefit of caching less-accessed files is small. Therefore, we benchmark our MPCCache on the set size $m \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. To understand the performance effect of the $k$ values discussed in Section 5.3, we use $k \in \{2^6, 2^7, 2^8, 2^9, 2^{10}\}$ in our $k$-priority experiments, and compare its performance to the state-of-the-art oblivious sort.

We assume there is an authenticated secure channel between each pair of parties (e.g., with TLS). Our MPCCache is very amenable to parallelization. Specifically, our algorithm can be parallelized at the level of bins. In our evaluation, however, we use a single thread to perform the

Table 1: The total runtime (minute) and communication per item (KB) of our $k$-priority construction and the state-of-the-art oblivious sort, where $m$ is the dataset size.

| $m$ | **Running Time** | | | | | **Communication** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Ours $k$-priority | | | | Sort | Ours $k$-priority | | | | Sort |
| | $k = 2^7$ | $k = 2^8$ | $k = 2^9$ | $k = 2^{10}$ | | $k = 2^7$ | $k = 2^8$ | $k = 2^9$ | $k = 2^{10}$ | |
| $2^{12}$ | 0.012 | 0.014 | 0.016 | 0.018 | 0.014 | 8.008 | 10.11 | 12.38 | 14.72 | 18.43 |
| $2^{14}$ | 0.049 | 0.056 | 0.068 | 0.087 | 0.071 | 8.05 | 10.21 | 12.6 | 15.2 | 25.09 |
| $2^{16}$ | 0.199 | 0.238 | 0.294 | 0.35 | 0.382 | 8.061 | 10.23 | 12.65 | 15.32 | 32.77 |
| $2^{18}$ | 0.786 | 0.996 | 1.217 | 1.449 | 1.964 | 8.063 | 10.24 | 12.67 | 15.35 | 41.47 |
| $2^{20}$ | 2.984 | 3.798 | 4.697 | 5.527 | 9.844 | 8.064 | 10.24 | 12.67 | 15.36 | 51.2 |

Table 2: The total runtime (minute) of our server-aided and decentralized MPCCache to find $k$-priority common items, where the number of parties $n$, each with dataset size $m$.

| **Parameters** | | **Server-aided** | | | | | **Decentralized** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $m$ | $n$ | $k = 2^6$ | $k = 2^7$ | $k = 2^8$ | $k = 2^9$ | $k = 2^{10}$ | $k = 2^6$ | $k = 2^7$ | $k = 2^8$ | $k = 2^9$ | $k = 2^{10}$ |
| $2^{12}$ | 4 | 0.04 | 0.043 | 0.045 | 0.047 | 0.05 | 0.18 | 0.19 | 0.2 | 0.2 | 0.22 |
| | 6 | 0.042 | 0.045 | 0.047 | 0.049 | 0.05 | 0.24 | 0.29 | 0.3 | 0.29 | 0.32 |
| | 8 | 0.043 | 0.047 | 0.049 | 0.052 | 0.05 | 0.33 | 0.41 | 0.38 | 0.41 | 0.43 |
| $2^{16}$ | 4 | 0.502 | 0.543 | 0.582 | 0.637 | 0.69 | 1.94 | 2.36 | 2.39 | 2.77 | 3.15 |
| | 6 | 0.502 | 0.545 | 0.584 | 0.639 | 0.7 | 3.01 | 3.09 | 3.57 | 3.8 | 3.92 |
| | 8 | 0.53 | 0.57 | 0.61 | 0.66 | 0.72 | 4.29 | 4.43 | 4.92 | 5.01 | 5.6 |
| $2^{20}$ | 4 | 7.67 | 7.89 | 7.97 | 8.02 | 8.07 | 28.17 | 29.3 | 29.41 | 32.13 | 35.7 |
| | 6 | 7.7 | 7.92 | 8.01 | 8.1 | 8.17 | 46.82 | 47.16 | 47.48 | 49.16 | 52.98 |
| | 8 | 8.03 | 8.11 | 8.26 | 8.32 | 8.37 | 66.63 | 67.52 | 68.16 | 69.27 | 69.51 |

Table 3: The total runtime (minute) and communication cost per item (KB) of our server-aided MPCCache with $k = 2^8$ for the number of parties $n$, each with set size $m$.

| **#party $n$** | **Role** | **Running Time** | | | | | **Communication** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $m = 2^{12}$ | $m = 2^{14}$ | $m = 2^{16}$ | $m = 2^{18}$ | $m = 2^{20}$ | $m = 2^{12}$ | $m = 2^{14}$ | $2^{16}$ | $m = 2^{18}$ | $m = 2^{20}$ |
| 4 | User | 0.008 | 0.028 | 0.105 | 0.388 | 1.441 | 0.58 | 0.66 | 0.73 | 0.81 | 0.88 |
| | Server | 0.045 | 0.171 | 0.582 | 2.154 | 7.971 | 24.47 | 26.34 | 28.06 | 29.74 | 31.41 |
| 6 | User | 0.01 | 0.032 | 0.107 | 0.395 | 1.467 | 1.17 | 1.32 | 1.46 | 1.61 | 1.76 |
| | Server | 0.047 | 0.175 | 0.584 | 2.153 | 8.009 | 24.77 | 26.67 | 28.43 | 30.14 | 31.85 |
| 8 | User | 0.012 | 0.036 | 0.13 | 0.481 | 1.775 | 1.75 | 1.97 | 2.19 | 2.42 | 2.64 |
| | Server | 0.049 | 0.179 | 0.607 | 2.252 | 8.263 | 25.06 | 27 | 28.79 | 30.54 | 32.28 |
| 16 | User | 0.02 | 0.058 | 0.24 | 0.912 | 3.374 | 4.09 | 4.61 | 5.12 | 5.64 | 6.15 |
| | Server | 0.058 | 0.199 | 0.634 | 2.287 | 8.315 | 26.23 | 28.32 | 30.26 | 32.15 | 34.04 |

computation between two parties. We use a one-time pad to implement the encryption scheme. For example, to compute $\mathsf{Enc}(x^i_j, y^i_j || w^i_j)$ in Step (II.1.a) of Figure 6, we first truncate $x^i_j$ to obtain a string with $\lambda + \theta + \log(n)$ bits long, and then xor the result with $y^i_j || w^i_j$. Together with OPPRF, the bit-length of $y^i_j$ is already chosen to guarantee the correctness and security.

## 7.1 $k$-priority performance

Our $k$-priority requires $(\frac{1}{4}\log(k) + \frac{1}{2})m\log(k) - \frac{1}{2}k\log(k)$ Compare-Swap instances. We use garbled circuit [Yao86, BMR90] to perform secure comparisons.

Table 1 presents the concrete communication cost of our $k$-priority for the different range of top-K. The cost is measured in KB per item as we would like to show an improved performance factor of our proposed protocol compared to the state-of-the-art oblivious sort as well as a performance change when increasing $k$. Recall that the oblivious sort takes about $\frac{1}{4}m\log^2(m)$ invocations of a Compare-Swap computation. Thus, for the dataset size $m = 2^{20}$ and a small $k = 2^7$ our approach shows a 6.3× improvement in terms of the communication cost. As expected, when increasing $k$ to $2^{10}$, the improved factor decreases to 3.3.

We also report the detailed running time of both constructions in Table 1. Our $k$-priority protocol requires only 0.78 minutes for $m = 2^{18}$ elements and $k = 2^7$ while the oblivious sort approach takes 1.96 minutes, a 2.5× improvement. For the same set size and $k = 2^{10}$, our total running time is 1.49 minutes.

To see more clearly the performance change for different $k$ values, Figure 8 in Appendix D presents the total runtime and communication cost per item of our $k$-priority and the oblivious sort in two different y-axes bar chart. All numbers are evaluated on the data set size $m = 2^{16}$. As you can see, there is a minor change in the running time on the $k$-priority performance when increasing $k$.

## 7.2 MPCCache performance

Table 2 presents the total running time for two variants of our MPCCache: decentralized and server-aided models described in Figure 6 and Figure 7, respectively. The main difference between the two constructions is in the steps of equality checks and $k$-priority. While the decentralized protocol requires all participants to jointly compute these steps, in the server-aided framework only two specific servers perform the computation. Therefore, the former model is expensive than the latter one but provides a stronger security guarantee where any subset of corrupted parties learns nothing about the dataset of honest parties. Note that the non-colluding assumption in our server-aided model is realistic for our cooperative edge-sharing application as mentioned in Section 4.2

As can been seen from Table 2, the running time of our server-aided MPCCache shows a minor change when increasing the number of participants. It dues to the fact that the cost of distributing the conditional secret shares with OPPRF is minimal. For MPCCache with $k = 2^8$, $m = 2^{16}$ items, the running time of the server-aided model is increasing from 0.58 minutes to 0.61 minutes when increasing the number of parties from 4 to 8. For the same parameters, the decentralized MPCCache model takes 4.29 minutes for $n = 4$ and 28 minutes for $n = 8$. Note that the running time of our scheme can be directly speedup by using multiple threads.

The communication cost of our decentralized MPCCache is approximate $n$× more than that of the server in the server-aided model presented in Table 3, which leads to a few GBs data transferred for a large dataset. However, in the case for our deployment, the dataset of participants is located in the same cloud region thus the high communication cost is not a bottleneck.

## 7.3 Server-aided MPCCache

Note that the numbers reported in Table 2 are for an end-to-end server-aided MPCCache execution, which includes the user's waiting time for the servers's computation. As discussed Section 6, the server-aided protocol is asymmetric with respect to the servers $P_{e\in\{1,2\}}$ and other parties (user). Table 3 presents the performance of different roles of the participants.

On the server's side, the communication is composed of (a) the centralized step, which has an amortized communication of at most $450 + \gamma|c|$ bits per bin, where $|c| = \lambda + \log(n) + \theta$ is the bit-length ciphertext obtained from the encryption scheme (recall, $\lambda + \log(n)$ and $\theta$ are the bit-length of zero-share and the associated value, respectively); (b) $\beta$ equality checks inside MPC, each requires to communicate $(\lambda + \log(n))$ AND gates, thus, costs $256(\lambda + \log(n))$ bits; (c) the communication of $k$-priority protocol.

Because the user only distributes its dataset to two servers via our conditional secret sharing, his workload is very light. For the small $m = 2^{12}$, only 0.008 minutes running time and 0.58 KB communication per item are required in 4-party MPCCache protocol on the user's side. When increasing $m$ to $2^{20}$, each user executes MPCCache in 1.44 minutes with 0.88 KB of communication. Note that the running time of our protocol on the user's side does not depend much on the number of parties due to the parallelizability with a separate secure channel between user and server.

The server's work is heavy due to equality checks and $k$-priority computation. The server requires 0.58 minutes to compute the $k$-priority common elements from $n = 4$ parties with $m = 2^{16}$, which is about 5.7× more than the user's load. For $m = 2^{20}$ and $n = 16$, the total runtime of the server's side is 8.3 minutes. The numbers show that our protocol also scales to a large number of parties.

### 7.4 Comparison with baseline

By blending standard circuit-based MPC protocols [BMR90, MNPS04] with recent optimized constructions [RJHK19], one can design a new protocol to address the multi-party cooperative cache sharing problem by following similar steps in our proposed protocol. Specifically, this solution can be implemented in two phases. Based on the recent protocol [RJHK19], MPCircuits, the first phase is to compute the secret share of the intersection. The second phase uses generic MPC protocols or our $k$-priority to compute the top-k function on the obtained results. We consider this solution as a *baseline*.

The high-level idea of MPCircuits is to compute the multi-party PSI (or secret shared PSI) in a binary tree structure as they observed that the set intersection of $n$ sets can be expressed as a consecutive set intersection of two sets until reaching the final result. Therefore, the intersection of two sets is computed at each node of the tree, and the final intersection of all sets is computed at the root of the tree. Using three operations as sort, merge, and compare, the complexity of their garbled circuit is $O(nm\ell \log(m)^2)$ where $\ell$ is the bit-length of the element identity. To keep track $\theta$-bit associated value of the identity, the MPCircuits-based solution requires a complexity of $O(nm(\ell + \theta)\log^2(m))$. In contrast, with the lightweight OPPRF, our solution requires only single equality comparison per bin. Thus, the complexity of our circuit is $O(m(|z| + \theta))$, where $z$ is a bit-length of the zero share which is equal to $\min(\ell, \lambda + \log(n))$. It is easy to see that the first phase of our solution is about $n\log^2(m)\times$ better than that of MPCircuit-based approach. For example, with $n = 8$ and $m = 2^{20}$ our solution shows about an $3,200\times$ improvement.

To hide the intersection set size, the output of the MPCircuits-based computation at the root of the tree consists of $mn$ secret shares of all zero strings and the elements in the final intersection. As a result, the second phase of the baseline solution takes $mn$ secret shares as an input of each party. On the other hand, our MPCCache only takes $\beta = 1.27m$ secret shares, each per bin.

## 8 Conclusion and Future Directions

In this work, we design a MPCCache framework for the cooperative content caching at the network edge where multiple network operators can jointly cache common data items in a shared cache. Our proposed solution is built on several state-of-the-art cryptographic primitives such as OPPRF and garbled circuits with various optimizations. In terms of technical contributions, this is the first work formally studying cooperative cache sharing, and top-K computation on the private set intersection in the multi-party setting. Underlying one of our protocols is a new exact top-K selection tailored to secure computation. Our performance results show that our MPCCache scales well to massive datasets and a large number of participants. We highlight some directions for future work:

- Incentive compatibility: In this work, we assume that the parties are truthful by using their true valuations for each content item in their databases. It is because the access frequency of each party to each cached file is measurable and known. Additionally, some economic penalty schemes can be used to enforce truthfulness as mentioned in Section 4.2. How to design efficient incentive-compatible mechanisms (e.g., specific payment rules, reputation update rules) [NRTV07] to encourage parties to act truthfully while considering privacy issues is an exciting research direction.

- Improving scalability: The current implementation of MPCCache only uses single-thread while the scheme can be implemented in a parallel fashion. To enhance scalability, a parallelization at the level of bins can be implemented to allow the scheme to deal with a big dataset (e.g. million points) in seconds. Moreover, our MPCCache construction is from symmetric-key operations, which requires a high communication cost. In some applications run on slow networks, the communication might be a bottleneck. Combination MPC and homomorphic encryption [Gen09] is a potential direction to reduce the network cost.

- Extension to malicious adversaries: In our cooperative cache sharing application, it is sufficient to design a MPCCache secure against semi-honest adversary. However, in other applications, the requirement of security guarantee is stronger. It is necessary to extend our scheme to the malicious adversarial setting. One promising direction is to investigate the SPDZ protocol [DPSZ12].

## References

[20117] Accessed November 2017. *AT&T Edge Cloud (AEC) - White Paper.* 2017. https://about.att.com/ecms/dam/innovationdocs/Edge_Compute_White_Paper%20FINAL2.pdf.

[AFL$^+$16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th ACM STOC*, pages 1–9. ACM Press, April 1983.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BBD14] E. Bastug, M. Bennis, and M. Debbah. Living on the edge: The role of proactive caching in 5g wireless networks. *IEEE Communications Magazine*, 52(8):82–89, Aug 2014.

[BD10] M. Burkhart and X. Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–7, Aug 2010.

[Bf12] Martin Burkhart and Xenofontas Dimitropoulos fontas. Fast private set operations with sepia. 2012.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 257–266. ACM Press, October 2008.

[CCD$^+$20] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M. Sadegh Riazi. Sanns: Scaling up secure approximate k-nearest neighbors search. In *USENIX Security*, August 2020.

[CDG$^+$21] Nishanth Chandran, Nishka Dasgupta, Divya Gupta, Sai Lakshmi Bhavana Obbattu, Sruthi Sekar, and Akash Shah. Efficient linear multiparty psi and extensions to circuit/quorum psi. Cryptology ePrint Archive, Report 2021/172, 2021. https://eprint.iacr.org/2021/172.

[CGK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 44–75, Cham, 2020. Springer International Publishing.

[CGS21] Nishanth Chandran, Divya Gupta, and Akash Shah. Circuit-psi with linear complexity via relaxed batch opprf. Cryptology ePrint Archive, Report 2021/034, 2021. https://eprint.iacr.org/2021/034.

[CJS12] Jung Hee Cheon, Stanislaw Jarecki, and Jae Hong Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Transactions*, 95-A(8):1366–1378, 2012.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1243–1255. ACM Press, October / November 2017.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 34–63. Springer, Heidelberg, August 2020.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 464–482. Springer, Heidelberg, September 2018.

[DG16] Zeev Dvir and Sivakanth Gopi. 2-server pir with subpolynomial communication. *J. ACM*, 63(4), September 2016.

[DPSZ12] Ivan Damg$\alpha$rd, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *PoPETs*, 2018(4):159–178, 2018.

[ENOPC21] Aner Ben Efraim, Olga Nissenbaum, Eran Omri, and Anat Paskin-Cherniavsky. Psimple: Practical multiparty maliciously-secure private set intersection. Cryptology ePrint Archive, Report 2021/122, 2021. https://eprint.iacr.org/2021/122.

[ETS19] ETSI. *Multi-access Edge Computing*. 2019. https://www.etsi.org/technologies/multi-access-edge-computing.

[FGPS19] Eric J. Friedman, Vasilis Gkatzelis, Christos-Alexandros Psomas, and Scott Shenker. Fair and efficient memory sharing: Confronting free riders. In *The Thirty-Third AAAI Conference on Artificial Intelligence, USA,*, pages 1965–1972, 2019.

[FGS$^+$18] Dennis Felsch, Martin Grothe, Jörg Schwenk, Adam Czubak, and Marcin Szymanek. The dangers of key reuse: Practical attacks on IPsec IKE. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 567–583. USENIX Association, August 2018.

[FHLS19] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In Moses Charikar and Edith Cohen, editors, *51st ACM STOC*, pages 997–1008. ACM Press, June 2019.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.

[Gen09] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[GN19] Satrajit Ghosh and Tobias Nilges. An algebraic approach to maliciously secure private set intersection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 154–185. Springer, Heidelberg, May 2019.

[Goo10] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In Moses Charika, editor, *21st SODA*, pages 1262–1277. ACM-SIAM, January 2010.

[GS19] Satrajit Ghosh and Mark Simkin. The communication complexity of threshold private set intersection. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 3–29. Springer, Heidelberg, August 2019.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[HFH99] Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *EC*, pages 78–86, 1999.

[HV17] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 175–203. Springer, Heidelberg, March 2017.

[IKN$^+$19] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. Cryptology ePrint Archive, Report 2019/723, 2019. https://eprint.iacr.org/2019/723.

[JKU11] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122, 2011. https://eprint.iacr.org/2011/122.

[JPV12] K. V. Jonsson, K. Palmskog, and Y. Vigfusson. Secure distributed top-k aggregation. In *2012 IEEE International Conference on Communications (ICC)*, pages 804–809, June 2012.

[KFMB17] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. Robus: Fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 219–234, New York, NY, USA, 2017. ACM.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

[KMCK20] Myungsun Kim, Abedelaziz Mohaisen, Jung Cheon, and Yongdae Kim. Private top-k aggregation protocols. 02 2020.

[KMP$^+$17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1257–1272. ACM Press, October / November 2017.

[KMR11] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. http://eprint.iacr.org/2011/272.

[KRS$^+$19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 486–498, Berlin, Heidelberg, 2008. Springer-Verlag.

[LRG19] Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. Two-party private set intersection with an untrusted third party. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2403–2420. ACM Press, November 2019.

[LSX19] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In Timothy M. Chan, editor, *30th SODA*, pages 2419–2438. ACM-SIAM, January 2019.

[Man09] Dilip Many. Privacy-preserving collaboration in network security. 2009.

[Mea86] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *IEEE S&P*, 1986.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *USENIX Security 2004*, pages 287–302. USENIX Association, August 2004.

[MR18] Payman Mohassel and Peter Rindal. ABY$^3$: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[NMW10] Georg Neugebauer, Ulrike Meyer, and Susanne Wetzel. Fair and privacy-preserving multi-party protocols for reconciling ordered input sets. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, page 136–151, Berlin, Heidelberg, 2010. Springer-Verlag.

[NMW13] Georg Neugebauer, Ulrike Meyer, and Susanne Wetzel. Smc-muse: A framework for secure multi-party computation on multisets. In Matthias Horbach, editor, *INFORMATIK 2013 – Informatik angepasst an Mensch, Organisation und Umwelt*, pages 131–133, Bonn, 2013. Gesellschaft für Informatik e.V.

[NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, USA, 2007.

[Ode09] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[OOS17] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively Secure 1-out-of-N OT Extension with Application to Private Set Intersection. In *CT-RSA 2017 - RSA Conference Cryptographers' Track*, 2017.

[PIA$^+$16] K. Poularakis, G. Iosifidis, A. Argyriou, I. Koutsopoulos, and L. Tassiulas. Caching and operator cooperation policies for layered video content delivery. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[PIT$^+$18] G. S. Paschos, G. Iosifidis, M. Tao, D. Towsley, and G. Caire. The role of caching in future communication systems and networks. *IEEE Journal on Selected Areas in Communications*, 36(6):1111–1125, June 2018.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 2004.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431. Springer, Heidelberg, August 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767. Springer, Heidelberg, May 2020.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.

[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 122–153. Springer, Heidelberg, May 2019.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 125–157. Springer, Heidelberg, April / May 2018.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 797–812. USENIX Association, August 2014.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21, 2018.

[RJHK19] M. Sadegh Riazi, Mojan Javaheripi, Siam U. Hussain, and Farinaz Koushanfar. MPCircuits: Optimized circuit generation for secure multi-party computation. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, may 2019.

[RS20] Rahul Rachuri and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. The Network and Distributed System Security Symposium (NDSS) 2020, 2020.

[RS21] Peter Rindal and Phillipp Schoppmann. Vole-psi: Fast oprf and circuit-psi from vector-ole. Cryptology ePrint Archive, Report 2021/266, 2021. https://eprint.iacr.org/2021/266.

[Sha80] Adi Shamir. On the power of commutativity in cryptography. In *Automata, Languages and Programming*, 1980.

[Shi19] Elaine Shi. Path oblivious heap : Optimal and practical oblivious priority queue. 2019.

[SS08] Yingpeng Sang and Hong Shen. Privacy preserving set intersection based on bilinear groups. In *Proceedings of the Thirty-first Australasian Conference on Computer Science - Volume 74*, ACSC '08, pages 47–54, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient Multi-Party computation toolkit. https://github.com/emp-toolkit, 2016.

[WYG$^+$17] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 299–313, Boston, MA, March 2017. USENIX Association.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YHA19] J. Yao, T. Han, and N. Ansari. On mobile edge caching. *IEEE Communications Surveys Tutorials*, 21(3):2525–2553, thirdquarter 2019.

[YWZ+18] Y. Yu, W. Wang, J. Zhang, Q. Weng, and K. Ben Letaief. Opus: Fair and efficient cache sharing for in-memory data analytics. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 154–164, July 2018.

[ZLH+18] K. Zhang, S. Leng, Y. He, S. Maharjan, and Y. Zhang. Cooperative content caching in 5g networks with mobile edge computing. *IEEE Wireless Communications*, 25(3):80–87, JUNE 2018.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

# A Related work

## A.1 Cache related work

Edge caching has been considered as a key building block in the future communication network [PIT+18]. Various benefits and technical issues (e.g., network architecture, optimal content placement, content popularity prediction, cooperative caching) of edge caching have been investigated extensively in the recent literature [YHA19, BBD14, ETS19, 20117, PIT+18, ZLH+18]. However, most of the existing works studied edge caching from the perspective of a single network operator and have largely ignored the cooperation ability among network operators by exploiting the nonrivalry nature of the cached data items. It is worth emphasizing that the extensive literature on cooperative edge caching [PIT+18, ZLH+18] refers to the cooperation among caches owned by a single Telco. The advantages of cooperative cache sharing among multiple parties have only been realized and examined formally in recent works [PIA+16, KFMB17, YWZ+18, FGS+18, FGPS19] in the context of multi-tenant data centers. In these works, the authors focus mainly on economic aspects such as fairness and efficiency of cache sharing among different tenants. Different from the previous literature, our paper aims to address the multi-party cooperative cache sharing problem from the privacy perspective, which opens a new and interesting research direction on secure computation on PSI.

## A.2 PSI related work

There exist multiple constructions for computing PSI, which can be split into two groups: (1) constructions that output the intersection itself (PSI); and (2) constructions that output the result of a function $f$ computed on the intersection ($f$-PSI).

In this section, we focus on the state-of-the-art semi-honest PSI protocols. The earliest two-party PSI protocols, based on Diffie-Hellman assumptions, had been proposed in 1980s [Sha80, Mea86, HFH99], and we refer the reader to [PSZ18] for an overview of different PSI paradigms. The latest PSI protocols in the two-party setting are [PSZ18, PRTY19, GN19, GS19, CM20, PRTY20, KRS+19].

A multi-party PSI protocol was first proposed by Freedman *et al.* [FNP04]. The following works are [SS08, CJS12, HV17] which are heavily based on the public-key operation. This makes multi-party PSI impractical for the large set size. Later, Kolesnikov *et al.* [KMP+17] presented the first multi-party construction from symmetric primitives.

There are very few works on $f$-PSI, and most of them are in a two-party setting. Suppose that there are two parties, each has an input set of $m$ items. A naive circuit for $f$-PSI is to compare all pair and thus compute $O(m^2)$ comparisons. The sort-compare-shuffle $f$-PSI circuit of [HEK12] computes $O(m\log(m))$ comparisons. The work of [PSSZ15] based Cuckoo hashing requires a circuit of $O(m\log(m)/\log(\log(m)))$ comparisons. By proposing a new variant of Cuckoo hashing in two dimensions, [PSWW18]'s circuit has an upper-linear complexity of $\omega(m)$ comparisons. The first $f$-PSI protocols with a linear complexity are [PSTY19] and [CO18]. Even the circuit of [CO18] computes a linear number of comparisons, but the total communication complexity is higher than that of [PSTY19].

There exists several multi-party $f$-PSI protocols [Man09, Bf12, JKU11, NMW10, NMW13, RJHK19]. In [NMW10, NMW13], the authors provided a construction that outputs common items with the highest preference (rank) among all parties, which can be considered as a special case of our $k$-priority problem. Specifically, they use the polynomial to present the set of each party, and do the computation on these polynomials based on public-key operations (e.g., homomorphic encryption, Paillier cryptosystem). More precise, to represent an item with a preference of $v$, they replicate the item $k$ times. In other words, the polynomial has a form $P_i(x) = (x-x_1)^v(x-x_2)^{v-1}\dots(x-x_m)^1$ where $\{x_1,\dots,x_m\}$ is the input of the $i$-th party in the decreased order of the preference. For ranking, their solution is reasonable with small associated value $v$. However, $v$ can be a very large value in our case. Clearly, to adapt their protocol for MPCCache, the complexity is $O(mn2^\ell)$ where $2^\ell$ is the domain of the weighted value (e.g. $\ell = 16$ for 16-bit integer). Thus, their construction is expensive.

The work [JKU11] improved the constructions of [Man09, Bf12] with an additional feature that allows to output the top-k items. Unlike MPCCache, the output items in [JKU11] no need to be common items of all parties. For each item, [JKU11] simply takes the sum of its associated values from all parties that have the item (some parties may not have it), then chooses top-k items with largest sum values. In our application, a cached item needs to be common among all the parties to make cache sharing beneficial. More importantly, their algorithm [JKU11] firstly sorts the complete list of all parties's items, ordered by the item's identity, then computes the sum of the associated values of the same items and removes the duplication. Due to the oblivious sort on the set of size $mn$ and the oblivious deduplication, their approach requires much more number of secure computations than our construction which is $1.27m$, where $n$ and $m$ is the number of parties and the party's set size, respectively. Moreover, we need to modify [JKU11] to output only the common items of all parties. One promising direction is to maintain a counter for each item, indicating a number of the parties has the item and output only the item with the counter $n$. This step introduces a certain amount of cost due to equality checks inside MPC.

A very recent work [RJHK19] propose a customized circuit [RJHK19] as MPCircuits to compute PSI itself. One can extend MPCircuits to compute a top-k function on the secret intersection. We explicitly compare our proposed MPCCache with the MPCircuits-based solution in Section 7.4. The result shows that our scheme is at least $n\times$ faster than the MPCircuits-based approach.

### A.3 Secure top-k and oblivious sorting

**Secure Top-k Queries.** Although the top-k queries problem has many practical applications, there are only very few works [JPV12, KMCK20, BD10, CCD⁺20] that have studied this problem in the privacy-persevering setting. In [JPV12, KMCK20], the authors present algorithm that allows several participants $P_i$, each of them holds a secret set $S_i = \{x^i_1, \ldots, x^i_m\}$, to learn top-k elements from the union of the input sets $\bigcup_i S_i$. In this paper, we are interested in the problem where all parties' inputs are under a secret share form. Specifically, each party holds *only a secret share* of a set $S = \{x_1, \ldots, x_m\}$ (i.e., each party does not know what set S is), and wants to learn the top-K elements of $S$. While the work [BD10, CCD⁺20] can solve this problem, its solution is approximate. For exact top-K selection, a popular method for securely finding the top-k elements is to use an oblivious sort algorithm.

**Oblivious Sorting.** Two widely-used sorting networks are the AKS [AKS83] and randomized shellsort [Goo10] whose complexities are $O(m \log(m))$ for the set size $m$. However, it has a very high constant behind the big-O notation, and requires more round complexity. For example, from the analysis [BA12] of the sorting, a randomized shellsort uses around $5m \log(m)$ Compare-Swap operations but has a depth (i.e., the number of consecutive Compare-Swap) of approximate $5m$. The number of rounds is equal to this value multiplied by the round complexity of a Compare-Swap. Additionally, their constructions are optimal due to recent lower bounds provided in the work of [FHLS19, LSX19].

In the context of MPC, a more practical sorting algorithm is Batcher's network [Bat68] which requires $O(m \log^2(m)$ comparisons. Although Batcher's network has a high complexity and requires about $\frac{1}{4} m \log^2(m)$ Compare-Swap operations, they use odd-even merge sort which can be more effectively parallelized using only $\log(m)$ consecutive Compare-Swap. Moreover, Batcher's network needs fewer comparisons than randomized shellsort for $m \leq 2^{20}$.

Very recently, Shi [Shi19] proposed a path oblivious sort, which is preferrable than the Batcher's sorting network when the client-side storage is small. In our MPCCache model, the Telco's storage size is typically quite large. Hence, Batcher sort would result in fewer roundtrips.

## B Security Model In Multi-Party Computation

We formally present the security definition considered in this work, which follows the definition of [KMR11, Ode09] in the multi-party setting.

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

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

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

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

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

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

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

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

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

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

# C Security Proof

**Theorem 1.** *The protocol in Figure 5 securely computes the $k$-priority functionality defined in Figure 4 in the colluding semi-honest setting, given the ideal oblivious sorting and odd-even merging primitives defined in Section 3.4.*

**Sketch of proof:** To prove this theorem, we analyze each step in the algorithm with respect the *$k$-priority* functionality defined in Figure 4. The first step of the algorithm is locally executed by each party, which leaks nothing. In the second step, parties perform oblivious sorting $\mathcal{F}_{\mathsf{obv\text{-}sort}}$. Moreover, the output of $\mathcal{F}_{\mathsf{obv\text{-}sort}}$ is under a secret shared form, thus, as long as the $\mathcal{F}_{\mathsf{obv\text{-}sort}}$ used is secure, so is $k$-$\mathsf{priority}$ until this step. Step 3 recursively calls $\mathsf{LevelMerge}$, which base case is oblivious odd-even merging, and all intermediate output values of each recursive step are secret-shared. When using the oblivious odd-even merging functionality, the proof of step (3) is elementary.

Therefore, the security of our *$k$-priority* construction follows in a straightforward way from the security of its building blocks (*i.e.* oblivious sorting and odd-even merging) and the fact that all intermediate values are secret-shared. □

**Theorem 2.** *The construction of Figure 6 securely implements the $\mathsf{MPCCache}$ functionality defined in Figure 2 in the colluding semi-honest model, given the OPPRF primitive, $\mathsf{GC}$, and $k$-priority functionality described in Section 3.2, Section 3.3, and Figure 4, respectively.*

*Proof.* Denote $\mathcal{C}$ as a coalition of corrupt parties. We exhibit simulators $\mathsf{Sim}$ for simulating $\mathcal{C}$. *Sim* simulates the view of corrupt parties, which consists of $\mathcal{C}$'s randomness, input, output and received messages. $\mathsf{Sim}$ proceeds as follows. It calls a simulator for the zero-sharing key setup, and appends its output to the view. $\mathsf{Sim}$ simulates Online step (1) bin-by-bin. Let $S_i'$ be the set of elements of $S_i$ that are mapped to the $b^{th}$ bucket. If $\mathcal{C}$ do not contain the leader $P_1$, the corrupt parties receives nothing from the OPPRF functionality. Therefore, we only consider a case where $\mathcal{C}$ contains the leader $P_1$. In this case, $\mathsf{Sim}$ first chooses $\widetilde{y}_b^i \leftarrow \{0,1\}^*$ and $\widetilde{w}_b^i \leftarrow \{0,1\}^*$, and then calls OPPRF, $\mathsf{Enc}$ simulators. $\mathsf{Sim}$ then appends its output to the general view.

Finally, to simulate the online steps (1e) and 2, $\mathsf{Sim}$ runs simulator $\mathsf{Sim}_{\mathsf{GC}}$ and $\mathsf{Sim}_{k\text{-}\mathsf{priority}}$ on the inputs $t_b^i, s_b^i$, and $u_b$ and also append its output to the view.

We now argue the indistinguishability of the produced transcript from the real execution. For this, we formally show the simulation by proceeding the sequence of hybrid transcripts $T_0, \dots, T_4$, where $T_0$ is real view of $\mathcal{C}$, and $T_4$ is the output of $\mathsf{Sim}$.

*Hybrid 1.* Let $T_1$ be the same as $T_0$, except the zero-sharing key setup execution is replaced with running its simulator. Because pseudorandomness guarantees of the underlying simulator, $T_0$ and $T_1$ are indistinguishable.

*Hybrid 2.* Let $T_2$ be the same as $T_1$, except the OPPRF and $\mathsf{Enc}$ executions are replaced as follows. Consider two following cases:

- If $\mathcal{C}$ contains $P_1$: if $x_b^1$ is not common, all OPPRF's outputs are uniformly random from the view of $P_1$ as well as $\mathcal{C}$. Otherwise, $P_1$ receives $\mathsf{Enc}(x_b^1, \widetilde{y}_b^i || \widetilde{w}_b^i)$ and thus $(z_j^i \oplus t_b^i || v_j^i - s_b^i)$ for the OPPRF involving the non-colluding party $P_i$. We obverse that $t_b^i$ and $s_b^i$ is used only in the above expression. Since these values are uniform, so is $\widetilde{y}_b^i || \widetilde{w}_b^i$. Therefore, we replace OPPRF's outputs with random.
- If $\mathcal{C}$ does not contain $P_1$: In this case, $\mathcal{C}$ receives nothing from the OPPRF functionality and $\mathsf{Enc}$ scheme.

In summary, $T_2$ and $T_1$ are indistinguishable.

*Hybrid 3.* Let $T_3$ be the same as $T_2$, except the equality $\mathsf{GC}$ execution is replaced with running the simulator $\mathsf{Sim}_{\mathsf{GC}}(t_b^i, s_b^i, u_b), \forall i \in [n], b \in [\beta]$. Because $\mathsf{Sim}_{\mathsf{GC}}$ is guaranteed to produce output indistinguishable from real, $T_3$ and $T_2$ are indistinguishable.

*Hybrid 4.* Let $T_4$ be the same as $T_3$, except the $k$-$\mathsf{priority}$ execution is replaced with running the simulator $\mathsf{Sim}_{k\text{-}\mathsf{priority}}(u_b^i, \forall i \in [n], b \in [\beta])$. $k$-$\mathsf{priority}$ takes the secret-shared values corresponding to each bin, which allows to hide the intersection items. Moreover, the output of $\mathsf{Sim}_{k\text{-}\mathsf{priority}}$ is indistinguishable from real execution, thus $T_4$ and $T_3$ are indistinguishable. □

**Theorem 3.** *The construction of Figure 7 securely implements the* $\mathsf{MPCCache}$ *functionality defined in Figure 2 in the non-colluding semi-honest model, given the OPPRF primitive,* $\mathsf{GC}$*, and k-priority functionality described in Section 3.2, Section 3.3, and Figure 4, respectively.*

*Proof.* Denote $\mathcal{C}$ as a coalition of corrupt parties. We exhibit simulators $\mathsf{Sim}$ for simulating $\mathcal{C}$. *Sim* simulates the view of corrupt parties, which consists of $\mathcal{C}$'s randomness, input, output and received messages.

$\mathsf{Sim}$ first chooses $\widetilde{z}_j^i \leftarrow \{0,1\}^*$ and $\widetilde{s}_j^i \leftarrow \{0,1\}^*$, and calls OPPRF and $\mathsf{Enc}$ simulators. $\mathsf{Sim}$ then appends its output to the general view. The proof for "Server-working" phase is elementary as $\mathsf{Sim}$ runs two-party decentralized $\mathsf{MPCCache}$ protocol and appends its output to the view.

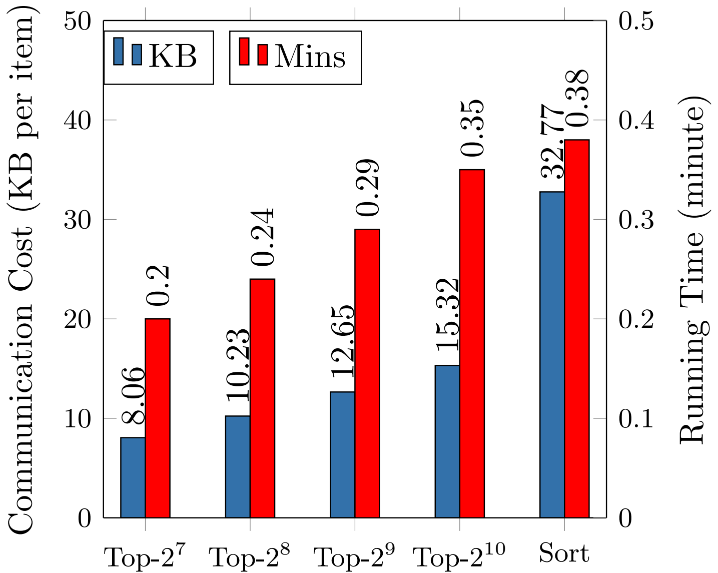


Figure 8: The total running time (red bar) in minute and communication cost (blue bar) per item in KB of our $k$-priority and oblivious sort for Top-$k$ and data set size $m = 2^{16}$.

If $\mathcal{C}$ do not contain any server $P_{e\in\{1,2\}}$, the corrupt parties receive nothing from the OPPRF functionality as well as the fist phase. Moreover, the second phase does not invoke $\mathcal{C}$. Therefore, our MPCCache protocol is secure in this case.

We consider a case where $\mathcal{C}$ corrupts one of two servers. Without loss of generality, we assume that $P_1 \in \mathcal{C}$. We now argue the indistinguishability of the produced transcript from the real execution. For this, we formally show the simulation by proceeding the sequence of hybrid transcripts $T_0, \ldots, T_2$, where $T_0$ is real view of $\mathcal{C}$, and $T_2$ is the output of Sim.

*Hybrid 1.* Let $T_1$ be the same as $T_0$, except the OPPRF execution is replaced as follows. For a specific item $x^1_j$, consider two following cases:

- If $x^1_j$ is not common, all OPPRF's outputs are uniformly random from the view of $P_1$ as well as $\mathcal{C}$.
- If $x^1_j$ is common, both servers $P_{e\in\{1,2\}}$ obtains the same $\widetilde{z^i_{j_i}}$ for the OPPRF involving the party $P_i$. In this protocol, we assume that $P_1$ and $P_2$ do not collude. Therefore, $\widetilde{z^i_{j_i}}$ looks random to the corrupted party $P_1$.

In summary, $T_1$ and $T_0$ are indistinguishable.

*Hybrid 2.* Let $T_2$ be the same as $T_1$, except the "server-working" execution is replaced with running the simulator our two-party MPCCache protocol. Because pseudorandomness guarantees of the underlying simulator (proved in Theorem 2) $T_2$ and $T_1$ are indistinguishable.

□

# D Cryptographic Gadget

## D.1 Garbled Circuit

Secure multi-party computation (MPC) allows a set of parties to jointly invoke a distributed computation while ensuring correctness, privacy of the parties' inputs, and more. Garbled Circuit (GC) is currently the most common generic technique for practical secure computation. GC was first introduced by Yao [Yao86] and Goldreich *et al.* [GMW87] for the two-party setting. Later, Beaver *et al.* [BMR90] and follow-up work [MNPS04, RJHK19] proposed GC for the multi-party setting with a constant-round protocol.

In the two-party setting, garbled circuit protocol [Yao86, GMW87] consists of a garbler and evaluator: the garbler encodes a function (e.g. equality, less than) into a garbled circuit using two random keys per each wire of the circuit; the evaluator first obtains corresponding keys of the input wires, and evaluates the circuit to learn the corresponding output wire key. The evaluator finally takes a decoding table, which maps the final output wire keys to the real values, and decodes the final output. In the multi-party setting, the garbled circuit [BMR90] is similar to the two-party construction, but each party jointly plays the role of both garbler and evaluator to garbling the circuit and evaluate it.

Garbled Circuit technique has seen dramatic improvements in recent years. The most notable optimized techniques are point-and-permute [BNP08], Free-XOR [KS08], the half-gate [ZRE15], and fixed-key AES garbling optimizations [BHKR13].

### D.2 Hashing Scheme

**Cuckoo hashing.** In basic Cuckoo hashing, there are $\beta$ bins denoted $B[1, \ldots, \beta]$, a stash, and $\widetilde{k}$ random hash functions $h_1, \ldots, h_{\widetilde{k}}$, each with range $\beta$. One can use a variant of Cuckoo hashing such that each item $x \in X$ is placed in exactly one of $\beta$ bins. Using the Cuckoo analysis [DRRT18] based on the set size $m$ of $X$, the parameters $\beta, \widetilde{k}$ are chosen so that with high probability $(1 - 2^{\lambda})$ every bin contains at most one item, and no item has to place in the stash during the Cuckoo eviction (i.e. no stash is required). In this paper, we therefore use $\widetilde{k} = 3$ hash functions and $\beta = 1.27m$ for our stash-less hashing scheme.

**Simple hashing.** One can map his points into bins using the same set of $\widetilde{k}$ Cuckoo hash functions (i.e., each item appears $\widetilde{k}$ times in the hash table). Using the standard ball and bin analysis based on $\beta, \widetilde{k}$, and the Cuckoo table size, one can deduce an upper bound $\gamma$ such that no bin contains more than $\gamma$ items with high probability $(1 - 2^{\lambda})$. According to [PSSZ15, DRRT18], $\gamma = O(\log(m))$. In this paper, we choose $\gamma$ equal to $2\log(m)$ for the set size of $m$.