Oblivious Batch Updates for Bloom-Filter-based Outsourced Cryptographic Protocols

Marten van Dijk[1,2,3] and Dandan Yuan[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, Netherlands
Marten.van.Dijk@cwi.nl, Dandan.Yuan@cwi.nl
[2] Vrije Universiteit Amsterdam, Amsterdam, Netherlands
[3] University of Connecticut, Storrs, USA

Abstract. In this work, we initiate the formal study of oblivious batch updates over outsourced encrypted Bloom filters, focusing on scenarios where a storage-limited sender must insert or delete batches of elements in a Bloom filter maintained on an untrusted server. Our survey identifies only two prior approaches (CCS 2008 and CCS 2012) that can be adapted to this problem. However, they either fail to provide adequate security in dynamic scenarios or incur prohibitive update costs that scale with the filter's maximum capacity rather than the actual batch size.

To address these limitations, we introduce a new cryptographic primitive, *Oblivious Bloom Filter Insertion* (OBFI), and propose novel constructions. At the core of our design is a novel building block, *Oblivious Bucket Distribution* (OBD), which enables a storage-limited sender to distribute a large array of elements, uniformly sampled from a finite domain, into small, fixed-size buckets in a data-oblivious manner determined by element order. The design of OBD is further supported by identifying and proving a new structural property of such arrays, which establishes tight and explicit probabilistic bounds on the number of elements falling within predefined subranges of the domain.

Our OBFI constructions achieve adaptive data-obliviousness and ensure that batch update costs scale primarily with the batch size. Depending on the variant, the sender's storage requirement ranges from $O(\lambda)$, where λ is the security parameter, down to $O(1)$. Finally, we demonstrate the practicality of OBFI by integrating it into representative Bloom-filter-based cryptographic protocols for Searchable Symmetric Encryption, Public-key Encryption with Keyword Search, and Outsourced Private Set Intersection, thereby obtaining batch-updatable counterparts with state-of-the-art security and performance.

1 Introduction

A wide range of cryptographic protocols—including Encrypted Search [6, 13, 14, 33, 36, 38, 42], Private Set Intersection and Union [1, 10, 18, 19, 31, 34, 39, 45], and Oblivious RAM [22, 50, 51] employ Bloom filters [11] or their variants as fundamental building blocks. A Bloom filter is a compact bit array that encodes a set and supports efficient membership queries by mapping each element to

multiple positions determined by independent hash functions. In line with the broader shift toward cloud-based storage and computation [29], many of these protocols [1,6,13,14,18,22,33,34,36,39,42,50,51] are either inherently designed for, or can be extended to, an outsourced setting. In this model, a Bloom filter encoding a set of elements is maintained on an untrusted server in encrypted form and can be queried securely by authorized parties.

In this work, we study the setting in which designated parties, referred to as *data senders*, are permitted to update the set encoded in the Bloom filter in an *oblivious* manner, *i.e.,* without revealing which bits are updated or how they are modified, while preserving the correctness and security of subsequent queries. Supporting updates obliviously is crucial: on the one hand, modern datasets are inherently dynamic; on the other, prior work [47, 54] has shown that non-oblivious updates could lead to significant leakage during query execution and render schemes vulnerable to plaintext-recovery attacks. While a sender can trivially perform a single oblivious insertion by linearly scanning the outsourced Bloom filter, we identify that the central challenge, discussed in detail later, lies in supporting efficient *batch updates* (*i.e.,* updating multiple elements simultaneously) under stringent local storage constraints. This challenge is practically significant: modern data continues to grow at scale, storage-limited senders such as IoT devices and mobile clients are increasingly prevalent, and batch updates form the backbone of large-scale data processing pipelines (*e.g.,* nightly record updates in enterprise systems). In this paper, we present the first formal study of this previously unexplored yet practically important problem.

Bloom Filter. A Bloom filter ($\mathbb{BF}$) is parameterized by the bit array size n and the number of hash functions d. Each hash function maps an arbitrary element to a position in the range $[0, n-1]$. Initially, all bits in $\mathbb{BF}$ are set to 0. To insert an element, the d hash functions are applied to compute d positions, and the corresponding bits are set to 1. To test whether an element belongs to the represented set, the Bloom filter checks whether all d corresponding bits are set to 1. As shown in the survey [43], the Bloom filter's compactness and query efficiency have led to its widespread adoption in modern non-cryptographic systems such as databases and computer networks. In cryptographic settings, Bloom filters offer additional advantages. In particular, the access pattern, *i.e.,* the sequence of memory locations accessed during a membership query, does not directly reveal the query result. This property has been leveraged in several works [13,22,36,42,50,51] to provide strong security guarantees without requiring storage proportional to the size of the element universe. Moreover, the Bloom filter's simple bit array structure makes it naturally compatible with cryptographic primitives that operate over arrays or Boolean values, including Oblivious Transfer [20,46], Private Information Retrieval [15,35], and Garbled-circuit-based Secure Computation [37,52]. These integrations have been used to realize more advanced encrypted query functionalities, such as subset queries [14, 36], Boolean queries [42], and set intersection [19, 34].

Bloom-Filter-based Outsourced Cryptographic Building Block. To abstract away design-specific details, we isolate the Bloom filter components that

recur across a wide range of cryptographic protocols [1, 6, 13, 14, 18, 22, 33, 34, 36, 39, 42, 50, 51] into a unified model, which we term the *Bloom-Filter-Based Outsourced Cryptographic (BFOC) Building Block*. A BFOC building block enables a data sender to construct an encrypted Bloom filter and outsources it to an untrusted server. Data users, who may or may not coincide with the sender, can then interact with the outsourced Bloom filter through secure queries, with the precise security guarantees determined by the underlying protocol. In most BFOC designs, the Bloom filter is encrypted bit by bit, with each ciphertext generated independently of the others. We refer to this property as *bitwise-independent encryption*. In this work, we restrict our attention to BFOC blocks employing this encryption paradigm. We further observe that many existing BFOC protocols would benefit substantially from supporting secure updates (by one or more senders) to outsourced Bloom filters, *e.g.,* to enable dynamic databases. For clarity, we focus first on insertion operations. Securely inserting a single element is straightforward: the sender can sequentially scan the outsourced Bloom filter, set the corresponding bits to 1, and write back their re-encrypted values. In contrast, enabling batch updates under stringent sender-side storage constraints is considerably more challenging. In what follows, we illustrate this challenge by analyzing naive approaches.

Naive Solutions for Oblivious Batch Insertions. When inserting a set $\mathbb{X}$, the sender first computes the hash positions of each element. Let $\mathbb{V}$ denote the array of all such hash positions. For each index $j \in [0, n-1]$, the sender must then ensure that $\mathbb{BF}[j] = 1$ if $j \in \mathbb{V}$, and leave it unchanged otherwise. A straightforward way to accomplish this is to stream each encrypted bit of the Bloom filter from the server, decrypt it locally, apply the update if necessary, and re-encrypt the result before writing it back.[4] The principal drawback of this approach lies in its sender-side storage requirements. While the dataset $\mathbb{X}$ itself can be outsourced and processed in a streaming fashion, the position array $\mathbb{V}$ must be held locally in order to determine which Bloom filter bits require modification securely. In practice, $\mathbb{V}$ may far exceed the sender's available memory, particularly when $\mathbb{X}$ is large or the sender is a resource-constrained device. This makes the approach infeasible in many realistic settings.

To mitigate this limitation, we consider two straightforward alternatives: (1) a partitioning-based method and (2) an approach leveraging the Oblivious Set (OSet) primitive [49]. In the partitioning approach, the sender divides $\mathbb{X}$ into l smaller subsets such that the hash positions generated from each subset fit within local storage. The sender then processes these subsets sequentially, updating the Bloom filter one subset at a time. However, this method is highly inefficient, as it requires scanning the entire Bloom filter l times. The second approach employs the OSet primitive, which extends Path ORAM [48] to enable a storage-limited sender to dynamically construct a set on the server and support membership queries without leaking information beyond the set size. In our context, the sender could outsource $\mathbb{V}$ via OSet and for each $j \in [0, n-1]$, test whether

[4] In settings where the sender is not permitted to learn the original bit values, homomorphic encryption could be used to implement the updates.

$j \in \mathbb{V}$. While this removes the local storage bottleneck, it is still impractical for two reasons: the high round complexity of constructing the oblivious set, and the cost of performing n expensive oblivious membership tests.

Literature Review. We review the existing literature to investigate whether prior work addresses the problem of oblivious batch updates in outsourced Bloom filters. To the best of our knowledge, no existing scheme directly targets this problem. However, two BFOC protocols proposed by Williams *et al.* in 2008 [51] and 2012 [50], referred to as WSC08 and WS12 respectively, provide techniques that can be adapted for this purpose. Both schemes were originally designed to support the oblivious creation of outsourced Bloom filters whose size exceeds the sender's local storage capacity, but their methods can be extended to enable batch insertions. Concretely, given a batch of elements $\mathbb{X}$ to be inserted into the outsourced Bloom filter $\mathbb{BF}$, the sender can use the two techniques to securely construct a fresh outsourced Bloom filter $\mathbb{BF}'$ that encodes $\mathbb{X}$. A bitwise update is then performed by scanning both $\mathbb{BF}$ and $\mathbb{BF}'$, and updating each position as $\mathbb{BF}[j] \leftarrow \mathbb{BF}[j] \vee \mathbb{BF}'[j]$ for all $0 \leq j < n$.

While this approach is technically feasible, both schemes exhibit critical limitations. WSC08 [51] fails to provide adequate security in dynamic settings—a limitation we will discuss in more detail later—and additionally requires substantial local storage. WS12 [50] addresses these weaknesses but introduces another significant inefficiency: its technique incurs a considerable overhead that scales with the size of the Bloom filter n, irrespective of the batch size. In particular, the protocol relies on oblivious sorting [8, 27] over a number of elements proportional to the Bloom filter size, even when inserting only a relatively small batch of elements. This dependence on n, rather than the actual batch size, significantly limits the practicality of WS12 in scenarios that involve frequent small-batch (relative to the filter size) updates.

1.1 Our Contributions and Techniques.

In this work, we contribute along three key dimensions: *formalization*, by establishing rigorous definitions of security and correctness; *construction*, by designing efficient protocols that meet these definitions; and *application*, by demonstrating their practicality in the context of well-studied cryptographic primitives.

1. Problem Formalization. We formalize the batch update problem within the context of the BFOC building block as a novel cryptographic primitive, which we refer to as *Oblivious Bloom Filter Insertion* (OBFI). The OBFI framework is designed to support both insertions and deletions, and to be compatible with both private-key and public-key settings. For clarity and simplicity, however, our formalization concentrates on batch insertions in the private-key setting. This formulation emphasizes two key aspects.

– **Generality.** The OBFI definition is deliberately broad, ensuring its applicability across a wide range of existing BFOC protocols. This generality

Table 1. Oblivious Batch Insertion into Outsourced Bloom Filters: Comparison

Schemes	Local Storage	Computation & Communication	Rounds	Adaptive Security	False Positive Rate
Naive Solutions					
Partition	$O(\lambda)$	$O\left(\frac{mn}{\lambda}\right)$	$O\left(\frac{m}{\lambda}\right)$	✔	*Standard*
OSet [49]	$O(\log m)$	$O(n \log^2 m)$	$O(m \log^2 m)$	✔	*Standard*
Previous Work					
WSC08 [51]	$O(\lambda\sqrt{m})$	$O(m \log \log m)$	$O(\log \log m)$	✘	
WS12 [50]	$O(1)$	$O(n \log n)$	$O(\log n)$	✔	*Standard*
Our Work					
OBFI (Variant 1)	$O(\lambda)$	$O(m \log m)$	$O(\log m)$	✔	*Standard*
OBFI (Variant 2)	$O(1)$	$O(m \log m)$	$O(\log m)$	✔	*Slight Increase*

All reported computational & communication overheads exclude the cost of a single linear scan of the Bloom filter. Throughout, λ denotes the security parameter, m the batch size, and n the size of the Bloom filter. We assume that the partition-based naive solution requires $O(\lambda)$ sender-side storage, consistent with OBFI (Variant 1). `False Positive Rate` denotes the probability that a membership query on the updated Bloom filter returns `true` for an element that is not a member of the represented set. *Standard* indicates that the false positive rate of the updated Bloom filter matches that of a standard Bloom filter, as defined in Section 2.2. In contrast, OBFI (Variant 2) introduces a *slight increase* in the false positive rate after some batch updates, as formalized in Lemma 2.

stems from the observation that any BFOC protocol employing a bitwise-independent encryption scheme $\mathcal{BE}$ preserves query correctness as long as the encrypted Bloom filter produced after each batch insertion correctly encodes all elements inserted thus far under $\mathcal{BE}$. Accordingly, our algorithmic definition focuses exclusively on the insertion operation, while abstracting away the description of the query phase. Correctness is then defined by requiring that the updated Bloom filter faithfully represents the complete set of inserted elements and remains encrypted under a consistent bitwise-independent encryption scheme.

– **Adaptive Security.** We define a notion of adaptive data-obliviousness for OBFI, which guarantees that the access patterns generated during batch insertions leak no information beyond the batch size, even against an adaptive adversary that selects batches based on previously observed views. This property provides strong security guarantees that are crucial in dynamic cryptographic protocols. We do not treat the batch size as sensitive, since any batch can be padded to a predefined upper bound, and disclosing this bound may not compromise security.

2. Novel OBFI Constructions and Extensions. We propose two secure and efficient constructions, OBFI (Variant 1) and OBFI (Variant 2), along with their extensions. A comparative evaluation of the two constructions against naive approaches and prior work is summarized in Table 1.

High-level Framework. Our design begins with the observation that a single linear scan of the Bloom filter is unavoidable to re-encrypt all bits and hide which positions are modified. The central performance goal is therefore to ensure that all other overhead scales with the batch size m, rather than with the Bloom

filter size n. Following the high-level idea of WSC08 [51], we partition the Bloom filter into contiguous, non-overlapping, equal-sized segments. By modeling each hash output as uniform randomness, the array of hash positions $\mathbb{V}$ matched by the batch can be obliviously distributed into small buckets, one per segment. After distribution, each bucket contains the positions belonging to its segment, enabling the sender to update the Bloom filter segment by segment by checking membership against the associated bucket.

Technical Challenge & Insufficient Security of WSC08. The main challenge lies in making the distribution process oblivious, while maintaining the desired performance guarantees. In contrast, the method in WSC08 fails to meet both goals. They set the number of segments to $\sqrt{m}$, maintain local counters for each segment, and then set the bucket size to the maximum value of these counters. This leaks information since the bucket size (visible to the server) may vary across batches of the same size, revealing patterns such as batch equality. Moreover, their construction requires the sender to maintain local storage of size $O(\lambda\sqrt{m})$, where λ denotes the security parameter, which may impose a storage burden when handling large batches.

Oblivious Bucket Distribution (OBD). To address this, we introduce a new cryptographic primitive, *Oblivious Bucket Distribution* (OBD), and design an efficient OBD protocol. A key technical contribution is a lemma establishing a threshold $t = \Theta(\lambda)$ (with an explicit value specified in Lemma 1) that, for an appropriate segment size, simultaneously serves as an upper bound on the number of values from $\mathbb{V}$ that may fall into any single segment, and as a lower bound across any two adjacent segments. This threshold applies to arbitrary arrays $\mathbb{V}$ with overwhelming probability and allows us to set a uniform bucket size t without leaking information. In our design, each bucket includes both the real values from $\mathbb{V}$ in its segment and padded values from the next segment. The lemma guarantees that each element is assigned to at most two buckets. Oblivious distribution is then achieved via two black-box invocations of the oblivious sort (OSORT) primitive [5, 8, 28]: The first sort is applied to $\mathbb{V}$, after which the sender scans the sorted array to assign two bucket indices (with a dummy index if only one is valid) to each element with a scan. The second sort groups elements by bucket index and eliminates dummy entries. The protocol uses only constant sender-side storage, with its cost dominated by the oblivious sorting of $2m$ elements.

Two Variants. After OBD, the writing phase diverges into two variants. In Variant 1, the sender processes one bucket at a time, retaining it locally to update the corresponding filter segment. This approach requires sender storage linear with the bucket size. To eliminate this storage cost, Variant 2 outsources each bucket as an individual Bloom filter using the technique of WS12 [50], thereby reducing the sender's storage to a constant. Updates are then performed through membership queries to these outsourced filters. The trade-off is that Bloom filter membership queries inherently admit false positives, which may cause some zero bits to be erroneously set to one, increasing the overall false positive rate. Nevertheless, we prove that this increase remains minor even after 10^4 batch updates, as formalized in Lemma 2.

Extensions. We extend our framework in three additional directions: enabling deletions by incorporating counting Bloom filters [21]; supporting a public-key setting; and devising a strategy to further reduce the sender-side storage in Variant 1 without increasing the false positive rate.
Summary of Advantages. Overall, our constructions provide three key benefits:

- We propose the *first* adaptively secure OBFI protocols, with update cost dependent only on batch size (except for a single linear scan over the filter).
- We introduce a new OBD primitive based on *novel probabilistic bounds* for uniformly distributed arrays. This technique is of independent interest and may find applications beyond the scope of this work.
- Our protocols treat OSORT as a black-box primitive and require only a small constant number of additional linear-time scans. This *modular design* allows implementations to directly leverage well-established optimizations, such as the highly parallelizable Bitonic sort [8], as well as advanced extensions like outsourced two-server oblivious sorting [7].

3. Applications of OBFI. We further demonstrate the versatility of OBFI by integrating it into three representative BFOC protocols drawn from well-studied cryptographic primitives: Searchable Symmetric Encryption (SSE), Public-key Encryption with Keyword Search (PEKS), and Outsourced Private Set Intersection (OPSI). In each case, OBFI enables efficient batch updates while achieving state-of-the-art security and performance. Our main results are as follows:

- **SSE.** By integrating OBFI with the conjunctive static SSE scheme proposed by Lai *et al.* [36], we obtain a dynamic SSE construction that effectively hides intermediate results during conjunctive search. Compared with the state-of-the-art dynamic conjunctive SSE scheme [53], our construction achieves substantially better storage efficiency while providing comparable query security and performance guarantees.
- **PEKS.** Applying OBFI to the only known access-pattern-hiding PEKS scheme of Boneh *et al.* [13], we significantly improve insertion efficiency. This enhancement is achieved through a novel public-key realization of oblivious batch updates into a Bloom filter variant.
- **OPSI.** By directly incorporating OBFI into the outsourced Bloom filters used in Kerschbaum's OPSI scheme [34], we obtain the first updatable OPSI construction that hides access patterns during updates.

2 Preliminaries

In this section, we first introduce the notations used throughout the paper, followed by a brief overview of Bloom filters. We then present the BFOC building block and its associated sub-concept, Bitwise-Independent Encryption. Next, we define several oblivious protocols: OSORT, OBD, and OBFI, where OBD and OBFI are proposed for the first time in this work.

Throughout the paper, the term *untrusted server* refers to an *honest-but-curious* adversary: the server follows the prescribed protocol but attempts to infer as much private information as possible from its execution.

2.1 Notations

We denote by $\{0,1\}^\ell$ the set of all binary strings of length ℓ, and by $\{0,1\}^*$ the set of binary strings of arbitrary length. The notation $[a,b]$ represents the set of integers from a to b. The assignment $a_1 \leftarrow a_2$ means that the value of a_2 is assigned to the variable a_1, while $a \xleftarrow{\$} S$ denotes that a is sampled uniformly at random from the set S. For a container C (*e.g.,* a set or array), $|\mathsf{C}|$ denotes its cardinality. We write $\perp$ for the empty symbol. For a pair p, $p.\mathsf{first}$ and $p.\mathsf{second}$ denote its first and second components, respectively. When comparing two pairs, the comparison is lexicographic: first by the first components, and, if these are equal, by the second.

We denote by $\mathcal{SE}$ a semantically secure symmetric encryption scheme, consisting of the following algorithms: a key-generation algorithm $\mathcal{SE}.\mathsf{Gen}(1^\lambda) \to \{0,1\}^\lambda$, an encryption algorithm $\mathcal{SE}.\mathsf{Enc}(\{0,1\}^\lambda, \{0,1\}^*) \to \{0,1\}^*$, and a decryption algorithm $\mathcal{SE}.\mathsf{Dec}(\{0,1\}^\lambda, \{0,1\}^*) \to \{0,1\}^*$.

Given two distribution ensembles $\{X_\lambda\}$ and $\{Y_\lambda\}$ indexed by the security parameter λ, we write $\{X_\lambda\} \stackrel{c}{\equiv} \{Y_\lambda\}$ to denote that the ensembles are computationally indistinguishable, *i.e.,* no probabilistic polynomial-time (PPT) adversary can distinguish them with more than negligible advantage.

For a protocol Prot, we write $\mathsf{Prot}^{\mathbf{\Pi}}$, where $\mathbf{\Pi}$ denotes one or more encryption schemes, to indicate that the inputs and/or outputs of Prot may include components produced by the algorithms of the schemes in $\mathbf{\Pi}$.

2.2 Bloom Filter

A Bloom filter [11] is a space-efficient probabilistic data structure that represents a set of M elements using an n-bit array. It supports element insertions and membership queries, *i.e.,* determining whether a given element belongs to the set. Bloom filters may yield *false positives*, incorrectly reporting that an element is in the set, but they never produce *false negatives*. To construct a Bloom filter, d independent hash functions $\mathcal{H} = \{H_i\}_{i=0}^{d-1}$ are chosen, each mapping an input to an index in the range $[0, n-1]$. The bit array is initialized with all entries set to 0. To insert an element x, the filter computes the positions $\{H_i(x)\}_{i=0}^{d-1}$ and sets the corresponding bits to 1. To query whether an element x' is in the set, the filter checks whether all bits at positions $\{H_i(x')\}_{i=0}^{d-1}$ are set to 1. If so, the query returns *true*; otherwise, it returns *false*. False positives occur when all d positions associated with a queried element not in the set have already been set to 1 by previous insertions. Assuming the hash functions are uniformly random, the false positive probability can be approximated as

$$\Pr_{\mathsf{fp}} \approx \left(1 - e^{-dM/n}\right)^d.$$

2.3 Bloom-Filter-Based Outsourced Cryptographic Building Block

To abstract away design-specific details, we introduce a reusable building block, denoted BFOC, which can be instantiated across a wide range of cryptographic protocols [1, 6, 13, 14, 18, 22, 33, 34, 36, 39, 42, 50, 51]. In a BFOC block, a Bloom filter is encrypted under a bitwise-independent encryption scheme and outsourced to an untrusted server by a data sender. Authorized data users can then interact with the outsourced Bloom filter via secure queries, while system-specific security guarantees are preserved. For clarity, we assume in this definition that the data sender and data user are the same entity, which we refer to collectively as the *client*, holding the same secret key. We first define bitwise-independent encryption, and then formally present the BFOC protocol.

Bitwise-Independent Encryption A bitwise-independent encryption scheme $\mathcal{BE}$ for a Bloom filter of size n consists of three algorithms:

- $\mathcal{BE}.\mathsf{BGen}(1^\lambda) \to K$: On input the security parameter 1^λ, this randomized algorithm outputs a secret key K.
- $\mathcal{BE}.\mathsf{BEnc}(K, j, b) \to c$: On input a key K, a position $j \in [0, n-1]$, and a bit $b \in \{0, 1\}$, this (possibly randomized) algorithm outputs a ciphertext c.
- $\mathcal{BE}.\mathsf{BDec}(K, j, c) \to b$: On input a key K, a position $j \in [0, n-1]$, and a ciphertext c, this deterministic algorithm outputs the plaintext bit b.

To encrypt a Bloom filter $\mathbb{BF}$ of size n using $\mathcal{BE}$, we compute the encrypted Bloom filter $\mathbb{EBF}$ as an array of size n, where each entry is given by $\mathbb{EBF}[i] \leftarrow \mathcal{BE}.\mathsf{BEnc}(K, i, \mathbb{BF}[i])$ for all $0 \le i < n$. For every security parameter λ, every key $K \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$, every position $j \in [0, n-1]$, and every bit $b \in \{0, 1\}$, correctness requires $\mathcal{BE}.\mathsf{BDec}(K, j, \mathcal{BE}.\mathsf{BEnc}(K, j, b)) = b$ holds except with negligible probability in λ.

The security of $\mathcal{BE}$ is defined with respect to the entire encrypted Bloom filter $\mathbb{EBF}$. If $\mathcal{BE}.\mathsf{BEnc}$ is deterministic, the scheme is secure against a weak form of Ciphertext-Only Attack (COA), which we refer to as single-instance COA (S-COA) security. Specifically, given only a single ciphertext array $\mathbb{EBF}$, no efficient adversary can infer any information about the underlying Bloom filter $\mathbb{BF}$ beyond its size. If $\mathcal{BE}.\mathsf{BEnc}$ is randomized, the scheme is secure under Chosen-Plaintext Attack (CPA). That is, even if an efficient adversary adaptively queries encryptions of Bloom filters of its choice, it cannot learn any information about the underlying bits of $\mathbb{BF}$. We formalize both security notions in Appendix A.

BFOC Building Block A BFOC building block specifies how encrypted Bloom filters are constructed and queried. Different instantiations may adopt different $\mathcal{BE}$ schemes and support different query functionalities. Let $\mathcal{F}_\mathsf{q}$ denote a query functionality, defined as

$$\mathcal{F}_\mathsf{q}(Q_c, Q_s, \mathbb{BF}) \to (R_c, R_s),$$

where Q_c and Q_s are inputs from the client and server, respectively, and R_c and R_s are their corresponding outputs. A BFOC block consists of the following:

- $\mathbf{Setup}^{\mathcal{BE}}(1^\lambda) \to K$: On input the security parameter 1^λ, the client runs $K \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$ and outputs the secret key K.
- $\mathbf{EncryptBF}^{\mathcal{BE}}(K, \mathbb{BF}; \perp) \to (\perp; \mathbb{EBF})$: The client inputs the secret key K and the Bloom filter $\mathbb{BF}$. Finally, the server outputs the encrypted Bloom filter $\mathbb{EBF}$ under $\mathcal{BE}$.
- $\mathbf{QueryEBF}^{\mathcal{BE}}(K, Q_c; Q_s, \mathbb{EBF}) \to (R_c; R_s)$: The client provides (K, Q_c) and the server provides $(Q_s, \mathbb{EBF})$. The protocol returns output R_c to the client and R_s to the server.

A BFOC building block with respect to query functionality $\mathcal{F}_\mathsf{q}$ typically satisfies the following correctness: For every λ, every $K \leftarrow \mathbf{Setup}^{\mathcal{BE}}(1^\lambda)$, every $\mathbb{BF} \in \{0,1\}^n$, every $\mathbb{EBF} \leftarrow \mathbf{EncryptBF}^{\mathcal{BE}}(K, \mathbb{BF}; \perp)$, and every valid (Q_c, Q_s), the protocol satisfies

$$\mathbf{QueryEBF}^{\mathcal{BE}}(K, Q_c; Q_s, \mathbb{EBF}) = \mathcal{F}_\mathsf{q}(Q_c, Q_s, \mathbb{BF}),$$

except with negligible probability in λ.

The query-security guarantees of BFOC blocks vary significantly across instantiations. Some schemes [18, 39] support only a single query, while others [1,6,13,14,22,33,34,36,42,51] allow queries to be issued repeatedly. Moreover, some constructions [1, 13, 14, 18, 22, 34, 39, 51] achieve full query security, leaking no information to the server during queries, whereas others [6, 33, 36, 42] trade a small amount of leakage for improved efficiency.

Dynamic Sets. The correctness property extends naturally to dynamic settings in which elements may be inserted or deleted after outsourcing. In particular, correctness is preserved as long as the most recent encrypted Bloom filter $\mathbb{EBF}$ satisfies:

$$\mathbb{BF}[j] = \mathcal{BE}.\mathsf{BDec}(K, j, \mathbb{EBF}[j]) \quad \forall j \in [0, n-1],$$

where $\mathbb{BF}$ denotes the Bloom filter encoding the up-to-date set.

This property underpins the notion of *Oblivious Bloom Filter Insertion* (OBFI), introduced in the following subsection. OBFI focuses solely on the secure update of encrypted Bloom filters while deliberately abstracting away query functionalities. In this setting, correctness reduces to verifying that the encrypted Bloom filter $\mathbb{EBF}$ satisfies the above condition. From a security standpoint, the update procedure must leak no information about the newly inserted elements or the contents of $\mathbb{EBF}$. In the context of concrete constructions, such obliviousness plays a crucial role in mitigating overall security risks, while the precise query-security guarantees ultimately depend on the specific protocol design.

2.4 Definitions of Oblivious Protocols

Oblivious algorithms [24, 25] were introduced to simulate programs in the RAM model while concealing memory access patterns. In this model, the CPU is assumed to have access to a small amount of private storage (e.g., registers) hidden from the adversary, while all other memory accesses may be observed. This abstraction naturally extends to outsourced computation, where a client

with limited local storage wishes to process data stored on an untrusted server. In the client-server setting, single-party oblivious algorithms can be recast as two-party protocols: the client interacts with the server to perform computations while ensuring that data access patterns remain hidden from the server.

In this subsection, we formalize three oblivious protocols in the client-server model: OSORT, OBD, and OBFI. The latter two are proposed in this work for the first time.

Oblivious Sort. Following [16], we define the OSORT protocol as follows:

- $\mathbf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EA}) \rightarrow (\perp; \mathbb{EA}^{\uparrow})$: The client holds a secret key $K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$ for a symmetric encryption scheme $\mathcal{SE}$. The server holds an encrypted array $\mathbb{EA} = \{ea_i \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, \mathbb{A}[i])\}_{i=0}^{|\mathbb{A}|-1}$, where $\mathbb{A}$ is the client's plaintext array. After executing the protocol, the server outputs an encrypted sorted array $\mathbb{EA}^{\uparrow}$.

We denote by $\mathcal{F}_{\mathsf{sort}}(\mathbb{A})$ the ideal sorting functionality, which outputs the sorted array $\mathbb{A}^{\uparrow}$ in ascending order.

Let $\mathsf{output}_{\mathsf{OSORT}}(1^\lambda, \mathbb{A})$ denote the array $\{\mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EA}^{\uparrow}[i])\}_{i=0}^{|\mathbb{A}|-1}$, where $\mathbb{EA}^{\uparrow}$ is the server's output from $\mathbf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EA})$.

Let $\mathsf{addresses}_{\mathsf{OSORT}}(1^\lambda, \mathbb{A})$ denote the sequence of server-side memory accesses observed during the execution of OSORT on the encryption of $\mathbb{A}$.

Definition 1 (Oblivious Sort). *A protocol OSORT is an oblivious sorting protocol if there exists a PPT simulator $\mathcal{S}$ such that, for every security parameter λ and every input array $\mathbb{A}$,*

$$(\mathsf{output}_{\mathsf{OSORT}}(1^\lambda, \mathbb{A}), \mathsf{addresses}_{\mathsf{OSORT}}(1^\lambda, \mathbb{A})) \stackrel{c}{\equiv} (\mathcal{F}_{\mathsf{sort}}(\mathbb{A}), \mathsf{addresses} \leftarrow \mathcal{S}(|\mathbb{A}|))$$

Oblivious Bucket Distribution. Unlike OSORT, which operates on arbitrary arrays, the OBD protocol is tailored for arrays $\mathbb{V}$ whose elements are sampled uniformly at random from a finite domain. In particular, each element $\mathbb{V}[i]$ is drawn independently and uniformly from $[0, n-1]$, and we denote the array size by s. The objective of OBD is to enable the client to obliviously distribute the encrypted elements of $\mathbb{V}$, stored on the server, into z buckets of identical capacity t, while revealing no information about $\mathbb{V}$ beyond its length s and the domain size n. We assume that the domain $[0, n-1]$ is partitioned into z contiguous subranges $\mathcal{W}_0, \ldots, \mathcal{W}_{z-1}$, each of size ω (except possibly the last has size smaller than ω), where $z = \lceil n/\omega \rceil$. The protocol guarantees, with overwhelming probability, that Bucket i contains all ciphertexts whose corresponding plaintext values lie in $\mathbb{V} \cap \mathcal{W}_i$. Equivalently, each element $v \in \mathbb{V}$ is placed into the bucket indexed by $\lfloor v/\omega \rfloor$. Both the number of buckets z and the bucket size t are determined as functions of n, s, and possibly other auxiliary parameters; however, they must remain independent of the actual contents of $\mathbb{V}$.

– $\mathbf{OBD}^{\mathcal{SE}}(K_e; \mathbb{EV}) \rightarrow (\perp; \widehat{\mathbb{EV}})$: The client holds a secret key $K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$. The server holds an encrypted array $\mathbb{EV} = \{ev_i \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, \mathbb{V}[i])\}_{i=0}^{s-1}$. The server outputs a list of encrypted buckets $\widehat{\mathbb{EV}} = \{\widehat{\mathcal{B}}_0, \ldots, \widehat{\mathcal{B}}_{z-1}\}$.

Let $\{\mathcal{F}^z_{\mathsf{distri}}(\mathbb{V})\}_{z\in[1,n]}$ denote a family of ideal bucket distribution functionalities indexed by z. For input $\mathbb{V}$, $\mathcal{F}^z_{\mathsf{distri}}(\mathbb{V})$ outputs the partition $\{\mathbb{V} \cap \mathcal{W}_i\}_{i=0}^{z-1}$.

Let $\mathsf{output}_{\mathsf{OBD}}(1^\lambda, \mathbb{V})$ denote the list of z decrypted buckets $\mathcal{B}_0, \ldots, \mathcal{B}_{z-1}$, where each $\mathcal{B}_i$ retains only the elements that belong to $\mathcal{W}_i$. Formally,

$$\mathcal{B}_i = \{\, \mathcal{SE}.\mathsf{Dec}(K_e, ev) \mid ev \in \widehat{\mathcal{B}}_i \,\} \cap \mathcal{W}_i,$$

where $\{\, \widehat{\mathcal{B}}_i \,\}_{i=0}^{z-1}$ is the server's output from executing OBD on $\mathbb{V}$.

Let $\mathsf{addresses}_{\mathsf{OBD}}(1^\lambda, \mathbb{V})$ denote the sequence of server-side memory accesses observed during the execution of OBD.

Definition 2 (Oblivious Bucket Distribution). *A protocol OBD is an oblivious bucket distribution scheme if there exists a PPT simulator $\mathcal{S}$ such that, for every security parameter λ, every $n, s \in \mathbb{N}$, and every input array $\mathbb{V}$ of size s with elements drawn uniformly at random from $[0, n-1]$, there exists some $z \in [1, n]$,*

$$(\mathsf{output}_{\mathsf{OBD}}(1^\lambda, \mathbb{V}), \mathsf{addresses}_{\mathsf{OBD}}(1^\lambda, \mathbb{V})) \stackrel{c}{\equiv} (\mathcal{F}^z_{\mathsf{distri}}(\mathbb{V}), \mathsf{addresses} \leftarrow \mathcal{S}(n, s))$$

$\mathrm{OBFIC}_{\mathrm{correct}_{\mathcal{A}}}(1^\lambda)$:
1: $\mathcal{A}$ selects M and gives it to the challenger.
2: The challenger runs $\mathbf{OBFISetup}^{\mathcal{BE},\mathcal{SE}}(1^\lambda, M;)$ to generate $(\mathcal{K} = (K_{b,0}, K_e), \mathcal{H})$, and $\mathbb{EBF}_0$.
3: $\mathcal{A}$ receives the initial empty Bloom filter $\mathbb{BF}_0 = 0^n$ and $\mathcal{H}$.
4: **for** $i = 1$ to $poly(\lambda)$ **do**
5: $\mathcal{A}$ adaptively selects a set $\mathbb{X}_i$ and gives it to the challenger.
6: The challenger runs $\mathbf{OBFIInsert}^{\mathcal{BE},\mathcal{SE}}(\mathcal{K} = (K_{b,i-1}, K_e), \mathcal{H}; \mathbb{EX}_i, \mathbb{EBF}_{i-1})$, yielding $(K_{b,i}, \mathbb{EBF}_i)$.
7: $\mathcal{A}$ receives $\mathsf{output}_i = \{\mathcal{BE}.\mathsf{BDec}(K_{b,i}, \mathbb{EBF}_i[j])\}_{j=0}^{n-1}$.
8: $\mathbb{BF}_i \leftarrow \mathcal{F}_{\mathsf{bfins}}(\mathbb{X}_i, \mathcal{H}, \mathbb{BF}_{i-1})$
9: **if** $\mathsf{output}_i \neq \mathbb{BF}_i$ **then**
10: **return** 0
11: **end if**
12: **end for**
13: **return** 1

Fig. 1. Correctness Experiments of OBFI

$\text{OBFIR}_{\text{EAL}_{\mathcal{A}}}(1^\lambda)$:

1: $\mathcal{A}$ selects M and gives it to the challenger.
2: The challenger runs $\textbf{OBFISetup}^{\mathcal{BE},\mathcal{SE}}(1^\lambda, M; \perp)$ to generate $(\mathcal{K} = (K_{b,0}, K_e)$, $\mathcal{H})$, and $\mathbb{EBF}_0$.
3: $\mathcal{A}$ observes access patterns $\mathsf{addresses}_0$ generated during the setup protocol.
4: **for** $i = 1$ to $poly(\lambda)$ **do**
5: $\quad\mathcal{A}$ adaptively selects a set $\mathbb{X}_i$ and gives it to the challenger.
6: $\quad$The challenger runs $\textbf{OBFIInsert}^{\mathcal{BE},\mathcal{SE}}(\mathcal{K} = (K_{b,i-1}, K_e), \mathcal{H}; \mathbb{EX}_i, \mathbb{EBF}_{i-1})$, yielding $(K_{b,i}, \mathbb{EBF}_i)$.
7: $\quad\mathcal{A}$ receives access patterns $\mathsf{addresses}_i$ generated by the above execution.
8: **end for**
9: With the view observed by $\mathcal{A}$, $\mathcal{A}$ outputs a bit b.

$\text{OBFII}_{\text{DEAL}_{\mathcal{A},\mathcal{S}}}(1^\lambda)$:

1: $\mathcal{A}(1^\lambda)$ selects M.
2: $\mathcal{A}$ receives $\mathsf{addresses}_0 \leftarrow \mathcal{S}(M)$.
3: **for** $i = 1$ to $poly(\lambda)$ **do**
4: $\quad\mathcal{A}$ adaptively selects a set $\mathbb{X}_i$ and receives $\mathsf{addresses}_i \leftarrow \mathcal{S}(|\mathbb{X}_i|)$.
5: **end for**
6: With the view observed by $\mathcal{A}$, $\mathcal{A}$ outputs a bit b.

Fig. 2. Security Experiments of OBFI

Oblivious Bloom Filter Insertion. We introduce OBFI, a primitive that enables a storage-constrained client to maintain an encrypted Bloom filter on an untrusted server while supporting a sequence of batch insertions in an oblivious manner. For simplicity, we assume that the Bloom filter is initially empty. Let M denote an upper bound on the total number of distinct elements that will ever be represented in the Bloom filter. An OBFI primitive consists of two sub-protocols:

- $\textbf{OBFISetup}^{\mathcal{BE},\mathcal{SE}}(1^\lambda, M; \perp) \rightarrow (\mathcal{K}, \mathcal{H}; \mathbb{EBF})$: The client takes as input the security parameter 1^λ and the bound M; the server has no input. At the end of the protocol, the client outputs a key pair $\mathcal{K} = (K_b, K_e)$, where K_b and K_e are secret keys for the bitwise-independent encryption scheme $\mathcal{BE}$ and the symmetric encryption scheme $\mathcal{SE}$, respectively, together with a family of d hash functions $\mathcal{H} = \{H_i : \{0,1\}^* \rightarrow [0, n-1]\}_{i=0}^{d-1}$. The server outputs the initial encrypted Bloom filter $\mathbb{EBF}$ of size n representing the empty set.
- $\textbf{OBFIInsert}^{\mathcal{BE},\mathcal{SE}}(\mathcal{K} = (K_b, K_e), \mathcal{H}; \mathbb{EX}, \mathbb{EBF}) \rightarrow (K'_b; \mathbb{EBF}')$: The client provides the key pair $\mathcal{K} = (K_b, K_e)$ together with the hash function family $\mathcal{H}$. The server provides as input an encrypted batch $\mathbb{EX} = \{ex \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, x) \mid x \in \mathbb{X}\}$, where $\mathbb{X}$ is the plaintext batch to be inserted, along with the current encrypted Bloom filter $\mathbb{EBF}$. After executing the protocol, the client may output a refreshed encryption key K'_b for $\mathcal{BE}$, while the server outputs the updated encrypted Bloom filter $\mathbb{EBF}'$.

Let $\mathcal{F}_{\mathsf{bfins}}$ be the ideal Bloom-filter insertion functionality. Given $(\mathbb{X}, \mathcal{H}, \mathbb{BF})$, it outputs the updated Bloom filter $\mathbb{BF}'$ defined, for $0 \le i \le n-1$, by

$$\mathbb{BF}'[i] \;=\; \begin{cases} 1, & \text{if } i \in \{\, H(x) \mid x \in \mathbb{X},\, H \in \mathcal{H} \,\}, \\ 1, & \text{if } \mathbb{BF}[i] = 1, \\ 0, & \text{otherwise.} \end{cases}$$

Remark (Statefulness). Unlike the stateless protocols OSORT and OBD, the OBFI protocol is inherently stateful, as the Bloom filter is incrementally updated across executions. This persistent state necessitates a stronger security model. Consequently, we formalize correctness and security for OBFI in the adaptive setting.

Definition 3 (Correctness of Oblivious Bloom Filter Insertion). *An OBFI scheme is correct if there exists a negligible function $negl(\cdot)$ such that, for every security parameter λ and every PPT adversary $\mathcal{A}$,*

$$\Pr[\mathrm{OBFIC}_{\mathrm{correct}_{\mathcal{A}}}(1^\lambda) = 0] \le negl(\lambda)$$

where $\mathrm{OBFIC}_{\mathrm{correct}_{\mathcal{A}}}(1^\lambda)$ *is given in Fig.1.*

Definition 4 (Adaptive Security of Oblivious Bloom Filter Insertion). *An OBFI scheme is adaptively secure if there exists a PPT simulator $\mathcal{S}$ and a negligible function $negl(\cdot)$ such that, for every security parameter λ and every PPT adversary $\mathcal{A}$,*

$$|\Pr[\mathrm{OBFIR}_{\mathrm{EAL}_{\mathcal{A}}}(1^\lambda) = 1] - \Pr[\mathrm{OBFII}_{\mathrm{DEAL}_{\mathcal{A},\mathcal{S}}}(1^\lambda) = 1]| \le negl(\lambda)$$

where $\mathrm{OBFIR}_{\mathrm{EAL}_{\mathcal{A}}}(1^\lambda)$ *and* $\mathrm{OBFII}_{\mathrm{DEAL}_{\mathcal{A},\mathcal{S}}}(1^\lambda)$ *are given in Fig.2.*

3 A Construction for Oblivious Bucket Distribution

An OBD protocol processes an array $\mathbb{V}$ of length s, where each element is sampled independently and uniformly from the domain $[0, n-1]$. In this section, we establish a novel probabilistic structural property of such arrays—one that, to the best of our knowledge, has not previously been formalized or proven. We then demonstrate how this property, when combined with the OSORT protocol, enables the design of an efficient OBD construction. Throughout this section, we adopt the notation from Section 2.4: z denotes the number of buckets, t the bucket capacity, and ω the subrange size, with $z = \lceil n/\omega \rceil$. We write $\mathcal{W}_0, \mathcal{W}_1, \ldots, \mathcal{W}_{z-1}$ for the contiguous subranges partitioning $[0, n-1]$, where $\mathcal{W}_i = [i \cdot \omega, (i+1) \cdot \omega - 1]$ for all $0 \le i \le z-2$, and $\mathcal{W}_{z-1} = [(z-1) \cdot \omega, n-1]$.

Tight and Explicit Bounds. We establish a new property for uniformly random arrays $\mathbb{V}$, formally stated in Lemma 1, which introduces a robustness parameter ρ and a constant γ. Informally, when the subrange size ω is chosen according to Formula (1), there exists a threshold $c = \Theta(\rho + \ln s)$, given explicitly in Formula (2), that satisfies: c serves as a strict upper bound on the

number of elements of $\mathbb{V}$ contained in any individual subrange, and at the same time c serves as a strict lower bound on the number of elements contained in the union of any two consecutive subranges. Both bounds hold simultaneously with probability at least $1 - e^{-\rho}$, as formally expressed in Formula 3.

Lemma 1. *Let $n, s \in \mathbb{N}$ and let $\rho > 0$ be a robustness parameter. Define $\gamma = \frac{3\sqrt{17}+13}{2}$, and set the subrange size ω as*

$$\omega = \begin{cases} \left\lceil \frac{\gamma n}{s} \left(\rho + \ln(2s)\right) \right\rceil, & \text{if } \gamma(\rho + \ln(2s)) < s, \\ n, & \text{otherwise.} \end{cases} \tag{1}$$

Define the threshold c as

$$c = \begin{cases} \frac{s\omega}{n} + \frac{1}{2}\mu + \frac{1}{2}\sqrt{\mu\left(8 \cdot \frac{s\omega}{n} + \mu\right)}, & \text{if } \gamma(\rho + \ln(2s)) < s, \\ s + 1, & \text{otherwise,} \end{cases} \tag{2}$$

where $\mu = \rho - \ln \frac{\omega}{2n}$ which satisfies $0 < \mu < \rho + \ln s$.

Consider any array $\mathbb{V}$ of length s whose elements are drawn independently and uniformly at random from the domain $[0, n-1]$. Then, with probability at least $1 - e^{-\rho}$, the following bounds hold simultaneously:

$$\forall\, i \in [0, \lceil n/\omega \rceil - 1] :\ |\mathcal{W}_i \cap \mathbb{V}| < c, \quad \text{and} \quad \forall\, i \in [0, \sigma] :\ |(\mathcal{W}_i \cup \mathcal{W}_{i+1}) \cap \mathbb{V}| > c, \tag{3}$$

where

$$\sigma = \begin{cases} \lceil n/\omega \rceil - 3, & \text{if } \omega \nmid n, \\ \lceil n/\omega \rceil - 2, & \text{if } \omega \mid n. \end{cases}$$

In particular, if $\gamma(\rho + \ln(2s)) \geq s$, then Formula (3) holds with probability 1.

Proof. The quantity $|\mathcal{W}_i \cap \mathbb{V}|$ can be represented as a sum of indicator random variables Y_j, where $Y_j = 1$ if $\mathbb{V}[j] \in \mathcal{W}_i$ and $Y_j = 0$ otherwise. Likewise, the size $|(\mathcal{W}_i \cup \mathcal{W}_{i+1}) \cap \mathbb{V}|$ can be written as the sum of corresponding indicator variables. By choosing a threshold c, Chernoff bounds can be applied to upper-bound the probabilities that $|\mathcal{W}_i \cap \mathbb{V}| \geq c$ and that $|(\mathcal{W}_i \cup \mathcal{W}_{i+1}) \cap \mathbb{V}| \leq c$. The problem then reduces to determining values of c and ω such that the summed probability (over all indices i) is at most $e^{-\rho}$. This ensures that Formula 3 holds with overall probability at least $1 - e^{-\rho}$. The resulting choices of c and ω are given in Lemma 1, and the full proof is provided in Appendix B.

Parameter Selection. In our OBD construction, we set $\rho = \lambda$ to ensure that the probability of violating the bounds in Formula 3 remains negligible in the security parameter λ. We define the number of buckets as $z = \lceil n/\omega \rceil$ and the bucket capacity as $t = \lceil c \rceil$, where ω and c are computed according to Formula 1 and Formula 2,, respectively, as established in Lemma 1.

High-Level Overview. Before execution, the client holds a secret key K_e, and the server holds an encrypted array $\mathbb{EV} = \{ev_j \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, \mathbb{V}[j])\}_{j=0}^{s-1}$. The

Client:

$\rho \leftarrow \lambda$

Select the integer ω that satisfies Formula 1 in Lemma 1, and value c that satisfies Formula 2.

$z \leftarrow \lceil \frac{n}{\omega} \rceil$, $t \leftarrow \lceil c \rceil$

The client and the server execute $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server obtains the sorted array $\mathbb{EV}^{\uparrow}$.

The client sends t dummy ciphertexts, each computed as $\mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum})$, to the server, which appends them to $\mathbb{EV}^{\uparrow}$.

Client:

$i \leftarrow -1$, $v_{-1} \leftarrow -1$
for $j = 0$ to $(s + t - 1)$ **do**
  Read $\mathbb{EV}^{\uparrow}[j]$ from $\mathbb{EV}^{\uparrow}$
  $v_j \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EV}^{\uparrow}[j])$
  if $v_{j-1} < (i+1)\cdot\omega$ and $(i+1)\cdot\omega \le v_j < (i+2)\cdot\omega$ and $v_j \neq \mathsf{dum}$ **then**
    $i \leftarrow i + 1$, $\delta_i \leftarrow j$
  else if $v_{j-1} < (i+1)\cdot\omega$ and $v_j \ge (i+2)\cdot\omega$ and $v_j \neq \mathsf{dum}$ **then**
    return
  else if $v_j = \mathsf{dum}$ and δ_{z-1} is absent **then**
    $i \leftarrow z - 1, \delta_{z-1} \leftarrow j$
  end if
  if $\big(i \ge 1$ and $(\delta_i - \delta_{i-1}) \ge t\big)$ or $\big(i \ge 2$ and $(\delta_i - \delta_{i-2}) < t\big)$ **then**
    return
  end if
  if $v_j \in \mathcal{W}_0$ **then** ▷ Case ❶
    $p_0 \leftarrow (0, v_j)$, $p_1 \leftarrow (+\infty, v_j)$
  else if $v_j \in \mathcal{W}_i$ and $j < \delta_{i-1} + t$ **then** ▷ Case ❷
    $p_0 \leftarrow (i, v_j)$, $p_1 \leftarrow (i - 1, v_j)$
  else if $v_j \in \mathcal{W}_i$ and $j \ge \delta_{i-1} + t$ **then** ▷ Case ❸
    $p_0 \leftarrow (i, v_j)$, $p_1 \leftarrow (+\infty, v_j)$
  else if $v_j = \mathsf{dum}$ and $j < \delta_{z-2} + t$ **then** ▷ Case ❹
    $p_0 \leftarrow (z - 2, \mathsf{dum})$, $p_1 \leftarrow (z - 1, \mathsf{dum})$
  else if $v_j = \mathsf{dum}$ and $\delta_{z-2} + t \le j < \delta_{z-1} + t$ **then** ▷ Case ❺
    $p_0 \leftarrow (+\infty, \mathsf{dum})$, $p_1 \leftarrow (z - 1, \mathsf{dum})$
  else if $v_j = \mathsf{dum}$ and $j \ge \delta_{z-1} + t$ **then** ▷ Case ❻
    $p_0 \leftarrow (+\infty, \mathsf{dum})$, $p_1 \leftarrow (+\infty, \mathsf{dum})$
  end if
  $ep_0 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, p_0.\mathsf{first}), \mathcal{SE}.\mathsf{Enc}(K_e, p_0.\mathsf{second}))$
  $ep_1 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, p_1.\mathsf{first}), \mathcal{SE}.\mathsf{Enc}(K_e, p_1.\mathsf{second}))$
  Send ep_0 and ep_1 to the server
end for

Server:

$\mathbb{EP} \leftarrow$ empty array
while Receive ep_0 and ep_1 **do**
  $\mathbb{EP} \leftarrow \mathbb{EP} \cup \{ep_0, ep_1\}$
end while

The client and the server run $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EP})$, after which the server obtains the sorted array $\mathbb{EP}^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.

Server:

Discard $\mathbb{EP}^{\uparrow}[zt]$ through the end.
$\widehat{\mathbb{EV}} \leftarrow \mathbb{EP}^{\uparrow}$ with all first entries removed
return $\widehat{\mathbb{EV}}$

Fig. 3. Protocol $\mathbf{OBD}^{\mathcal{SE}}(K_e; \mathbb{EV})$

goal is to assign, in an oblivious manner, the encrypted counterparts of all elements in $\mathbb{V} \cap \mathcal{W}_i$ to the corresponding bucket $\widehat{\mathcal{B}}_i$, for all $0 \le i \le z - 1$, except with negligible probability. A naive sequential assignment of ciphertexts to their buckets would reveal subrange membership, breaking obliviousness. To mitigate this, we first apply OSORT to $\mathbb{EV}$, thereby hiding the original order of ciphertexts. However, this still reveals the number of real elements per subrange. To conceal this, each bucket is padded with dummy ciphertexts so that its size is exactly t, regardless of the actual number of elements it contains.

Padding is achieved by using the first $(t - |\mathbb{V} \cap \mathcal{W}_i|)$ smallest elements of $\mathbb{V}$ that are $\ge i \cdot \omega$. Lemma 1 guarantees that, with overwhelming probability, these fillers lie in $\mathcal{W}_{i+1}$, ensuring that each element of $\mathbb{V}$ contributes to at most two buckets. An exception arises for $\widehat{\mathcal{B}}_{z-2}$ and $\widehat{\mathcal{B}}_{z-1}$ when $\omega \nmid n$, since $\mathbb{V} \cap \mathcal{W}_{z-1}$ may be insufficient to pad $\widehat{\mathcal{B}}_{z-2}$ and $\widehat{\mathcal{B}}_{z-1}$. To address this, we append t *dummy*

elements, defined as values outside $[0, n-1]$, to the end of $\mathbb{V}$. These dummy elements ensure that both $\widehat{\mathcal{B}}_{z-2}$ and $\widehat{\mathcal{B}}_{z-1}$ can be padded to size t.

To guarantee that the above procedure remains oblivious, we make two invocations of the OSORT primitive. Specifically,

1. The client and server first perform an OSORT on the input array $\mathbb{V}$, after which the array is padded with t *dummy* ciphertexts.

2. The client then sequentially determines the bucket assignment for each element. As discussed earlier, each element can be assigned to at most two buckets, where once as a real value and once as a padding value. Each assignment is represented as a pair, where the first entry denotes the assigned bucket index and the second entry is the element itself. To prevent the server from distinguishing between elements assigned to one bucket versus two, the client generates an additional pair whenever an element corresponds to only a single bucket. In this case, the first entry of the additional pair is set to $+\infty$ (*i.e.,* any value strictly greater than $z-1$). Each assignment pair is generated sequentially and encrypted by the client before being sent to the server, requiring only constant client storage.

3. Once all encrypted pairs have been transmitted, an additional OSORT is performed over the encrypted list of pairs, primarily ordered by the first entry (the bucket index). In this way, pairs with the same bucket index are grouped together to form a bucket, while those with index $+\infty$ are placed at the end and subsequently discarded.

Protocol Details. The algorithmic description of the OBD protocol is presented in Fig. 3. For clarity, we first introduce the key notations used in the algorithm. The notation $\mathbb{EV}^{\uparrow}$ denotes the encrypted and sorted array, padded with t *dummy* ciphertexts, each is obtained by encrypting a *dummy* value dum. During the assignment-pair generation phase, the client scans $\mathbb{EV}^{\uparrow}$, where v_j denotes the decryption of $\mathbb{EV}^{\uparrow}[j]$. The variable i tracks the index of the subrange $\mathcal{W}_i$ to which v_j belongs. For $0 \le i \le z-1$, we define δ_i as the index of the first element in $\mathbb{V}^{\uparrow}$ whose value is at least $i \cdot \omega$; equivalently, v_{δ_i} is the smallest element in $\mathbb{V} \cap \mathcal{W}_i$. Lemma 1 guarantees that $\mathbb{V} \cap \mathcal{W}_i$ is non-empty for all $0 \le i \le z-2$, ensuring the existence of δ_i for these i except with negligible probability in λ. However, when $\omega \nmid n$, the final subrange $\mathcal{W}_{z-1}$ may be empty, and δ_{z-1} may not exist. In this case, if a dummy value $v_j = \mathsf{dum}$ is encountered while δ_{z-1} is undefined, we set $\delta_{z-1} = j$. As shown in Lines 20–32 of Fig. 3, the assignment logic is partitioned into six cases. Boundary conditions are handled by Cases ❶, ❹, ❺, and ❻, while Cases ❷ and ❸ capture the general setting. These cases are carefully designed to generate assignment pairs such that each bucket corresponds to exactly t pairs. Pairs whose first entry is $+\infty$ are subsequently discarded (Line 82). The number of such discarded pairs equals the difference between the total number of generated pairs, $2(s+t)$, and the number of valid pairs, zt.

Performance of the OBD. Beyond the two invocations of OSORT—one on s elements and the other on $2(s+t)$ pairs, where $t = \Theta(\lambda)$—our OBD construction requires only two rounds of interaction between the client and the server. The two rounds only incur $O(s)$ overhead. Thus, assuming OSORT requires only constant client storage, $O(s \log s)$ computation and communication, and $O(\log s)$ rounds

of interaction to sort $O(s)$ elements (as is typical of most oblivious sorting protocols [4, 5, 27, 28]), our OBD protocol inherits the same asymptotic complexity.
Correctness and Security of the OBD. The correctness and security guarantees of our OBD construction are summarized in Theorem 1.

Theorem 1. *Let $\mathcal{SE}$ be a CPA secure symmetric encryption scheme, and let OSORT be an oblivious sorting protocol as defined in Definition 1. Then the proposed OBD construction is correct and secure as defined in Definition 2.*

Proof. The correctness of the OBD protocol is established via a sequence of three hybrid modifications. In these hybrids, we progressively replace the $\mathcal{SE}$ decryption of *dummy* ciphertexts, the output of OSORT, and the $\mathcal{SE}$ decryption on the sorted arrays with their corresponding ideal-functionality outputs. For example, decrypted ciphertext entries are replaced with the corresponding plaintext entries that were originally encrypted, and the output of OSORT is replaced with that of the ideal sorting functionality. After these modifications, the output of the protocol can differ from $\mathcal{F}^z_{\mathsf{distri}}(\mathbb{V})$ only if the protocol terminates prematurely due to a violation of the condition in Formula 3 of Lemma 1. However, such a violation occurs with probability at most $e^{-\lambda}$, which is negligible. Consequently, it follows that the output of OBD is computationally indistinguishable from $\mathcal{F}^z_{\mathsf{distri}}(\mathbb{V})$ provided that both OSORT and $\mathcal{SE}$ are correct.
Obliviousness is also established via a sequence of hybrids. First, the $\mathcal{SE}$ encryptions in ep_0 and ep_1 are replaced with encryptions of 0^l, where l is the bit-length of the corresponding plaintext. Second, all checks that could trigger early termination are omitted. Finally, the real OSORT process is replaced with its simulator $\mathcal{S}_{\mathsf{osort}}(\cdot)$. In the final hybrid, the resulting access pattern is exactly $\mathcal{S}_{\mathsf{obd}}(n, s)$, whose indistinguishability from the real access pattern follows from the CPA security of $\mathcal{SE}$, the bound of Lemma 1, and the obliviousness of OSORT. The full proof is provided in Appendix C.

4 Constructions for Oblivious Bloom Filter Insertion

In this section, we build upon the OBD protocol introduced in Section 3 to construct an OBFI scheme, denoted as OBFI (Variant 1), which requires $\Theta(\lambda)$ client storage. We then extend this construction to derive OBFI (Variant 2), which further reduces the client storage requirement to a constant size.

4.1 OBFI (Variant 1): $\Theta(\lambda)$ Client Storage

Below, we separately describe OBFISetup and OBFIInsert.
OBFISetup. In the setup phase, the client generates two secret keys: $K_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$ and $K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$. Given an upper bound M on the number of elements to be stored, the client determines the Bloom filter parameters: the size n and the number d of hash functions, according to the target false positive rate. It then selects d hash functions $\mathcal{H} = \{H_i : \{0,1\}^* \to [0, n-1]\}_{i=0}^{d-1}$, each modeled as a uniformly random function. The initial encrypted Bloom filter

$m \leftarrow |\mathbb{EX}|$
Client:
$(K_b, K_e) \leftarrow \mathcal{K}$, $\{H_i\}_{i=0}^{d-1} \leftarrow \mathcal{H}$
for $i = 0$ to $m - 1$ **do**
  Read $\mathbb{EX}[i]$ from $\mathbb{EX}$
  $x_i \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EX}[i])$
  for $j = 0$ to d-1 **do**
    $v \leftarrow H_j(x_i)$, $ev \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, v)$
    Send ev to the server
  end for
end for

Server:
$\mathbb{EV} \leftarrow$ empty array
while Receive ev **do**
  $\mathbb{EV} \leftarrow \mathbb{EV} \cup \{ev\}$
end while

The client and the server execute $\mathbf{OBD}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server receives $\widehat{\mathbb{EV}} = \{\widehat{\mathcal{B}}_0, \cdots, \widehat{\mathcal{B}}_{z-1}\}$.
The client keeps the values $\omega = \lceil n/z \rceil$ and the bucket capacity t.

Client:
if $\mathcal{BE}$ is S-COA secure **then**
  $K'_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$
end if
for $i = 0$ to $z - 1$ **do**
  $\mathcal{B}_i \leftarrow$ empty set
  Read $\widehat{\mathcal{B}}_i$ from $\widehat{\mathbb{EV}}$
  for each $ev \in \widehat{\mathcal{B}}_i$ **do**
    $v \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, ev)$
    if $v \in \mathcal{W}_i$ **then**
      $\mathcal{B}_i \leftarrow \mathcal{B}_i \cup \{v\}$
    end if
  end for
  for each $j \in \mathcal{W}_i$ **do**
    Read $\mathbb{EBF}[j]$ from $\mathbb{EBF}$
    $b_j \leftarrow \mathcal{BE}.\mathsf{BDec}(K_b, \mathbb{EBF}[j])$
    if $b_j = 1$ or $j \in \mathcal{B}_i$ **then**
      $b'_j \leftarrow 1$
    else
      $b'_j \leftarrow 0$
    end if
    if $\mathcal{BE}$ is CPA secure **then**
      $eb'_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, b'_j)$
    else if $\mathcal{BE}$ is S-COA secure **then**
      $eb'_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K'_b, b'_j)$
    end if
    Send (j, eb'_j) to the server
  end for
end for
if $\mathcal{BE}$ is CPA secure **then**
  return K_b
else if $\mathcal{BE}$ is S-COA secure **then**
  Update K_b in $\mathcal{K}$ to K'_b
  return K'_b
end if

Server:
while Receive (j, eb'_j) **do**
  Update $\mathbb{EBF}[j]$ to eb'_j
end while
return $\mathbb{EBF}$

Fig. 4. Protocol $\mathbf{OBFIInsert}^{\mathcal{BE},\mathcal{SE}}(\mathcal{K}, \mathcal{H}; \mathbb{EX}, \mathbb{EBF})$ in OBFI (Variant 1)

$\mathbb{EBF}$ is created by computing $\mathbb{EBF}[j] = \mathcal{BE}.\mathsf{BEnc}(K_b, j, 0)$ for all $0 \le j < n$, and streaming these ciphertexts to the server.

OBFIInsert. Fig. 4 presents the algorithmic description for OBFIInsert. The client inputs the key pair $\mathcal{K} = (K_b, K_e)$ and the set of hash functions $\mathcal{H}$. The server stores two inputs: $\mathbb{EX}$, the encrypted batch of size m, and $\mathbb{EBF}$, the current encrypted Bloom filter of size n. The procedure consists of three main steps.

1: Hash computation. For each element in the batch, the client computes d hash values. Each hash value is encrypted and sent to the server, which aggregates them into an array $\mathbb{EV}$ of size $m \cdot d$.

2: Oblivious Bucket Distribution. The two parties then execute the OBD protocol on $\mathbb{EV}$ to obliviously distribute the encrypted hash values into buckets $\widehat{\mathcal{B}}_0, \ldots, \widehat{\mathcal{B}}_{z-1}$. We adopt the definitions of the number of buckets z, bucket capacity t, subrange size ω, and subranges $\mathcal{W}_0, \ldots, \mathcal{W}_{z-1}$ as specified in Section 3. By construction, each bucket $\widehat{\mathcal{B}}_i$ contains the ciphertexts of all hash values that fall within the corresponding subrange $\mathcal{W}_i$, for every $0 \le i < z$, except with negligible probability. Upon completion, the client stores ω and t locally.

3: Bloom Filter Rewriting via Bucket Processing. After the OBD execution, the client processes the buckets sequentially. For each entry in $\widehat{\mathcal{B}}_i$, the client decrypts the ciphertext to obtain v and inserts v into the plaintext bucket $\mathcal{B}_i$ if $v \in \mathcal{W}_i$. Once $\mathcal{B}_i$ is constructed, the client streams the encrypted Bloom filter entries $\mathbb{EBF}[j]$ for all $j \in \mathcal{W}_i$, decrypts each to recover b_j, and computes the updated bit b'_j. The updated bit is then encrypted to obtain eb'_j, and the pair (j, eb'_j) is sent to the server, which updates the outsourced Bloom filter as $\mathbb{EBF}[j] \leftarrow eb'_j$.
Note. As discussed in Section 2.3, the bitwise-encryption scheme $\mathcal{BE}$ is either CPA secure or only S-COA secure. If CPA secure, the client can safely reuse K_b when encrypting b'_j. If only S-COA secure, however, the client must generate a fresh key K'_b before processing the buckets and use it for all new encryptions.

Performance of OBFI (Variant 1). Generating the hash values incurs a computational and communication cost of $O(m)$. Executing the OBD protocol over the encrypted hash values adds $O(m \log m)$ overhead. Both steps require only constant client-side storage. Afterward, the client processes one bucket at a time while streaming the corresponding $\mathbb{EBF}$ segment. These steps incur $O(m + n)$ overhead and require $\Theta(\lambda)$ storage. Overall, OBFI (Variant 1) achieves an asymptotic cost of $O(m \log m)$ in both computation and communication, in addition to the final linear scan over $\mathbb{EBF}$, with $\Theta(\lambda)$ client storage.

Correctness and Security of OBFI (Variant 1). The correctness and security of OBFI (Variant 1) are formalized in Theorems 2 and 3.

Theorem 2. *Let every hash function in $\mathcal{H}$ be modeled as a uniformly random function, let $\mathcal{SE}$ be a correct symmetric encryption scheme, let $\mathcal{BE}$ be a correct bitwise-independent encryption scheme, and let the OBD protocol from Section 3 be used as the OBD implementation. Then OBFI (Variant 1) satisfies correctness as defined in Definition 3.*

Proof. We establish correctness by constructing a sequence of five hybrid games. The first game corresponds exactly to $\text{OBFIC}_{\text{correct}}$, while the final game never outputs 0 as it implements the ideal Bloom filter insertion. The transition from the initial to the final game is achieved by successively replacing the decryption of $\mathcal{SE}$, the hash functions, the OBD protocol, and the decryption of $\mathcal{BE}$ with their respective ideal-functionality outputs. Consequently, if each hash function behaves as a uniformly random function, the probability that $\text{OBFIC}_{\text{correct}}$ outputs 0 is upper-bounded by the sum of the advantages of any PPT adversary in violating the correctness of these primitives. The complete proof is provided in Appendix D.

Theorem 3. *Let every hash function in $\mathcal{H}$ be modeled as a uniformly random function, let $\mathcal{SE}$ be a CPA-secure symmetric encryption scheme, let $\mathcal{BE}$ be a bitwise-independent encryption scheme that is either CPA secure or S-COA secure, and let the OBD protocol from Section 3 be used as the OBD implementation. Then OBFI (Variant 1) satisfies adaptive security as defined in Definition 4.*

Proof. The security proof proceeds via a standard hybrid argument constructed as a sequence of five games. The first game corresponds to the real security experiment, while the final game represents the ideal experiment. We show that

the views produced in each pair of adjacent games are computationally indistinguishable, relying sequentially throughout on the following assumptions: that each hash function behaves as a uniformly random function, the CPA security or S-COA security of $\mathcal{BE}$, the obliviousness of the OBD protocol, and the CPA security of $\mathcal{SE}$. This sequence of reductions establishes the computational indistinguishability between the real and ideal experiments, thereby proving the theorem. The complete proof is provided in Appendix E.

4.2 OBFI (Variant 2): Constant Client Storage

1: Run Line 1-15 of Fig. 4.

Client:
2: Select bucket Bloom filter parameters: the size n', and the number of hash functions d'.
3: Select d' hash functions $\mathcal{H}' = \{H'_i : \{0,1\}^* \to [0, n'-1]\}_{i=0}^{d'-1}$
4: Select a key $SK \xleftarrow{\$} \{0,1\}^\lambda$ for PRF F_1

5: **for** $i = 0$ to $z-1$ **do**
The client and the server execute WS12 [50] on the bucket $\widehat{\mathcal{B}}_i$. Upon completion, the server obtains the encrypted bucket Bloom filter $\widehat{\mathcal{BBF}}_i$.
Let $\mathcal{BBF}_i$ denote the Bloom filter encoding real elements in $\widehat{\mathcal{B}}_i$, constructed using the size parameter n' and hash functions $\mathcal{H}'$. The encrypted version $\widehat{\mathcal{BBF}}_i$ is generated by running $\widehat{\mathcal{BBF}}_i[j] \leftarrow F_2(F_1(SK,i),j) \oplus \mathcal{BBF}_i[j]$ for all $0 \le j \le n'-1$.
6: **end for**

Client:
7: $j \leftarrow 0$
8: **while** Read $\mathbb{EBF}[j]$ from $\mathbb{EBF}$ **do**
9: $b_j \leftarrow \mathcal{BE}.\mathsf{BDec}(K_b, \mathbb{EBF}[j])$
10: $i \leftarrow \lfloor \frac{j}{\omega} \rfloor$ ▷ i is the index of the subrange to which j belongs.
11: $SK_i \leftarrow F_1(SK, i)$
12: $bs \leftarrow$ empty string
13: **for** $k = 0$ to $d'-1$ **do**
14: $\epsilon \leftarrow H'_k(j)$
15: Read $\widehat{\mathcal{BBF}}_i[\epsilon]$ from $\widehat{\mathcal{BBF}}_i$
16: $b \leftarrow F_2(SK_i, \epsilon) \oplus \widehat{\mathcal{BBF}}_i[\epsilon]$
17: $bs \leftarrow bs||b$
18: **end for**
19: **if** $b_j = 1$ or $bs = 1^{d'}$ **then**
20: $b'_j \leftarrow 1$
21: **else**
22: $b'_j \leftarrow 0$
23: **end if**
24: Following the same procedure as shown in Line 36-49 of Fig. 4: encrypt b'_j to obtain eb'_j; send (j, eb'_j) to the server; update the encryption key K_b if required.
25: **end while**

Server
26: **while** Receive (j, eb'_j) **do**
27: $\mathbb{EBF}[j] \leftarrow eb'_j$
28: **end while**
29: **return** $\mathbb{EBF}$

Fig. 5. Protocol $\mathbf{OBFIInsert}^{\mathcal{BE},\mathcal{SE}}(\mathcal{K},\mathcal{H};\mathbb{EX},\mathbb{EBF})$ in OBFI (Variant 2)

We obtain OBFI (Variant 2) by eliminating the need for the client to locally store an entire bucket in the Bloom filter writing phase. A straightforward approach is for the client, for each Bloom filter index $j \in \mathcal{W}_i$, to sequentially scan and decrypt every entry in $\widehat{\mathcal{B}}_i$ to check for a match. However, this naive method incurs an additional overhead of $O(\lambda n)$, which is undesirable since n can be significantly larger than the number of inserted elements. In this section, we propose a more efficient technique that, when a small increase in the false positive rate is acceptable, introduces no additional asymptotic complexity.

The algorithmic description of the insertion protocol for OBFI (Variant 2) is presented in Fig. 5. The core idea is that, after computing the hash values and executing OBD to distribute them into z buckets, the client obliviously constructs an *outsourced encrypted Bloom filter* for each bucket. We refer to these as *bucket Bloom filters* to distinguish them from the main Bloom filter. For each $0 \leq i < z$, let $\widehat{\mathcal{BBF}}_i$ denote the encrypted bucket Bloom filter corresponding to bucket i, and $\mathcal{BBF}_i$ its plaintext counterpart encoding all hash positions falling within $\mathcal{W}_i$. Following the approach of WS12 [50], which supports the oblivious construction of outsourced encrypted Bloom filters using constant client storage, we adapt it to construct the bucket Bloom filters in our setting. Each bucket Bloom filter is encrypted bit by bit using two pseudorandom functions (PRFs),

$$F_1 : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda \quad \text{and} \quad F_2 : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}.$$

The client samples a secret key SK for F_1. For each $0 \leq i < z$ and each position ϵ in the bucket Bloom filter, encryption is performed as

$$\widehat{\mathcal{BBF}}_i[\epsilon] \leftarrow F_2(SK_i, \epsilon) \oplus \mathcal{BBF}_i[\epsilon], \quad \text{where } SK_i \leftarrow F_1(SK, i).$$

Once all encrypted bucket Bloom filters $\widehat{\mathcal{BBF}}_0, \ldots, \widehat{\mathcal{BBF}}_{z-1}$ have been constructed, the client scans each position j in $\mathbb{EBF}$ as in the original OBFI protocol, but with a modified membership check. Specifically, if $j \in \mathcal{W}_i$, the client sends the hash indices of j to the server, which responds with the corresponding ciphertexts from $\widehat{\mathcal{BBF}}_i$. The client decrypts these values using F_1 and F_2: if all decrypted bits are 1, j is treated as a member of bucket $\mathcal{B}_i$; otherwise, it is not. The client then updates $\mathbb{EBF}[j]$ according to the standard OBFI procedure.

Notably, a direct application of WS12 [50] to construct bucket Bloom filters fails to conceal from the server the number of real elements contained in each bucket. To overcome this limitation, we adapt their approach by introducing padding with dummies and incorporating a tagging mechanism, thereby obscuring this size information. Since these modifications are conceptually straightforward, we provide only the description of the adapted algorithm in Appendix F.

False Positive Rate of OBFI (Variant 2). In OBFI (Variant 2), each bit in the Bloom filter $\mathbb{BF}$ is set to 1 if and only if the corresponding bucket Bloom filter returns true on a membership query; otherwise, the bit remains 0. Since membership queries on bucket Bloom filters may yield false positives, bits that should remain 0 in $\mathbb{BF}$ may be erroneously set to 1, thereby increasing the false positive rate of queries on the main filter. The following lemma bounds this rate under the standard assumption that all hash functions used by $\mathbb{BF}$ and the bucket Bloom filters behave as independent, uniformly random functions.

Lemma 2 (False Positive Bound). *Let $\Pr_{\mathsf{fp}}^{\mathcal{B}}$ denote the false positive rate of a bucket Bloom filter, assumed to remain constant throughout the execution. After μ batch insertions, provided that $(1 - \mu \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}}) > 0$, the false positive rate of the Bloom filter $\mathbb{BF}$ in OBFI (Variant 2) satisfies*

$$\Pr_{\mathsf{fp}}^{\text{OBFI (Variant 2)}} \leq \left(1 - e^{-dM/n} \cdot \left(1 - \mu \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}}\right)\right)^d,$$

where n is the size of $\mathbb{BF}$, M is the maximum number of elements representable by $\mathbb{BF}$, and d is the number of hash functions used in its construction.

Proof. Let m^* denote the total number of elements inserted into $\mathbb{BF}$ after μ batch updates. Following the classical analysis of [11], the probability that a bit in $\mathbb{BF}$ remains 0 after inserting m^* elements is approximately $e^{-dm^*/n}$. Since each position in $\mathbb{BF}$ is queried against a bucket Bloom filter μ times, the probability that a bit which should remain 0 is incorrectly set to 1 due to false positives is

$$\Pr_{\mathsf{err}}^{0\to 1} \approx \mu \cdot e^{-dm^*/n} \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}}.$$

Thus, the false positive rate of $\mathbb{BF}$—*i.e.,* the probability that all d hash positions for a non-existent element are set to 1—is approximated by

$$\Pr_{\mathsf{fp}}^{\mathsf{OBFI\ (Variant\ 2)}} \approx \left(1 - e^{-dm^*/n} + \Pr_{\mathsf{err}}^{0\to 1}\right)^d = \left(1 - e^{-dm^*/n} \cdot (1 - \mu \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}})\right)^d.$$

Under the condition $(1 - \mu \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}}) > 0$, adopting the optimal Bloom filter parameterization that minimizes the false positive rate [11] (*i.e.,* $d{\cdot}M \approx (\ln 2){\cdot}n$), we obtain

$$\Pr_{\mathsf{fp}}^{\mathsf{OBFI\ (Variant\ 2)}} \leq \left(1 - e^{-dM/n} \cdot (1 - \mu \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}})\right)^d = \left(1 - \tfrac{1}{2} \cdot (1 - \mu \cdot \Pr_{\mathsf{fp}}^{\mathcal{B}})\right)^d.$$

Hence, the overall false positive rate remains effectively bounded by appropriately selecting μ and $\Pr_{\mathsf{fp}}^{\mathcal{B}}$. For instance, with $d = 20$, $\Pr_{\mathsf{fp}}^{\mathcal{B}} = 10^{-6}$, and $\mu \leq 10^4$, the false positive rate is bounded by 1.16×10^{-6}, which is only about 22% higher than that of a standard Bloom filter (9.54×10^{-7}).

Performance, Correctness, and Security of OBFI (Variant 2). The only difference between OBFI (Variant 1) and OBFI (Variant 2) lies in the bucket-querying mechanism: Variant 1 retains an entire bucket locally, whereas Variant 2 leverages outsourced bucket Bloom filters. Adopting the WS12 construction for a single outsourced bucket Bloom filter requires only constant client storage and incurs overhead of $O(\lambda \log \lambda)$. Consequently, constructing z such filters yields a total cost of $O(m \log \lambda)$. For the subsequent n membership queries, each query can be processed with constant cost. Therefore, Variant 2 attains the same asymptotic computational and communication complexity as Variant 1.

The security of (Variant 2) follows directly from that of (Variant 1), the WS12 construction, and the pseudorandomness of the functions F_1 and F_2. The correctness guarantees are also the same, except for one compromise: each bit in $\mathbb{BF}$ that should remain 0 has a probability $\Pr_{\mathsf{err}}^{0\to 1}$ of being erroneously set to 1.

5 Extensions

We extend our OBFI framework in three distinct directions, as outlined below.

Support for Deletions. To support deletions, we extend the OBFI protocol using a *counting Bloom filter* [21]. In this extension, the server maintains, alongside

the encrypted Bloom filter, an *encrypted counting Bloom filter* (with plaintext counterpart $\mathbb{CBF}$) of size n, where each counter is initialized to zero. Given a batch of updates $\mathbb{X} = \{(op_0, x_0), (op_1, x_1), \ldots, (op_{m-1}, x_{m-1})\}$, where each $op_i \in \{\mathsf{insert}, \mathsf{delete}\}$ and all x_i are distinct, the client first hashes each x_i and attaches the corresponding operation tag op_i to each hash value. Each tagged hash is then encrypted and sent to the server. The encrypted, tagged hashes are obliviously distributed into buckets using the OBD protocol. When processing bucket i, the client updates each position $\mathbb{BF}[j]$ for $j \in \mathcal{W}_i$ based on the corresponding counter in $\mathbb{CBF}$. Let $\pi_{j,0}$ denote the number of hash values in bucket i tagged as insert with value j, and $\pi_{j,1}$ the number tagged as delete with value j. $\mathbb{CBF}[j]$ is updated to $\mathbb{CBF}[j]+\pi_{j,0}-\pi_{j,1}$. Finally, if $\mathbb{CBF}[j] > 0$, we set $\mathbb{BF}[j] = 1$; otherwise, we set $\mathbb{BF}[j] = 0$. This extension preserves the false positive rate of $\mathbb{BF}$, but may introduce false negatives if deletions are performed incorrectly [30].

OBFI in the Public-Key Setting. Our OBFI protocols can be extended to the *public-key* setting, where any data sender possessing the public key can insert elements into the Bloom filter, while only the party possessing the corresponding secret key can perform queries. To achieve this, each Bloom filter bit is encrypted using a homomorphic encryption scheme that supports bitwise logical AND operations [23]. This design will ensure that the sender learns nothing about the current Bloom filter contents. In particular, to insert a batch, the data sender first runs our OBFI protocols to construct an encrypted Bloom filter (representing the batch) of the same length as the target filter, encrypting each bit using the homomorphic encryption scheme. The server then performs a component-wise homomorphic AND between the sender's encrypted Bloom filter and the current encrypted Bloom filter. However, note that the homomorphic encryption scheme may not support the specific query functionality or efficiency requirements. In such cases, the secret-key holder can scan the entire encrypted Bloom filter and re-encrypt it under a desirable encryption scheme.

Lowering the Robustness Parameter. To reduce sender storage in OBFI Variant 1 without increasing the false positive rate, we lower the robustness parameter ρ to a value where $e^{-\rho}$ remains small but not necessarily negligible in λ (*e.g.,* $\rho = 20$). This reduces the bucket size, but the bounds in Lemma 1 may then fail with a small probability, potentially leading to underfilled or overflowing buckets during the OBD phase. To address this, immediately after the first OSORT phase, the client performs a single scan to record information about both overflowing elements and underfilled buckets locally. Additionally, we modify the pair generation procedure to ensure that all buckets can be filled. Finally, during the Bloom filter writing phase, the overflow elements stored locally by the client are inserted into the Bloom filter alongside the corresponding buckets as they are read. Further details are provided in Appendix G.

6 Applications of the OBFI Primitive in BFOC Protocols

Our OBFI protocols apply broadly to BFOC protocols. A direct application is to eliminate the assumption that the sender must possess sufficient local re-

sources to construct a large-scale Bloom filter, an issue also addressed in [50,51]. More importantly, the primary contribution of OBFI lies in enabling a wide range of BFOC protocols to support secure and efficient batch updates. As illustrative case studies, we examine one representative Bloom-filter-based protocol from each of Searchable Symmetric Encryption (SSE), Public-key Encryption with Keyword Search (PEKS), and Outsourced Private Set Intersection (OPSI), and show how OBFI equip these protocols with batch-update functionalities while achieving state-of-the-art security and performance in the query phase.

SSE. An SSE scheme allows a client to outsource a collection of documents to an untrusted server securely. To enable efficient search, the client extracts the set of keywords contained in each document and builds an encrypted index. This index enables the secure and efficient retrieval of documents that match a keyword-based formula. Typical queries include *single-keyword search*, which returns all documents containing a given keyword, as well as more complex queries such as *conjunctive search*, which returns documents containing all queried keywords. We focus on HXT, a conjunctive SSE scheme proposed by Lai *et al.* [36], which represents the state-of-the-art in mitigating leakage of intermediate results during search, thereby effectively thwarting powerful file-injection attacks [54]. However, HXT is limited to static datasets.

At a high level, HXT integrates two components: (1) a single-keyword SSE scheme, and (2) an outsourced encrypted Bloom filter encoding all keyword–document pairs (w, id) existing in the database, where a pair exists if document id contains keyword w. For a conjunctive query $w_0 \wedge w_1 \wedge \cdots \wedge w_{\iota-1}$, the first component retrieves all documents containing w_0 (chosen as the least frequent keyword). For each candidate document id, the second component supports a subset query to verify whether all pairs $\{(w_i, id)\}_{i=1}^{\iota-1}$ are encoded in the outsourced Bloom filter. If the subset query returns *true*, id is included in the final result. To mitigate leakage, each Bloom filter bit is encrypted as a string of length $\Theta(\lambda)$, pseudorandomly derived from the bit value and its position under a secret key. Subset queries are then executed using the XOR-MAC technique [9], ensuring that negative results reveal nothing about which pair is absent. This substantially reduces the leakage of intermediate results.

Although subsequent work [44, 53] extends the first component of HXT to support dynamic datasets, its second component remains static. Our OBFI protocols directly address this limitation, as their bitwise-independent encryption strategy enables batch updates of keyword–document pairs in the outsourced Bloom filter. Assuming that m pairs are updated, our construction achieves an amortized update cost of $O(\log m + n/m)$, where n is the Bloom filter size (linear with an upper bound on the number of keyword-document pairs existing in the database), while preserving query correctness, efficiency, and security.

The only other conjunctive dynamic SSE scheme with comparable query security is HDXT proposed by Yuan *et al.* [53], where the Bloom filter component of HXT is replaced with a giant hash table of size equal to the product of the total number of documents and the total number of keywords (this product is typically much larger than n, particularly in sparse datasets), and the amortized update

cost scales proportionally to this product divided by the available client storage. By contrast, our OBFI-enabled dynamic HXT construction achieves substantially better batch update efficiency and storage efficiency.

PEKS. Unlike SSE, where the data sender and the data user are typically the same entity, PEKS separates these roles: the data user holds the private key, while any party with access to the user's public key may act as a sender. Such a sender can upload encrypted documents, each associated with a set of keywords, to an untrusted server. Only the user, who holds the private key, can search for documents containing a specific keyword. In 2007, Boneh *et al.* [13] introduced the first (and, to date, the only known) PEKS construction that achieves *full query privacy* (that is, the access pattern is completely hidden from the server) while maintaining sublinear query communication complexity, under the assumption that each keyword is associated with only a constant number of documents. We refer to this scheme as BKO+07.

BKO+07 builds on a Bloom filter variant called the the *Bloom Filter with Storage (BFS)*. In BFS, each bit of a standard Bloom filter is replaced with an array that stores elements, thereby enabling keyword-based retrieval. Specifically, BFS supports inserting a pair (x, y) by computing the hash positions of x and placing y into the arrays at those positions. To query x, the arrays at the corresponding hash positions are retrieved, and their intersection yields the result. In BKO+07, the BFS stores (*keyword*, *document*) associations, and it is kept by the server in an encrypted form. To search for a keyword, the user issues Private Information Retrieval (PIR) queries to the relevant positions, retrieves the corresponding arrays, and computes their intersection locally. Oblivious insertion in BKO+07 is achieved by implementing each BFS array as a dynamic buffer [40] proposed by Ostrovsky and Skeith [40], encrypted under a public-key encryption scheme supporting homomorphic evaluation of quadratic polynomials [12]. When a sender uploads a document, for each keyword it contains, the document is inserted into the corresponding buffers via homomorphic polynomial evaluation. To conceal the number of keywords per document, dummy insertions are added so that every document triggers exactly |W| insertions, where |W| is an upper bound on the number of keywords per document. This yields amortized communication overhead proportional to $\sqrt{n}$, while the amortized computational cost is proportional to n.

We improve the insertion phase of BKO+07 by modifying OBFI to enable batched, oblivious insertions of a document into all buffers associated with its keywords. Our construction incorporates two modifications to the original OBFI protocol. First, we preserve the BFS structure from BKO+07. During the BFS writing phase (performed after using OBD to distribute the hash values of the keywords into buckets), if a Bloom filter index falls within the corresponding bucket, the document is obliviously written into the associated buffer; otherwise, an empty string is inserted. As demonstrated by Ostrovsky and Skeith [40], such oblivious buffer writing can be efficiently realized by integrating their dynamic buffer construction with the Paillier cryptosystem [41]. Second, to hide the number of keywords per document, the generated hash values are padded

with dummy values (*i.e.,* uniformly random integers from $[0, n-1]$). Each hash value is labeled as "real" or "dummy", allowing dummy entries to be ignored during the BFS writing phase. This design achieves amortized insertion overhead proportional to $(\log|\mathrm{W}|+n/|\mathrm{W}|)$ in both communication and computation. Consequently, our scheme attains favorable communication complexity, where the precise comparison with BKO+07 depends on the value of $|\mathrm{W}|$, and provides a substantial improvement in computational efficiency.

OPSI. PSI allows two or more parties to compute the intersection of their private sets without revealing any information beyond the intersection itself. Recently, OPSI [1–3, 32, 34] has garnered increasing interest, where parties delegate both the storage of their private sets and the computation of their intersections to an untrusted cloud server. As Abadi *et al.* [1] demonstrate, supporting updates to outsourced private sets is important. As far as we know, they proposed the only existing updatable OPSI scheme; however, it incurs access pattern leakage. Here we present the first updatable OPSI scheme that hides access patterns, by extending Kerschbaum's Bloom-filter-based construction [34] with OBFI.

In Kerschbaum's scheme, user A encodes its private set into a Bloom filter, encrypts each bit using the Goldwasser-Micali (GM) homomorphic encryption [26], and sends the encrypted Bloom filter to an untrusted server. Another user B, using A's public key, generates a list of encrypted tokens (one for each element of its private set) using GM encryption and auxiliary techniques, and transmits them to the server. The server then performs homomorphic computation over A's encrypted Bloom filter and B's encrypted tokens, yielding an encrypted intersection result, which is returned to A for decryption.

Because A's encrypted Bloom filter is a bitwise-independent encryption of a standard Bloom filter, our OBFI protocols can be directly applied to enable user A to perform efficient batch updates. If B wishes to add elements, it can simply generate encrypted tokens for the new elements and send them to the server for inclusion in its encrypted token list. If B wishes to delete elements, however, it must regenerate tokens for its entire set, thereby preventing the server from learning which specific elements were removed.

7 Conclusion

In this work, we present the first formal study of the problem of batch updating elements in an encrypted Bloom filter outsourced to an untrusted server by introducing a new cryptographic primitive, OBFI. We design efficient OBFI protocols whose main technical contribution is the introduction of the OBD protocol, which we define and construct for the first time. The OBD protocol leverages our newly derived tight and explicit probabilistic bounds on the distribution of elements drawn uniformly at random from the domain $[0, n-1]$ across its subranges. Finally, we demonstrate the applicability of our OBFI protocols through concrete case studies, showing that they enable batch-updatable, Bloom-filter-based cryptographic protocols with state-of-the-art security and efficiency.

8 Acknowledgement

We want to thank Lynda Hardman for her valuable assistance with the organization of the paper.

This work was supported by the Dutch Research Council (NWO) Gravitation Programme *Challenges in Cyber Security* [Grant 024.006.037].

References

1. Abadi, A., Dong, C., Murdoch, S.J., Terzis, S.: Multi-party updatable delegated private set intersection. In: 26th Financial Cryptography and Data Security, FC 2022, Grenada, May 2-6, 2022. pp. 100–119. https://doi.org/10.1007/978-3-031-18283-9_6
2. Abadi, A., Terzis, S., Dong, C.: Vd-psi: Verifiable delegated private set intersection on outsourced private datasets. In: 20th International Conference on Financial Cryptography and Data Security, FC 2016, Christ Church, Barbados, February 22-26, 2016. pp. 149–168. https://doi.org/10.1007/978-3-662-54970-4_9
3. Abadi, A., Terzis, S., Metere, R., Dong, C.: Efficient delegated private set intersection on outsourced private datasets. IEEE Transactions on Dependable and Secure Computing **16**, 608–624 (2017). https://doi.org/10.1109/TDSC.2017.2708710
4. Ajtai, M., Komlós, J., Szemerédi, E.: An 0(n log n) sorting network. In: 15th Annual ACM Symposium on Theory of Computing, STOC 1983, Boston, Massachusetts, USA, April 25-27, 1983. pp. 1–9
5. Asharov, G., Chan, T.H.H., Nayak, K., Pass, R., Ren, L., Shi, E.: Bucket oblivious sort: An extremely simple oblivious sort. In: 3rd Symposium on Simplicity in Algorithms, SOSA 2020, Salt Lake City, UT, USA, January 6-7, 2020. pp. 8–14. https://doi.org/10.1137/1.9781611976014.2
6. Bag, A., Patranabis, S., Mukhopadhyay, D.: Tokenised multi-client provisioning for dynamic searchable encryption with forward and backward privacy. In: 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024. pp. 1691–1707. https://doi.org/10.1145/3634737.3657018
7. Baldimtsi, F., Ohrimenko, O.: Sorting and searching behind the curtain. In: 19th International Conference on Financial Cryptography and Data Security, FC 2015, San Juan, Puerto Rico, January 26-30, 2015. pp. 127–146. https://doi.org/10.1007/978-3-662-47854-7_8
8. Batcher, K.: Sorting networks and their applications. In: American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, April 30-May 2, 1968. pp. 307–314. https://doi.org/10.1145/1468075.1468121
9. Bellare, M., Guérin, R., Rogaway, P.: Xor macs: New methods for message authentication using finite pseudorandom functions. In: 15th Annual International Cryptology Conference, CRYPTO 1995, Santa Barbara, California, USA, August 27-31, 1995. pp. 15–28. https://doi.org/10.1007/3-540-44750-4_2
10. Ben-Efraim, A., Nissenbaum, O., Omri, E., Paskin-Cherniavsky, A.: Psimple: Practical multiparty maliciously-secure private set intersection. In: 17th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2022, Nagasaki, Japan, 30 May 2022 - 3 June 2022. pp. 1098–1112. https://doi.org/10.1145/3488932.3523254

11. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (July 1970). https://doi.org/10.1145/362686.362692
12. Boneh, D., Goh, E.J., Nissim, K.: Evaluating 2-dnf formulas on ciphertexts. In: 2nd Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. pp. 325–341. https://doi.org/10.1007/978-3-540-30576-7_18
13. Boneh, D., Kushilevitz, E., Ostrovsky, R., III, W.E.S.: Public key encryption that allows pir queries. In: 27th Annual International Cryptology Conference, CRYPTO 2007, Santa Barbara, CA, USA, August 19-23, 2007. pp. 50–67. https://doi.org/10.1007/978-3-540-74143-5_4
14. Boneh, D., Waters, B.: Conjunctive, subset, and range queries on encrypted data. In: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. pp. 535–554. https://doi.org/10.1007/978-3-540-70936-7_29
15. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 1999, Prague, Czech Republic, May 2-6, 1999. pp. 402–414. https://doi.org/10.1007/3-540-48910-X_28
16. Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: Cache-oblivious and data-oblivious sorting and applications. In: 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018. pp. 2201–2220. https://doi.org/10.1137/1.9781611975031.143
17. Chernoff, H.: A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. The Annals of Mathematical Statistics **23**(4), 493–507 (1952)
18. Davidson, A., Cid, C.: An efficient toolkit for computing private set operations. In: 22nd Australasian Conference on Information Security and Privacy, ACISP 2017, Auckland, New Zealand, July 3-5, 2017. pp. 261–278. https://doi.org/10.1007/978-3-319-59870-3_15
19. Dong, C., Chen, L., Wen, Z.: When private set intersection meets big data: An efficient and scalable protocol. In: 20th ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013. pp. 789–800. https://doi.org/10.1145/2508859.2516701
20. Even, S., Goldreich, O., Lempel, A.: A randomized protocol for ellis horowitz editor signing contracts. Communications of the ACM **28**(6), 637–647 (1985)
21. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: A scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking **8**, 281–293 (2000). https://doi.org/10.1109/90.851975
22. Fletcher, C., Naveed, M., Ren, L., Shi, E., Stefanov, E.: Bucket oram: Single online roundtrip, constant bandwidth oblivious ram. https://eprint.iacr.org/2015/1065 (2015)
23. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009. pp. 169–178. https://doi.org/10.1145/1536414.1536440
24. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: 19th Annual ACM Symposium on Theory of Computing, STOC 1987, New York, New York, USA, 1987. pp. 182–194. https://doi.org/10.1145/28395.28416
25. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM **3**, 431–473 (43). https://doi.org/10.1145/233551.233553

26. Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences **28**, 270–299 (1984). https://doi.org/10.1016/0022-0000(84)90070-9
27. Goodrich, M.T.: Randomized shellsort: A simple oblivious sorting algorithm. In: 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010. pp. 1262–1277. https://doi.org/10.1145/2049697.2049701
28. Goodrich, M.T.: Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In: 46th Annual ACM Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31-June 03, 2014. pp. 684–693. https://doi.org/10.1145/2591796.2591830
29. Grand View Research, Inc.: Cloud migration services market size, share & trends analysis report by platform, deployment, enterprise size, end use, region, and segment forecasts, 2025–2030 (2025)
30. Guo, D., Liu, Y., Li, X., Yang, P.: False negative problem of counting bloom filter. IEEE Transactions on Knowledge and Data Engineering **22**(5), 651–664 (2010). https://doi.org/10.1109/TKDE.2009.209
31. Inbar, R., Omri, E., Pinkas, B.: Efficient scalable multiparty private set-intersection via garbled bloom filters. In: 11th International Conference on Security and Cryptography for Networks, SCN 2018, Amalfi, Italy, September 5-7, 2018. pp. 235–252. https://doi.org/10.1007/978-3-319-98113-0_13
32. Jolfaei, A.A., Mala, H., Zarezadeh, M.: Eo-psi-ca: Efficient outsourced private set intersection cardinality. Journal of Information Security and Applications **65** (2022). https://doi.org/10.1016/j.jisa.2021.102996
33. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2017, Paris, France, April 30 - May 4, 2017. pp. 94–124. https://doi.org/10.1007/978-3-319-56617-7_4
34. Kerschbaum, F.: Outsourced private set intersection using homomorphic encryption. In: 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2012, Seoul, Korea, May 2-4, 2012. pp. 85–86. https://doi.org/10.1145/2414456.241450
35. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, Miami Beach, Florida, USA, October 19-22, 1997. pp. 364–373. https://doi.org/10.1109/SFCS.1997.646125
36. Lai, S., Patranabis, S., Sakzad, A., Liu, J.K., Mukhopadhyay, D., Steinfeld, R., Sun, S.F., Liu, D., Zuo, C.: Result pattern hiding searchable encryption for conjunctive queries. In: 25th ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 745–762. https://doi.org/10.1145/3243734.3243753
37. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two party computation. Journal of Cryptology **22**(2), 161–188 (2009). https://doi.org/10.1007/s00145-008-9036-8
38. Liu, D., Wang, W., Xu, P., Yang, L.T., Luo, B., Liang, K.: d-dse: Distinct dynamic searchable encryption resisting volume leakage in encrypted databases. In: 33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024

39. Niu, C., Wu, F., Tang, S., Hua, L., Jia, R., Lv, C., Wu, Z., Chen, G.: Billion-scale federated learning on mobile clients: A submodel design with tunable privacy. In: 26th Annual International Conference on Mobile Computing and Networking, MobiCom 2020, London, UK, September 21-25, 2020. pp. 1–14. https://doi.org/10.1145/3372224.3419188
40. Ostrovsky, R., III, W.E.S.: Private searching on streaming data. In: 25th Annual International Cryptology Conference, CRYPTO 2005, Santa Barbara, CA, USA, August 14-18, 2005. pp. 223–240. https://doi.org/10.1007/11535218_14
41. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: 18th International Conference on the Theory and Application of Cryptographic Techniques, EUROCRYPT 1999, Prague, Czech Republic, May 2-6, 1999. pp. 223–238. https://doi.org/10.1007/3-540-48910-X_16
42. Pappa, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S.G., George, W., Keromytis, A., Bellovin, S.: Blind seer: A scalable private dbms. In: 35th IEEE Symposium on Security and Privacy, S&P 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 359–374. https://doi.org/10.1109/SP.2014.30
43. Patgiri, R., Nayak, S., Muppalaneni, N.B.: Bloom Filter: A Data Structure for Computer Networking, Big Data, Cloud Computing, Internet of Things, Bioinformatics and Beyond. Academic Press (2023)
44. Patranabis, S., Mukhopadhyay, D.: Forward and backward private conjunctive searchable symmetric encryption. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, February 21-25, 2021. https://doi.org/10.14722/ndss.2021.23116
45. Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on ot extension. In: 23rd USENIX Security Symposium, USENIX Security 2014, San Diego, CA, USA, August 20-22, 2014
46. Rabin, M.O.: How to exchange secrets by oblivious transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard University (1981)
47. Salem, A., Bhattacharya, A., Backes, M., Fritz, M., Zhang, Y.: Updates-leak: Data set inference and reconstruction attacks in online learning. In: 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. pp. 1291–1308
48. Stefanov, E., van Dijk, M., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: 20th ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013. pp. 299–310. https://doi.org/10.1145/2508859.2516660
49. Wang, X.S., Nayak, K., Liu, C., Chan, T.H.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: 21st ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014. pp. 215–226. https://doi.org/10.1145/2660267.2660314
50. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: 19th ACM SIGSAC Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012. pp. 293–304. https://doi.org/10.1145/2382196.2382229
51. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: 15th ACM SIGSAC Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008. pp. 139–148. https://doi.org/10.1145/1455770.1455790

52. Yao, A.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, FOCS 1982, Chicago, Illinois, USA, November 3-5, 1982. https://doi.org/10.1109/SFCS.1982.38
53. Yuan, D., Zuo, C., Cui, S., Russello, G.: Result-pattern-hiding conjunctive searchable symmetric encryption with forward and backward privacy. In: 23rd Privacy Enhancing Technologies Symposium, PETS 2023, Lausanne, Switzerland, July 10–15, 2023. pp. 40–58. https://doi.org/10.56553/popets-2023-0040
54. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 707–720

A Security Definitions for Bitwise-Independent Encryption

We now formalize the security notions for a bitwise-independent encryption scheme $\mathcal{BE}$, namely S-COA security and CPA security. These are defined through the indistinguishability experiments $\mathsf{Exp}^{\mathsf{S\text{-}COA}}_{\mathcal{A},\mathcal{BE}}(1^\lambda)$ and $\mathsf{Exp}^{\mathsf{CPA}}_{\mathcal{A},\mathcal{BE}}(1^\lambda)$ shown below.

A.1 S-COA Security

Experiment $\mathsf{Exp}^{\mathsf{S\text{-}COA}}_{\mathcal{A},\mathcal{BE}}(1^\lambda)$ (*S-COA Indistinguishability for* $\mathcal{BE}$):

1: The adversary $\mathcal{A}$ is given input 1^λ and outputs two Bloom filters $\mathbb{BF}_0$ and $\mathbb{BF}_1$ of equal length n.
2: A secret key $K \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$ is generated, and a random bit $b \xleftarrow{\$} \{0,1\}$ is chosen.
3: The challenger computes the encrypted Bloom filter

$$\mathbb{EBF} \leftarrow \{\, eb_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K, j, \mathbb{BF}_b[j]) \,\}_{j=0}^{n-1}$$

and sends $\mathbb{EBF}$ to $\mathcal{A}$.
4: The adversary outputs a bit b'.
5: The experiment outputs 1 if $b' = b$, and 0 otherwise.

A bitwise-independent encryption scheme $\mathcal{BE}$ is said to be *S-COA secure* if, for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$,

$$\mathsf{Adv}^{\mathsf{S\text{-}COA}}_{\mathcal{A},\mathcal{BE}}(\lambda) = \left| \Pr[\mathsf{Exp}^{\mathsf{S\text{-}COA}}_{\mathcal{A},\mathcal{BE}}(1^\lambda) = 1] - \frac{1}{2} \right|$$

is negligible in λ.

A.2 CPA Security

Experiment $\mathsf{Exp}^{\mathsf{CPA}}_{\mathcal{A},\mathcal{BE}}(1^\lambda)$ (*CPA Indistinguishability for* $\mathcal{BE}$):

1: A secret key $K \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$ is generated.

2: The adversary $\mathcal{A}$, given input 1^λ and oracle access to $\mathcal{BE}.\mathsf{BEnc}_K(\cdot,\cdot)$, outputs two Bloom filters $\mathbb{BF}_0$ and $\mathbb{BF}_1$ of equal length n.

3: A random bit $b \xleftarrow{\$} \{0,1\}$ is chosen. The challenger computes

$$\mathbb{EBF} \leftarrow \{\, eb_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K, j, \mathbb{BF}_b[j]) \,\}_{j=0}^{n-1}$$

and returns $\mathbb{EBF}$ to $\mathcal{A}$.

4: The adversary continues to have oracle access to $\mathcal{BE}.\mathsf{BEnc}_K(\cdot,\cdot)$ and eventually outputs a bit b'.

5: The experiment outputs 1 if $b' = b$, and 0 otherwise.

A bitwise-independent encryption scheme $\mathcal{BE}$ is said to be *CPA-secure* if, for all PPT adversaries $\mathcal{A}$,

$$\mathsf{Adv}^{\mathsf{CPA}}_{\mathcal{A},\mathcal{BE}}(\lambda) = \left| \Pr[\mathsf{Exp}^{\mathsf{CPA}}_{\mathcal{A},\mathcal{BE}}(1^\lambda) = 1] - \frac{1}{2} \right|$$

is negligible in λ.

B Proof for Lemma 1

Proof. Let $s, n \in \mathbb{N}$, and consider an array $\mathbb{V}$ of size s, where each element of $\mathbb{V}$ is drawn independently and uniformly at random from $[0, n-1]$.

Applying the Chernoff Inequalities. For an arbitrary subset $\mathcal{W} \subseteq [0, n-1]$ of size $|\mathcal{W}| = \omega$, define

$$Y = \sum_{i=0}^{s-1} Y_i, \qquad Y_i = \begin{cases} 1 & \text{if } \mathbb{V}[i] \in \mathcal{W}, \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, for a subset $\mathcal{W}' \subseteq [0, n-1]$ of size $|\mathcal{W}'| = 2\omega$, define

$$Y' = \sum_{i=0}^{s-1} Y'_i, \qquad Y'_i = \begin{cases} 1 & \text{if } \mathbb{V}[i] \in \mathcal{W}', \\ 0 & \text{otherwise.} \end{cases}$$

By construction, $Y = |\mathcal{W} \cap \mathbb{V}|$ and $Y' = |\mathcal{W}' \cap \mathbb{V}|$. Since each element of $\mathbb{V}$ is chosen uniformly at random, the expectations are

$$\mathbb{E}(Y) = \frac{s\omega}{n}, \qquad \mathbb{E}(Y') = \frac{2s\omega}{n}.$$

By Chernoff inequalities [17], for any real c satisfying $\mathbb{E}[Y] \le c \le \mathbb{E}[Y']$, we have:

$$\begin{aligned} \Pr[Y < c] &= \sum_{i=0}^{\lfloor c \rfloor} \binom{s}{i} \left(\tfrac{\omega}{n}\right)^i \left(1 - \tfrac{\omega}{n}\right)^{s-i} \\ &\ge 1 - \sum_{i=\lceil c \rceil}^{s} \binom{s}{i} \left(\tfrac{\omega}{n}\right)^i \left(1 - \tfrac{\omega}{n}\right)^{s-i} \\ &\ge 1 - \exp\left(- \frac{(c - \frac{s\omega}{n})^2}{\frac{s\omega}{n} + c} \right). \end{aligned} \tag{4}$$

and

$$\begin{aligned}\Pr[Y' > c] &= \sum_{i=\lceil c\rceil}^{s} \binom{s}{i} \left(\tfrac{2\omega}{n}\right)^i \left(1-\tfrac{2\omega}{n}\right)^{s-i} \\ &\geq 1 - \sum_{i=0}^{\lfloor c\rfloor} \binom{s}{i} \left(\tfrac{2\omega}{n}\right)^i \left(1-\tfrac{2\omega}{n}\right)^{s-i} \\ &\geq 1 - \exp\left(-\frac{(\frac{2s\omega}{n}-c)^2}{\frac{4s\omega}{n}}\right). \end{aligned} \tag{5}$$

Partitioning the Domain into Subranges. Partition $[0, n-1]$ into contiguous subranges

$$\mathcal{W}_0, \mathcal{W}_1, \ldots, \mathcal{W}_{\lceil n/\omega\rceil - 1},$$

where $\mathcal{W}_i = [i\omega, (i+1)\omega - 1]$ for $0 \leq i \leq \lceil n/\omega\rceil - 2$ and $\mathcal{W}_{\lceil n/\omega\rceil-1} = [(\lceil n/\omega\rceil - 1)\omega, n-1]$. Let

$$\sigma = \begin{cases} \lceil n/\omega\rceil - 3 & \text{if } \omega \nmid n, \\ \lceil n/\omega\rceil - 2 & \text{if } \omega \mid n. \end{cases}$$

Applying inequalities (4) and (5) across all subranges, we obtain

$$\begin{aligned}&\Pr\left[\forall i : |\mathcal{W}_i \cap \mathbb{V}| < c \text{ and } \forall i \leq \sigma : |(\mathcal{W}_i \cup \mathcal{W}_{i+1}) \cap \mathbb{V}| > c\right] \\ &\quad \geq 1 - \frac{n}{\omega}\exp\left(-\frac{(c-\frac{s\omega}{n})^2}{\frac{s\omega}{n}+c}\right) - \frac{n}{\omega}\exp\left(-\frac{(\frac{2s\omega}{n}-c)^2}{\frac{4s\omega}{n}}\right). \end{aligned} \tag{6}$$

Introducing Robustness Parameter. Let $\rho > 0$ be a robustness parameter. If ω and c satisfy

$$\frac{(c-\frac{s\omega}{n})^2}{\frac{s\omega}{n}+c} \;\geq\; \rho - \ln\left(\tfrac{\omega}{2n}\right), \qquad \frac{(\frac{2s\omega}{n}-c)^2}{\frac{4s\omega}{n}} \;\geq\; \rho - \ln\left(\tfrac{\omega}{2n}\right),$$

then the probability in Inequality 6,

$$\Pr\left[\forall i : |\mathcal{W}_i \cap \mathbb{V}| < c \text{ and } \forall i \leq \sigma : |(\mathcal{W}_i \cup \mathcal{W}_{i+1}) \cap \mathbb{V}| > c\right]$$

is guaranteed to be at least $1 - e^{-\rho}$.

Solving for c. Rearranging the first condition gives

$$c^2 - \left(\tfrac{2s\omega}{n} - \rho + \ln\tfrac{\omega}{2n}\right)c + \left(\tfrac{s\omega}{n}\right)^2 - \left(\rho - \ln\tfrac{\omega}{2n}\right)\tfrac{s\omega}{n} \geq 0,$$

which is implied by

$$c \;\geq\; \frac{s\omega}{n} + \tfrac{1}{2}\left(\rho - \ln\tfrac{\omega}{2n}\right) + \tfrac{1}{2}\sqrt{\left(\rho - \ln\tfrac{\omega}{2n}\right)\left(8\tfrac{s\omega}{n} + \rho - \ln\tfrac{\omega}{2n}\right)}. \tag{7}$$

The second condition simplifies to

$$c \;\le\; \frac{2s\omega}{n} - 2\sqrt{\tfrac{s\omega}{n}\Big(\rho - \ln \tfrac{\omega}{2n}\Big)}. \tag{8}$$

For feasibility, inequalities (7) and (8) must both hold. This requires

$$\frac{s\omega}{n} + \tfrac{1}{2}\big(\rho - \ln \tfrac{\omega}{2n}\big) + \tfrac{1}{2}\sqrt{\big(\rho - \ln \tfrac{\omega}{2n}\big)\big(8\tfrac{s\omega}{n} + \rho - \ln \tfrac{\omega}{2n}\big)} \;\le\; \frac{2s\omega}{n} - 2\sqrt{\tfrac{s\omega}{n}\big(\rho - \ln \tfrac{\omega}{2n}\big)}. \tag{9}$$

Assume

$$\frac{s\omega}{n} = \beta\Big(\rho - \ln \tfrac{\omega}{2n}\Big),$$

for some constant β. Substituting into (9), we find that feasibility requires $\beta \ge \gamma$, where $\gamma = \frac{3\sqrt{17}+13}{2}$.

Solving for ω. We now aim to determine a value of ω such that the following inequality is satisfied:

$$\frac{s\omega}{n} \ge \gamma\Big(\rho - \ln \tfrac{\omega}{2n}\Big). \tag{10}$$

which guarantees the existence of a valid c.

To solve Inequality (10), we first subtract $\gamma \ln(2s)$ from both sides and then collect all terms involving ω on the left-hand side, moving all remaining terms to the right-hand side. This yields:

$$\frac{s\omega}{n} + \gamma \ln \frac{s\omega}{n} \ge \gamma\Big(\rho + \ln(2s)\Big). \tag{11}$$

It is straightforward to observe that if

$$\frac{s\omega}{n} \ge \gamma\Big(\rho + \ln(2s)\Big),$$

i.e.,

$$\omega \ge \frac{\gamma n}{s}\Big(\rho + \ln(2s)\Big),$$

then Inequality (11) — and equivalently Inequality (10) — is satisfied. We emphasize that the condition derived above is only a *sufficient* condition for Inequality (10) to hold; smaller values of ω may also satisfy the inequality. For simplicity, we directly set

$$\omega = \frac{\gamma n}{s}\Big(\rho + \ln(2s)\Big),$$

which ensures that $c = \Theta(\rho + \ln s)$.

Finally, we must ensure that $\omega < n$. This requirement is implied by the condition

$$\gamma\big(\rho + \ln(2s)\big) < s.$$

If s is too small to satisfy this condition, we simply set $\omega = n$ and consequently $c = s + 1$.

Conclusion. For arbitrary positive integers n and s, and robustness parameter ρ, if

$$\gamma(\rho + \ln(2s)) < s,$$

then by setting

$$\omega = \left\lceil \frac{\gamma n}{s}\big(\rho + \ln(2s)\big) \right\rceil,$$

$$c = \frac{s\omega}{n} + \frac{1}{2}\Big(\rho - \ln\frac{\omega}{2n}\Big) + \frac{1}{2}\sqrt{\Big(\rho - \ln\frac{\omega}{2n}\Big)\Big(8\frac{s\omega}{n} + \rho - \ln\frac{\omega}{2n}\Big)},$$

inequality (6) holds with probability at least $1 - e^{-\rho}$. If

$$\gamma(\rho + \ln(2s)) \geq s,$$

then by setting $\omega = n$ and $c = s + 1$, inequality (6) holds with probability 1.

C Proof of Theorem 1

We now provide the proof of Theorem 1, which establishes the correctness and obliviousness of the proposed OBD protocol. Note that the correctness and obliviousness of $\mathsf{OSORT}^{\mathcal{SE}}$ directly imply, respectively, the correctness and CPA security of $\mathcal{SE}$. For clarity, however, we treat $\mathsf{OSORT}^{\mathcal{SE}}$ as a stand-alone primitive and analyze the use of $\mathcal{SE}$ in other parts of the protocol separately.

Proof. **Correctness.** Let $\mathsf{OBD}_0^{\mathsf{cor}}$ denote the protocol in Fig. 3. For every input array $\mathbb{V}$ and the corresponding parameter z defined therein, we show

$$\mathsf{output}_{\mathsf{OBD}_0^{\mathsf{cor}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathcal{F}^z_{\mathsf{distri}}(\mathbb{V}),$$

where $\stackrel{c}{\equiv}$ denotes computational indistinguishability. We establish this via a sequence of hybrids.

$\mathsf{OBD}_1^{\mathsf{cor}}$. As shown in Fig. 6, $\mathsf{OBD}_1^{\mathsf{cor}}$ (all lines except the boxed ones) replaces the $\mathcal{SE}$ decryptions of $\mathbb{EV}^\uparrow[s], \ldots, \mathbb{EV}^\uparrow[s+t-1]$ (each an encryption of dum) with the constant output dum. If $\mathcal{SE}$ is correct, then

$$\mathsf{output}_{\mathsf{OBD}_0^{\mathsf{cor}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathsf{output}_{\mathsf{OBD}_1^{\mathsf{cor}}}(1^\lambda, \mathbb{V}).$$

$\mathsf{OBD}_2^{\mathsf{cor}}$. As in Fig. 6, $\mathsf{OBD}_2^{\mathsf{cor}}$ (boxed lines included, underlined lines excluded) replaces every call to $\mathsf{OSORT}^{\mathcal{SE}}(K_e, \cdot)$ with the ideal sorting functionality $\mathcal{F}_{\mathsf{sort}}(\cdot)$. We introduce the following notation: $\mathbb{V}^\uparrow$ is the ideal sorted output of $\mathbb{V}$; $\mathbb{P}$ is the array of $2(s+t)$ plaintext pairs generated during bucket index assignment; $\mathbb{P}^\uparrow$ is the ideal sorted output of $\mathbb{P}$; and $\widehat{\mathbb{V}}$ is obtained from $\mathbb{P}^\uparrow$ by removing invalid pairs and keeping only the second entry of each pair. In $\mathsf{OBD}_2^{\mathsf{cor}}$, the decryption of $\mathbb{EV}^\uparrow[j]$ is replaced with $\mathbb{V}^\uparrow[j]$, and we define $\mathsf{output}'_{\mathsf{OBD}_2^{\mathsf{cor}}}(\mathbb{V})$ by further

$\mathsf{OBD}_1^{\mathsf{cor}}$ (without boxed lines):
$\mathsf{OBD}_2^{\mathsf{cor}}$ (with boxed lines but without underlined lines):

Client:
$\rho \leftarrow \lambda$
Select the integer ω that satisfies Formula 1 in Lemma 1, and value c that satisfies Formula 2.
$z \leftarrow \lceil \frac{n}{\omega} \rceil$, $t \leftarrow \lceil c \rceil$

The client and the server execute $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server obtains the sorted array $\mathbb{EV}^{\uparrow}$.
$\boxed{\mathbb{V}^{\uparrow} \leftarrow \mathcal{F}_{\mathsf{sort}}(\mathbb{V})}$

Client:
$\boxed{\mathbb{P} \leftarrow \text{empty array}}$
$i \leftarrow -1$, $v_{-1} \leftarrow -1$
for $j = 0$ to $(s + t - 1)$ **do**
  if $j < s$ **then**
    Read $\mathbb{EV}^{\uparrow}[j]$ from $\mathbb{EV}^{\uparrow}$
    $v_j \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EV}^{\uparrow}[j])$
    $\boxed{v_j \leftarrow \mathbb{V}^{\uparrow}[j]}$
  else
    $v_j \leftarrow \mathsf{dum}$
  end if
  if $v_{j-1} < (i+1)\cdot\omega$ and $(i+1)\cdot\omega \le v_j < (i+2)\cdot\omega$ and $v_j \neq \mathsf{dum}$ **then**
    $i \leftarrow i + 1$, $\delta_i \leftarrow j$
  else if $v_{j-1} < (i+1)\cdot\omega$ and $v_j \ge (i+2)\cdot\omega$ and $v_j \neq \mathsf{dum}$ **then**
    return
  else if $v_j = \mathsf{dum}$ and δ_{z-1} is absent **then**
    $i \leftarrow z - 1$, $\delta_{z-1} \leftarrow j$
  end if
  if $\big(i \ge 1$ and $(\delta_i - \delta_{i-1}) \ge t\big)$ or $\big(i \ge 2$ and $(\delta_i - \delta_{i-2}) < t\big)$ **then**
    return
  end if
  if $v_j \in \mathcal{W}_0$ **then** ▷ Case ❶
    $p_0 \leftarrow (0, v_j)$, $p_1 \leftarrow (+\infty, v_j)$
  else if $v_j \in \mathcal{W}_i$ and $j < \delta_{i-1} + t$ **then** ▷ Case ❷
    $p_0 \leftarrow (i, v_j)$, $p_1 \leftarrow (i - 1, v_j)$
  else if $v_j \in \mathcal{W}_i$ and $j \ge \delta_{i-1} + t$ **then** ▷ Case ❸
    $p_0 \leftarrow (i, v_j)$, $p_1 \leftarrow (+\infty, v_j)$
  else if $v_j = \mathsf{dum}$ and $j < \delta_{z-2} + t$ **then** ▷ Case ❹
    $p_0 \leftarrow (z - 2, \mathsf{dum})$, $p_1 \leftarrow (z - 1, \mathsf{dum})$
  else if $v_j = \mathsf{dum}$ and $\delta_{z-2} + t \le j < \delta_{z-1} + t$ **then** ▷ Case ❺
    $p_0 \leftarrow (+\infty, \mathsf{dum})$, $p_1 \leftarrow (z - 1, \mathsf{dum})$
  else if $v_j = \mathsf{dum}$ and $j \ge \delta_{z-1} + t$ **then** ▷ Case ❻
    $p_0 \leftarrow (+\infty, \mathsf{dum})$, $p_1 \leftarrow (+\infty, \mathsf{dum})$
  end if
  $ep_0 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, p_0.\mathsf{first}), \mathcal{SE}.\mathsf{Enc}(K_e, p_0.\mathsf{second}))$
  $ep_1 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, p_1.\mathsf{first}), \mathcal{SE}.\mathsf{Enc}(K_e, p_1.\mathsf{second}))$
  Send ep_0 and ep_1 to the server
  $\boxed{\mathbb{P} \leftarrow \mathbb{P} \cup \{\, p_0, p_1\}}$
end for
$\boxed{\mathbb{P}^{\uparrow} \leftarrow \mathcal{F}_{\mathsf{sort}}(\mathbb{P})}$ ▷ Ordered primarily by the first entry of each pair.
$\boxed{\text{Discard the last } (2(s + t) - zt) \text{ pairs from } \mathbb{P}^{\uparrow}}$
$\boxed{\widehat{\mathbb{V}} \leftarrow \mathbb{P} \text{ with all first entries removed}}$

Server:
$\mathbb{EP} \leftarrow$ empty array
while Receive ep_0 and ep_1 **do**
  $\mathbb{EP} \leftarrow \mathbb{EP} \cup \{ep_0, ep_1\}$
end while

The client and the server run $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EP})$, after which the server obtains the sorted array $\mathbb{EP}^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.

Server:
Discard $\mathbb{EP}^{\uparrow}[z \cdot t]$ through the end.
$\widehat{\mathbb{EV}} \leftarrow \mathbb{EP}^{\uparrow}$ with all first entries removed
return $\widehat{\mathbb{EV}}$

$\mathsf{output}'_{\mathsf{OBD}_2^{\mathsf{cor}}}(1^{\lambda}, \mathbb{V})$:
Run the $\mathsf{OBD}_2^{\mathsf{cor}}$ protocol over the input array $\mathbb{V}$, where it keeps the parameter z, t, and the array $\widehat{\mathbb{V}}$.
for $i = 0$ to $z - 1$ **do**
  $\mathcal{B}_i \leftarrow$ empty set
  for $j = i \cdot t$ to $j = (i + 1) \cdot t - 1$ **do**
    if $\widehat{\mathbb{V}}[j] \in \mathcal{W}_i$ **then**
      $\mathcal{B}_i \leftarrow \mathcal{B}_i \cup \{\widehat{\mathbb{V}}[j]\}$
    end if
  end for
end for
return $\{\mathcal{B}_i\}_{i=0}^{z-1}$

Fig. 6. $\mathsf{OBD}_1^{\mathsf{cor}}$ and $\mathsf{OBD}_2^{\mathsf{cor}}$

$\mathbf{OBD}_1^{\mathsf{sec}}$:

Client:

$\rho \leftarrow \lambda$
Select the integer ω that satisfies Formula 1 in Lemma 1, and value c that satisfies Formula 2.
$z \leftarrow \lceil \frac{n}{\omega} \rceil$, $t \leftarrow \lceil c \rceil$

The client and the server execute $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server obtains the sorted array $\mathbb{EV}^{\uparrow}$.
The client sends t dummy ciphertexts, each computed as $\mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum})$, to the server, which appends them to $\mathbb{EV}^{\uparrow}$.

Client:

$i \leftarrow -1$, $v_{-1} \leftarrow -1$
for $j = 0$ to $(s + t - 1)$ **do**
  Read $\mathbb{EV}^{\uparrow}[j]$ from $\mathbb{EV}^{\uparrow}$
  $v_j \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EV}^{\uparrow}[j])$
  if $v_{j-1} < (i+1)\cdot\omega$ and $(i+1)\cdot\omega \le v_j < (i+2)\cdot\omega$ and $v_j \neq \mathsf{dum}$ **then**
    $i \leftarrow i + 1$, $\delta_i \leftarrow j$
  else if $v_{j-1} < (i+1)\cdot\omega$ and $v_j \ge (i+2)\cdot\omega$ and $v_j \neq \mathsf{dum}$ **then**
    return
  else if $v_j = \mathsf{dum}$ and δ_{z-1} is absent **then**
    $i \leftarrow z - 1, \delta_{z-1} \leftarrow j$
  end if
  if $\big(i \ge 1$ and $(\delta_i - \delta_{i-1}) \ge t\big)$ or $\big(i \ge 2$ and $(\delta_i - \delta_{i-2}) < t\big)$ **then**
    return
  end if
  $ep_0 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, 0^l), \mathcal{SE}.\mathsf{Enc}(K_e, 0^l))$
  $ep_1 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, 0^l), \mathcal{SE}.\mathsf{Enc}(K_e, 0^l))$
  Send ep_0 and ep_1 to the server
end for

Server:

$\mathbb{EP} \leftarrow$ empty array
while Receive ep_0 and ep_1 **do**
  $\mathbb{EP} \leftarrow \mathbb{EP} \cup \{ep_0, ep_1\}$
end while

The client and the server run $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EP})$, after which the server obtains the sorted array $\mathbb{EP}^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.

Server:

Discard $\mathbb{EP}^{\uparrow}[z \cdot t]$ through the end.
$\widehat{\mathbb{EV}} \leftarrow \mathbb{EP}^{\uparrow}$ with all first entries removed
return $\widehat{\mathbb{EV}}$

Fig. 7. $\mathbf{OBD}_1^{\mathsf{sec}}$

replacing the decryption of $\widehat{\mathbb{EV}}[j]$ with $\widehat{\mathbb{V}}[j]$. By the correctness of $\mathsf{OSORT}^{\mathcal{SE}}$ (Definition 1),

$$\mathsf{output}_{\mathsf{OBD}_1^{\mathsf{cor}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathsf{output}'_{\mathsf{OBD}_2^{\mathsf{cor}}}(1^\lambda, \mathbb{V}).$$

Comparison with $\mathcal{F}^z_{\mathsf{distri}}$. The only difference between $\mathsf{output}'_{\mathsf{OBD}_2^{\mathsf{cor}}}(1^\lambda, \mathbb{V})$ and $\mathcal{F}^z_{\mathsf{distri}}(\mathbb{V})$ is that the former may abort if a size-bound condition is violated: either $|\mathbb{V} \cap \mathcal{W}_i| \ge t$ for some i, or $|\mathbb{V} \cap (\mathcal{W}_i \cup \mathcal{W}_{i+1})| < t$ for some $i \le z-3$ ($|\mathbb{V} \cap \mathcal{W}_i| < t$ and $|\mathbb{V} \cap (\mathcal{W}_i \cup \mathcal{W}_{i+1})| \ge t$ imply $\mathbb{V} \cap \mathcal{W}_i \neq \emptyset$). Lemma 1 bounds the abort probability by $e^{-\lambda}$. Therefore,

$$\mathsf{output}'_{\mathsf{OBD}_2^{\mathsf{cor}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathcal{F}^z_{\mathsf{distri}}(\mathbb{V}).$$

This completes the correctness proof.

Obliviousness. We now construct a simulator $\mathcal{S}_{\mathsf{obd}}$ such that, for every $\mathbb{V}$ of size s over domain $[0, n-1]$, $\mathcal{S}_{\mathsf{obd}}(n, s)$ outputs an access pattern indistinguishable from that of OBD. Let $\mathsf{OBD}_0^{\mathsf{sec}}$ denote the real protocol. We argue via hybrids.

$\mathsf{OBD}_1^{\mathsf{sec}}$. As in Fig. 7, $\mathsf{OBD}_1^{\mathsf{sec}}$ changes only the ciphertexts in ep_0 and ep_1: assuming each integer is l bits, every ciphertext is replaced with an encryption of 0^l. By the CPA security of $\mathcal{SE}$,

$$\mathsf{addresses}_{\mathsf{OBD}_0^{\mathsf{sec}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathsf{addresses}_{\mathsf{OBD}_1^{\mathsf{sec}}}(1^\lambda, \mathbb{V}).$$

$\mathsf{OBD}_2^{\mathsf{sec}}$ (without boxed lines):
$\mathsf{OBD}_3^{\mathsf{sec}}$ (with boxed lines but without underlined lines):

Client:
$\rho \leftarrow \lambda$
Select the integer ω that satisfies Formula 1 in Lemma 1, and value c that satisfies Formula 2.
$z \leftarrow \lceil \frac{n}{\omega} \rceil$, $t \leftarrow \lceil c \rceil$

The client and the server execute $\underline{\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EV})}$, after which the server obtains the sorted array $\mathbb{EV}^{\uparrow}$.
Run $\mathcal{S}_{\mathsf{osort}}(s)$,
after which the server obtains $\mathbb{EV}^{\uparrow}$.
The client sends t dummy ciphertexts, each computed as $\mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum})$, to the server, which appends them to $\mathbb{EV}^{\uparrow}$.

Client:
for $j = 0$ to $(s + t - 1)$ **do**
  Read $\mathbb{EV}^{\uparrow}[j]$ from $\mathbb{EV}^{\uparrow}$
  $ep_0 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, 0^l), \mathcal{SE}.\mathsf{Enc}(K_e, 0^l))$
  $ep_1 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, 0^l), \mathcal{SE}.\mathsf{Enc}(K_e, 0^l))$
  Send ep_0 and ep_1 to the server
end for

Server:
$\mathbb{EP} \leftarrow$ empty array
while Receive ep_0 and ep_1 **do**
  $\mathbb{EP} \leftarrow \mathbb{EP} \cup \{ep_0, ep_1\}$
end while

The client and the server run $\underline{\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EP})}$, after which the server obtains the sorted array $\mathbb{EP}^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.
Run $\mathcal{S}_{\mathsf{osort}}(2 \cdot (s + t))$,
after which the server obtains $\mathbb{EP}^{\uparrow}$.

Server:
Discard $\mathbb{EP}^{\uparrow}[z \cdot t]$ through the end.
$\widehat{\mathbb{EV}} \leftarrow \widehat{\mathbb{EP}^{\uparrow}}$ with all first entries removed
return $\widehat{\mathbb{EV}}$

Fig. 8. $\mathbf{OBD}_2^{\mathsf{sec}}$ and $\mathbf{OBD}_3^{\mathsf{sec}}$

$\mathsf{OBD}_2^{\mathsf{sec}}$. Relative to $\mathsf{OBD}_1^{\mathsf{sec}}$, $\mathsf{OBD}_2^{\mathsf{sec}}$ omits decryption of $\mathbb{EV}[j]$ and the associated checks (Fig. 8, excluding boxed code). This affects the access pattern only if the size-bound condition is violated, in which case $\mathsf{OBD}_1^{\mathsf{sec}}$ aborts while $\mathsf{OBD}_2^{\mathsf{sec}}$ does not. Since this event occurs with probability at most $e^{-\lambda}$, we obtain

$$\mathsf{addresses}_{\mathsf{OBD}_1^{\mathsf{sec}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathsf{addresses}_{\mathsf{OBD}_2^{\mathsf{sec}}}(1^\lambda, \mathbb{V}).$$

$\mathsf{OBD}_3^{\mathsf{sec}}$. $\mathsf{OBD}_3^{\mathsf{sec}}$ replaces the calls to OSORT with the simulator $\mathcal{S}_{\mathsf{osort}}$, as in Fig. 8. If $\mathcal{S}_{\mathsf{osort}}$ correctly simulates OSORT, then

$$\mathsf{addresses}_{\mathsf{OBD}_2^{\mathsf{sec}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathsf{addresses}_{\mathsf{OBD}_3^{\mathsf{sec}}}(1^\lambda, n, s).$$

Simulator. Finally, the simulator $\mathcal{S}_{\mathsf{obd}}(n, s)$ is obtained as in Fig. 9. The only difference from $\mathsf{OBD}_3^{\mathsf{sec}}$ is that the secret key K_e is freshly generated via $\mathcal{SE}.\mathsf{Gen}(1^\lambda)$. Clearly,

$$\mathsf{addresses}_{\mathsf{OBD}_3^{\mathsf{sec}}}(1^\lambda, \mathbb{V}) \stackrel{c}{\equiv} \mathsf{addresses} \leftarrow \mathcal{S}_{\mathsf{obd}}(n, s).$$

This completes the proof of obliviousness.

D Proof of Theorem 2

Proof. We establish the correctness of OBFI (Variant 1) by introducing a sequence of hybrid games $\mathsf{G}_0^{\mathsf{cor}}$–$\mathsf{G}_4^{\mathsf{cor}}$.

$\mathcal{S}_{\mathsf{obd}}(n, s)$:

$\rho \leftarrow \lambda$
Select the integer ω that satisfies Formula 1 in Lemma 1, and value c that satisfies Formula 2.
$z \leftarrow \lceil \frac{n}{\omega} \rceil$, $t \leftarrow \lceil c \rceil$
$\mathsf{addresses} \leftarrow$ empty array
$K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$

$\mathsf{addresses}_{\mathsf{osort}} \leftarrow \mathcal{S}_{\mathsf{osort}}(s)$,
after which the server obtains $\mathbb{EV}^\uparrow$.
$\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \mathsf{addresses}_{\mathsf{osort}}$

for $i = s$ to $s + t - 1$ **do**
  $ev_i \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum})$
  $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{write}, \mathbb{EV}^\uparrow[i], ev_i)\}$
end for

The server execute: $\mathbb{EP} \leftarrow$ empty array
for $j = 0$ to $(s + t - 1)$ **do**
  $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{read}, \mathbb{EV}^\uparrow[j])\}$
  $ep_0 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, 0^l), \mathcal{SE}.\mathsf{Enc}(K_e, 0^l))$
  $ep_1 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, 0^l), \mathcal{SE}.\mathsf{Enc}(K_e, 0^l))$
  $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{write}, \mathbb{EP}, ep_0)\}$
  $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{write}, \mathbb{EP}, ep_1)\}$
end for

$\mathsf{addresses}'_{\mathsf{osort}} \leftarrow \mathcal{S}_{\mathsf{osort}}(2 \cdot (s + t))$,
after which the server obtains $\mathbb{EP}^\uparrow$.
$\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \mathsf{addresses}'_{\mathsf{osort}}$
$\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{write}, \mathbb{EP}^\uparrow[z \cdot t] - \mathbb{EP}^\uparrow[2 \cdot (s + t) - 1], \mathsf{empty})\}$
The server execute: $\widehat{\mathbb{EV}} \leftarrow$ empty array
$\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{write}, \widehat{\mathbb{EV}}, \mathbb{EP}^\uparrow$ with all first entries removed$)\}$
Output addresses

Fig. 9. The Simulator $\mathcal{S}_{\mathsf{obd}}(n, s)$

OBFISetup in $\mathsf{G}_0^{\mathsf{cor}}$–$\mathsf{G}_4^{\mathsf{cor}}$:

Client:
$K_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$, $K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$
Determine the Bloom filter parameters from M and the target false positive rate: the size n and the number of hash functions d.
Select $\mathcal{H} = \{H_j : \{0,1\}^* \to [0, n-1]\}_{j=0}^{d-1}$
for $i = 0$ to $n - 1$ **do**
  $eb_i = \mathcal{BE}.\mathsf{BEnc}(K_b, i, 0)$
  Send (i, eb_i) to the server
end for
return $\mathcal{K} \leftarrow (K_b, K_e)$, $\mathcal{H}$, and $\mathbb{BF} \leftarrow 0^n$

Server:
$\mathbb{EBF} \leftarrow$ empty array
while Receive (i, eb_i) **do**
  $\mathbb{EBF} \leftarrow eb_i$
end while

Fig. 10. OBFISetup in $\mathsf{G}_0^{\mathsf{cor}}$–$\mathsf{G}_4^{\mathsf{cor}}$

$\mathsf{G}_0^{\mathsf{cor}}$. As illustrated in Fig. 10 and Fig. 11, we define $\mathsf{G}_0^{\mathsf{cor}}$ to be identical to $\mathrm{OBFIC}_{\mathrm{correct}}$, except that in $\mathsf{G}_0^{\mathsf{cor}}$, OBFISetup initializes the bit array $\mathbb{BF}$ as 0^n, and OBFIInsert updates $\mathbb{BF}[j] \leftarrow b'_j$ (the new bit in the Bloom filter) for all $0 \le j \le n-1$. This change does not affect the output of the OBFIInsert protocol, and hence

$$\Pr\big[\mathrm{OBFIC}_{\mathrm{correct}_{\mathcal{A}}}(1^\lambda) = 0\big] \;=\; \Pr[\mathsf{G}_0^{\mathsf{cor}} = 0]\,.$$

$\mathsf{G}_1^{\mathsf{cor}}$. In $\mathsf{G}_1^{\mathsf{cor}}$, each hash function in $\mathcal{H}$ is modeled as a random oracle (see Fig. 11). For every $0 \le j \le d-1$, a table h_j is maintained to program the oracle H_j. On query x, if $h_j[x]$ has already been defined, the stored value is returned; otherwise, a value is sampled uniformly at random from $[0, n-1]$, assigned to $h_j[x]$, and returned. Since each hash function in $\mathcal{H}$ is modeled as a random oracle, we obtain

$$\Pr[\mathsf{G}_0^{\mathsf{cor}} = 0] \;=\; \Pr[\mathsf{G}_1^{\mathsf{cor}} = 0]\,.$$

OBFIInsert in $\mathsf{G}_0^{\mathsf{cor}}$ (without boxed lines), $\mathsf{G}_1^{\mathsf{cor}}$ (without boxed lines), and $\mathsf{G}_2^{\mathsf{cor}}$ (with boxed lines but without underlined lines):

$m \leftarrow |\mathbb{EX}|$
Client:
$(K_b, K_e) \leftarrow \mathcal{K}$
for $i = 0$ to $m - 1$ **do**
  Read $\mathbb{EX}[i]$ from $\mathbb{EX}$
  $x_i \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EX}[i])$
  $\boxed{x_i \leftarrow \mathbb{X}[i]}$
  for $j = 0$ to d-1 **do**
    $v \leftarrow H_j(x_i)$
    $ev \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, v)$
    Send ev to the server
  end for
end for

Server:
$\mathbb{EV} \leftarrow$ empty array
while Receive ev **do**
  $\mathbb{EV} \leftarrow \mathbb{EV} \cup \{ev\}$
end while

The client and the server execute $\mathbf{OBD}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server receives $\widehat{\mathbb{EV}} = \{\widehat{\mathcal{B}}_0, \cdots, \widehat{\mathcal{B}}_{z-1}\}$.
The client keeps the values $\omega = \lceil n/z \rceil$ and the bucket capacity t.

Client:
if $\mathcal{BE}$ is S-COA secure **then**
  $K_b' \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$
end if
$i \leftarrow 0$
for $i = 0$ to $z - 1$ **do**
  $\mathcal{B}_i \leftarrow$ empty set
  Read $\widehat{\mathcal{B}}_i$ from $\widehat{\mathbb{EV}}$
  for each $ev \in \widehat{\mathcal{B}}_i$ **do**
    $v \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, ev)$
    if $v \in \mathcal{W}_i$ **then**
      $\mathcal{B}_i \leftarrow \mathcal{B}_i \cup \{v\}$
    end if
  end for
  for each $j \in \mathcal{W}_i$ **do**
    Read $\mathbb{EBF}[j]$ from $\mathbb{EBF}$
    $b_j \leftarrow \mathcal{BE}.\mathsf{BDec}(K_b, \mathbb{EBF}[j])$
    if $b_j = 1$ or $j \in \mathcal{B}_i$ **then**
      $b_j' \leftarrow 1$
    else
      $b_j' \leftarrow 0$
    end if
    $\mathbb{BF}[j] \leftarrow b_j'$
    if $\mathcal{BE}$ is CPA secure **then**
      $eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, b_j')$
    else if $\mathcal{BE}$ is S-COA secure **then**
      $eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b', b_j')$
    end if
    Send (j, eb_j') to the server
    $j \leftarrow j + 1$
  end for
end for
if $\mathcal{BE}$ is CPA secure **then**
  return K_b and $\mathbb{BF}$
else if $\mathcal{BE}$ is S-COA secure **then**
  Update K_b in $\mathcal{K}$ to K_b'
  return K_b' and $\mathbb{BF}$
end if

Server:
while Receive (j, eb_j') **do**
  Update $\mathbb{EBF}[j]$ to eb_j'
end while
return $\mathbb{EBF}$

Random oracle $H_j(x)$ in $\mathsf{G}_1^{\mathsf{cor}}$–$\mathsf{G}_4^{\mathsf{cor}}$
if $h_j[x]$ exists **then**
  $v \leftarrow h_j[x]$
else
  $v \xleftarrow{\$} [0, n-1]$, $h_j[x] \leftarrow v$
end if
return v

Fig. 11. OBFIInsert in $\mathsf{G}_0^{\mathsf{cor}}$– $\mathsf{G}_2^{\mathsf{cor}}$, with Random Oracles $\{H_j\}_{j=0}^{d-1}$ in $\mathsf{G}_1^{\mathsf{cor}}$– $\mathsf{G}_4^{\mathsf{cor}}$

OBFIInsert in $\mathsf{G}_3^{\mathsf{cor}}$ (without boxed lines) and $\mathsf{G}_4^{\mathsf{cor}}$(with boxed lines but without underlined lines):

$m \leftarrow |\mathbb{EX}|$
Client:
$\mathbb{V} \leftarrow$ empty array
$(K_b, K_e) \leftarrow \mathcal{K}$
$i \leftarrow 0$
for $i = 0$ to $m - 1$ **do**
  $x_i \leftarrow \mathbb{X}[i]$
  for $j = 0$ to d-1 **do**
    $v \leftarrow H_j(x_i)$
    $\mathbb{V} \leftarrow \mathbb{V} \cup \{v\}$
  end for
end for

Select ω according to Lemma 1. ▷ using n and $s \leftarrow |\mathbb{V}|$ as inputs.
$z \leftarrow \lceil \frac{n}{\omega} \rceil$
$\{\mathbb{V} \cap \mathcal{W}_i\}_{i=0}^{z-1} \leftarrow \mathcal{F}_{\mathsf{distri}}^{z}(\mathbb{V})$
▷ $\mathcal{W}_i \leftarrow [i \cdot \omega, (i+1) \cdot \omega - 1]$ for all $0 \le i \le z-2$, and $\mathcal{W}_{z-1} \leftarrow [(z-1) \cdot \omega, n-1]$.

Client:
if $\mathcal{BE}$ is S-COA secure **then**
  $K_b' \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$
end if
$i \leftarrow 0$
for $i = 0$ to $z - 1$ **do**
  $\mathcal{B}_i \leftarrow \mathbb{V} \cap \mathcal{W}_i$
  for each $j \in \mathcal{W}_i$ **do**
    Read $\mathbb{EBF}[j]$ from $\mathbb{EBF}$
    $\underline{b_j \leftarrow \mathcal{BE}.\mathsf{BDec}(K_b, \mathbb{EBF}[j])}$
    $\boxed{b_j \leftarrow \mathbb{BF}[j]}$
    if $b_j = 1$ or $j \in \mathcal{B}_i$ **then**
      $b_j' \leftarrow 1$
    else
      $b_j' \leftarrow 0$
    end if
    $\mathbb{BF}[j] \leftarrow b_j'$
    if $\mathcal{BE}$ is CPA secure **then**
      $\underline{eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, b_j')}$
    else if $\mathcal{BE}$ is S-COA secure **then**
      $\underline{eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b', b_j')}$
    end if
    Send $\underline{(j, eb_j')}$ to the server
    $j \leftarrow j + 1$
  end for
end for
if $\mathcal{BE}$ is CPA secure **then**
  return $\underline{K_b}$ and $\mathbb{BF}$
else if $\mathcal{BE}$ is S-COA secure **then**
  Update $\underline{K_b}$ in $\mathcal{K}$ to $\underline{K_b'}$
  return $\underline{K_b'}$ and $\mathbb{BF}$
end if

Server:
while Receive $\underline{(j, eb_j')}$ **do**
  Update $\underline{\mathbb{EBF}[j]}$ to $\underline{eb_j'}$
end while
return $\underline{\mathbb{EBF}}$

Define $\mathbf{output}_i$ in $\mathsf{G}_4^{\mathsf{cor}}$ as $\mathbb{BF}$, where $\mathbb{BF}$ is the bit array returned by the client after sequentially executing **OBFIInsert**$(\mathbb{X})$ within $\mathsf{G}_4^{\mathsf{cor}}$ for $\mathbb{X} = \mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_i$.

Fig. 12. OBFIInsert in $\mathsf{G}_3^{\mathsf{cor}}$ and $\mathsf{G}_4^{\mathsf{cor}}$, with output_i in $\mathsf{G}_4^{\mathsf{cor}}$

$\mathsf{G}^{\mathsf{cor}}_2$**.** In $\mathsf{G}^{\mathsf{cor}}_2$ (see Fig. 11), the element x_i is taken directly from the plaintext batch $\mathbb{X}$, rather than being obtained by decrypting $\mathbb{EX}[i]$. The difference between $\mathsf{G}^{\mathsf{cor}}_2$ and $\mathsf{G}^{\mathsf{cor}}_1$ can be reduced to the correctness of $\mathcal{SE}$. Let $\mathsf{Adv}^{\mathsf{cor}}_{\mathcal{SE}}$ denote the advantage of a PPT adversary in breaking the correctness of $\mathcal{SE}$. Then

$$\Pr[\mathsf{G}^{\mathsf{cor}}_1 = 0] - \Pr[\mathsf{G}^{\mathsf{cor}}_2 = 0] \;\leq\; \mathsf{Adv}^{\mathsf{cor}}_{\mathcal{SE}}.$$

$\mathsf{G}^{\mathsf{cor}}_3$**.** In $\mathsf{G}^{\mathsf{cor}}_3$ (Fig. 12), the output of the real OBD execution on $\mathbb{EV}$ is replaced by the output of the ideal bucket distribution functionality $\mathcal{F}^{z}_{\mathsf{distri}}(\mathbb{V})$, where $\mathbb{V}$ is the array of computed hash values and $z \leftarrow \lceil n/\omega \rceil$, with ω computed exactly as in the OBD protocol (*i.e.,* Lemma 1). For each $0 \leq i \leq z-1$, instead of constructing bucket $\mathcal{B}_i$ by decrypting the output of the real OBD and discarding elements outside $\mathcal{W}_i$, $\mathsf{G}^{\mathsf{cor}}_3$ directly sets $\mathcal{B}_i \leftarrow \mathbb{V} \cap \mathcal{W}_i$.

Since each batch $\mathbb{X}$ consists of distinct elements, $\mathbb{V}$ can be regarded as an array of independent uniform samples from $[0, n-1]$. Let $\mathsf{Adv}^{\mathsf{cor}}_{\mathsf{OBD}}$ denote the advantage of breaking the correctness of OBD. It follows that

$$\Pr[\mathsf{G}^{\mathsf{cor}}_2 = 0] - \Pr[\mathsf{G}^{\mathsf{cor}}_3 = 0] \;\leq\; \mathsf{Adv}^{\mathsf{cor}}_{\mathsf{OBD}}.$$

$\mathsf{G}^{\mathsf{cor}}_4$**.** In $\mathsf{G}^{\mathsf{cor}}_4$ (Fig. 12), instead of decrypting $\mathbb{EBF}[j]$ to obtain b_j, we directly set $b_j \leftarrow \mathbb{BF}[j]$ for all $0 \leq j \leq n-1$. Moreover, we define output_i in $\mathsf{G}^{\mathsf{cor}}_4$ as $\mathbb{BF}$ obtained after running OBFIInsert on $\mathbb{X}_1, \ldots, \mathbb{X}_i$ adaptively chosen by the adversary. Recall that in $\mathsf{G}^{\mathsf{cor}}_3$, output_i is instead defined as the $\mathcal{BE}$-decryption of $\mathbb{EBF}$. Let $\mathsf{Adv}^{\mathsf{cor}}_{\mathcal{BE}}$ denote the advantage of a PPT adversary in breaking the correctness of $\mathcal{BE}$. We then have

$$\Pr[\mathsf{G}^{\mathsf{cor}}_3 = 0] - \Pr[\mathsf{G}^{\mathsf{cor}}_4 = 0] \;\leq\; \mathsf{Adv}^{\mathsf{cor}}_{\mathcal{BE}}.$$

Finally, for every $i = 1$ to $poly(\lambda)$, the output_i in $\mathsf{G}^{\mathsf{cor}}_4$ is exactly

$$\mathcal{F}_{\mathsf{bfins}}(\mathbb{X}_i, \mathcal{H}, \mathbb{BF}_{i-1}),$$

where $\mathbb{BF}_{i-1} \leftarrow \mathcal{F}_{\mathsf{bfins}}(\mathbb{X}_{i-1}, \mathcal{H}, \mathbb{BF}_{i-2})$ for $i > 1$, and $\mathbb{BF}_0 \leftarrow 0^n$. Thus,

$$\Pr[\mathsf{G}^{\mathsf{cor}}_4 = 0] = 0.$$

Conclusion. Combining the above transitions, if each hash function in $\mathcal{H}$ is modeled as a random oracle, then

$$\Pr\big[\mathrm{OBFIC}_{\mathrm{correct}_{\mathcal{A}}}(1^\lambda) = 0\big] \;\leq\; \mathsf{Adv}^{\mathsf{cor}}_{\mathcal{SE}} + \mathsf{Adv}^{\mathsf{cor}}_{\mathsf{OBD}} + \mathsf{Adv}^{\mathsf{cor}}_{\mathcal{BE}}.$$

This completes the correctness proof for OBFI (Variant 1).

E Proof of Theorem 3

Proof. We prove the security of OBFI (Variant 1) via a sequence of hybrid games $\mathsf{G}^{\mathsf{sec}}_0$–$\mathsf{G}^{\mathsf{sec}}_4$.

OBFISetup in $\mathsf{G}_1^{\mathsf{sec}}$–$\mathsf{G}_4^{\mathsf{sec}}$:

Client:

$K_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$, $K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$
Determine the Bloom filter parameters from M and the target false positive rate: the size n and the number of hash functions d.
for $i = 0$ to $n-1$ **do**
    $eb_i = \mathcal{BE}.\mathsf{BEnc}(K_b, i, 0)$
    Send (i, eb_i) to the server
end for
for $j = 0$ to $d-1$ **do**
    $h_j \leftarrow$ empty table
end for
return $\mathcal{K} \leftarrow (K_b, K_e)$ and $\{h_j\}_{j=0}^{d-1}$

Server:

$\mathbb{EBF} \leftarrow$ empty array
while Receive (i, eb_i) **do**
    $\mathbb{EBF} \leftarrow eb_i$
end while

Fig. 13. OBFISetup in $\mathsf{G}_1^{\mathsf{sec}}$–$\mathsf{G}_4^{\mathsf{sec}}$

OBFIInsert in $\mathsf{G}_1^{\mathsf{sec}}$ (without boxed lines) and $\mathsf{G}_2^{\mathsf{sec}}$ (with boxed lines but without underlined lines):

$m \leftarrow |\mathbb{EX}|$

Client:

$(K_b, K_e) \leftarrow \mathcal{K}$, $\{H_i\}_{i=0}^{d-1} \leftarrow \mathcal{H}$
for $i = 0$ to $m-1$ **do**
    Read $\mathbb{EX}[i]$ from $\mathbb{EX}$
    $x_i \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EX}[i])$
    for $j = 0$ to d-1 **do**
        $v \leftarrow H_j(x_i)$
        if $h_j[x_i]$ exists **then**
            $v \leftarrow h_j[x_i]$
        else
            $v \xleftarrow{\$} [0, n-1]$
            $h_j[x] \leftarrow v$
        end if
        $ev \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, v)$
        Send ev to the server
    end for
end for

Server:

$\mathbb{EV} \leftarrow$ empty array
while Receive ev **do**
    $\mathbb{EV} \leftarrow \mathbb{EV} \cup \{ev\}$
end while

The client and the server execute $\mathbf{OBD}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server receives $\widehat{\mathbb{EV}} = \{\widehat{\mathcal{B}}_0, \cdots, \widehat{\mathcal{B}}_{z-1}\}$.
The client keeps the values $\omega = \lceil n/z \rceil$ and the bucket capacity t.

Client:

if $\mathcal{BE}$ is S-COA secure **then**
    $K_b' \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$
end if
for $i = 0$ to $z-1$ **do**
    $\mathcal{B}_i \leftarrow$ empty set
    Read $\widehat{\mathcal{B}}_i$ from $\widehat{\mathbb{EV}}$
    for each $ev \in \widehat{\mathcal{B}}_i$ **do**
        $v \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, ev)$
        if $v \in \mathcal{W}_i$ **then**
            $\mathcal{B}_i \leftarrow \mathcal{B}_i \cup \{v\}$
        end if
    end for
    for each $j \in \mathcal{W}_i$ **do**
        Read $\mathbb{EBF}[j]$ from $\mathbb{EBF}$
        $b_j \leftarrow \mathcal{BE}.\mathsf{BDec}(K_b, \mathbb{EBF}[j])$
        if $b_j = 1$ or $j \in \mathcal{B}_i$ **then**
            $b_j' \leftarrow 1$
        else
            $b_j' \leftarrow 0$
        end if
        if $\mathcal{BE}$ is CPA secure **then**
            $eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, b_j')$
            $\boxed{eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, 0)}$
        else if $\mathcal{BE}$ is S-COA secure **then**
            $eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b', b_j')$
            $\boxed{eb_j' \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b', 0)}$
        end if
        Send (j, eb_j') to the server
        $j \leftarrow j + 1$
    end for
end for
if $\mathcal{BE}$ is CPA secure **then**
    return K_b
else if $\mathcal{BE}$ is S-COA secure **then**
    Update K_b in $\mathcal{K}$ to K_b'
    return K_b'
end if

Server:

while Receive (j, eb_j') **do**
    Update $\mathbb{EBF}[j]$ to eb_j'
end while
return $\mathbb{EBF}$

Fig. 14. OBFIInsert in $\mathsf{G}_1^{\mathsf{sec}}$ and $\mathsf{G}_2^{\mathsf{sec}}$

OBFIInsert in $\mathsf{G}_3^{\mathsf{sec}}$ (without boxed lines) and $\mathsf{G}_4^{\mathsf{sec}}$ (with boxed lines but without underlined lines):

In $\mathsf{G}_4^{\mathsf{sec}}$, for a batch $\mathbb{X}$, the encrypted batch $\mathbb{EX}$ is replaced by $\mathbb{EX} \leftarrow \{ ex_i \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, 0^{l_x}) \}_{i=0}^{|\mathbb{X}|-1}$, where l_x denotes the bit-length of an element in $\mathbb{X}$.

$m \leftarrow |\mathbb{EX}|$

Client:
1: $(K_b, K_e) \leftarrow \mathcal{K}$, $\{H_i\}_{i=0}^{d-1} \leftarrow \mathcal{H}$
2: **for** $i = 0$ to $m - 1$ **do**
3: Read $\mathbb{EX}[i]$ from $\mathbb{EX}$
4: $x_i \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EX}[i])$
5: **for** $j = 0$ to d-1 **do**
6: **if** $h_j[x_i]$ exists **then**
7: $v \leftarrow h_j[x_i]$
8: **else**
9: $v \xleftarrow{\$} [0, n-1]$
10: $h_j[x] \leftarrow v$
11: **end if**
12: $ev \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, v)$
13: $ev \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, 0^l)$
14: Send ev to the server
15: **end for**
16: **end for**

Server:
17: $\mathbb{EV} \leftarrow$ empty array
18: **while** Receive ev **do**
19: $\mathbb{EV} \leftarrow \mathbb{EV} \cup \{ev\}$
20: **end while**
21: Run $\mathcal{S}_{\mathsf{obd}}(m \cdot d)$, after which the server receives $\widehat{\mathbb{EV}} = \{\widehat{\mathcal{B}}_0, \cdots, \widehat{\mathcal{B}}_{z-1}\}$.

Client:
22: **if** $\mathcal{BE}$ is S-COA secure **then**
23: $K'_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$
24: **end if**
25: **for** $i = 0$ to $z - 1$ **do**
26: Read $\widehat{\mathcal{B}}_i$ from $\widehat{\mathbb{EV}}$
27: **for** each $j \in \mathcal{W}_i$ **do**
28: Read $\mathbb{EBF}[j]$ from $\mathbb{EBF}$
29: **if** $\mathcal{BE}$ is CPA secure **then**
30: $eb'_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, 0)$
31: **else if** $\mathcal{BE}$ is S-COA secure **then**
32: $eb'_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K'_b, 0)$
33: **end if**
34: Send (j, eb'_j) to the server
35: $j \leftarrow j + 1$
36: **end for**
37: **end for**
38: **if** $\mathcal{BE}$ is CPA secure **then**
39: **return** K_b
40: **else if** $\mathcal{BE}$ is S-COA secure **then**
41: Update K_b in $\mathcal{K}$ to K'_b
42: **return** K'_b
43: **end if**

Server:
44: **while** Receive (j, eb'_j) **do**
45: Update $\mathbb{EBF}[j]$ to eb'_j
46: **end while**
47: **return** $\mathbb{EBF}$

Fig. 15. OBFIInsert in $\mathsf{G}_3^{\mathsf{sec}}$ and $\mathsf{G}_4^{\mathsf{sec}}$

$\mathsf{G}_0^{\mathsf{sec}}$. $\mathsf{G}_0^{\mathsf{sec}}$ is defined exactly as the real experiment $\text{OBFIR}_{\text{EAL}\mathcal{A}}(1^\lambda)$. Thus,

$$\Pr[\text{OBFIR}_{\text{EAL}\mathcal{A}}(1^\lambda) = 1] \;=\; \Pr[\mathsf{G}_0^{\mathsf{sec}} = 1].$$

$\mathsf{G}_1^{\mathsf{sec}}$. As shown in Fig. 13 and Fig. 14, OBFISetup in $\mathsf{G}_1^{\mathsf{sec}}$ introduces d tables $h_0, h_1, \ldots, h_{d-1}$. In OBFIInsert, for each $0 \le j \le d-1$: on a fresh query x, the hash function H_j is replaced by outputting a uniformly random value $v \in [0, n-1]$ and sets $h_j[x] \leftarrow v$; on a repeated query x, it outputs $h_j[x]$. Under the assumption that each $H_j \in \mathcal{H}$ is a uniformly random function, it follows that

$$\Pr[\mathsf{G}_0^{\mathsf{sec}} = 1] \;=\; \Pr[\mathsf{G}_1^{\mathsf{sec}} = 1].$$

$\mathsf{G}_2^{\mathsf{sec}}$. In $\mathsf{G}_2^{\mathsf{sec}}$, as illustrated in Fig. 14, each ciphertext eb'_j corresponding to a Bloom filter bit is generated by encrypting 0 under $\mathcal{BE}$. The encryption key is either the original secret key K_b (if $\mathcal{BE}$ is CPA secure) or a freshly derived key K'_b (if $\mathcal{BE}$ is only S-COA secure). Let $\mathsf{Adv}_{\mathcal{BE}}^{\mathsf{sec}}$ denote the advantage of any PPT

$\mathcal{S}(M)$:

$\mathsf{addresses}_0 \leftarrow$ empty array
$K_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$, $K_e \leftarrow \mathcal{SE}.\mathsf{Gen}(1^\lambda)$
Determine the Bloom filter parameters from M and the target false positive rate: the size n and the number of hash functions d.
The server execute: $\mathbb{EBF} \leftarrow$ empty array
for $i = 0$ to $n - 1$ **do**
    $eb_i = \mathcal{BE}.\mathsf{BEnc}(K_b, i, 0)$
    $\mathsf{addresses}_0 \leftarrow \mathsf{addresses}_0 \cup \{(\mathsf{write}, \mathbb{EBF}[i], eb_i)\}$
end for
$\mathcal{S}$ keeps (K_b, K_e, n, d) and outputs $\mathsf{addresses}_0$

$\mathcal{S}(|\mathbb{X}|)$: ▷ The server store $\mathbb{EX} \leftarrow \{ ex_i \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, 0^{l_x}) \}_{i=0}^{|\mathbb{X}|-1}$ and $\mathbb{EBF}$.

$\mathsf{addresses} \leftarrow$ empty array
The server execute: $\mathbb{EV} \leftarrow$ empty array
for $i = 0$ to $|\mathbb{X}| - 1$ **do**
    $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{read}, \mathbb{EX}[i])\}$
    for $j = 0$ to d-1 **do**
        $ev \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, 0^l)$
        $\mathsf{addresses} \cup \{(\mathsf{write}, \mathbb{EV}, ev)\}$
    end for
end for
$\mathsf{addresses}_{\mathsf{obd}} \leftarrow \mathcal{S}_{\mathsf{obd}}(m \cdot d)$, after which the server receives $\widehat{\mathbb{EV}} = \{\widehat{\mathcal{B}}_0, \cdots, \widehat{\mathcal{B}}_{z-1}\}$.
$\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \mathsf{addresses}_{\mathsf{obd}}$

if $\mathcal{BE}$ is S-COA secure **then**
    $K'_b \leftarrow \mathcal{BE}.\mathsf{BGen}(1^\lambda)$
end if
for $i = 0$ to $z - 1$ **do**
    $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{read}, \widehat{\mathcal{B}}_i)\}$
    for each $j \in \mathcal{W}_i$ **do**
    ▷ $\mathcal{W}_i$ is determined directly from n and z.
        $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{read}, \mathbb{EBF}[j])\}$
        if $\mathcal{BE}$ is CPA secure **then**
            $eb'_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K_b, 0)$
        else if $\mathcal{BE}$ is S-COA secure **then**
            $eb'_j \leftarrow \mathcal{BE}.\mathsf{BEnc}(K'_b, 0)$
        end if
        $\mathsf{addresses} \leftarrow \mathsf{addresses} \cup \{(\mathsf{write}, \mathbb{EBF}[j]), eb'_j)\}$
        $j \leftarrow j + 1$
    end for
end for
if $\mathcal{BE}$ is CPA secure **then**
    return K_b
else if $\mathcal{BE}$ is S-COA secure **then**
    $K_b \leftarrow K'_b$
end if
$\mathcal{S}$ keeps (K_b, K_e, n, d) and outputs $\mathsf{addresses}$

Fig. 16. The Simulator $\mathcal{S}$ for OBFI (Variant 1)

adversary in violating the corresponding security notion of $\mathcal{BE}$. Then,

$$\left| \Pr[\mathsf{G}_1^{\mathsf{sec}} = 1] - \Pr[\mathsf{G}_2^{\mathsf{sec}} = 1] \right| \leq \mathsf{Adv}_{\mathcal{BE}}^{\mathsf{sec}}.$$

$\mathsf{G}_3^{\mathsf{sec}}$. In $\mathsf{G}_3^{\mathsf{sec}}$, as shown in Fig. 15, the real execution of OBD on $\mathbb{EV}$ is replaced with the simulated execution $\mathcal{S}_{\mathsf{obd}}(m \cdot d)$, where $m \cdot d$ is the size of $\mathbb{EV}$. Let $\mathsf{Adv}_{\mathsf{OBD}}^{\mathsf{obli}}$ denote the advantage of any PPT adversary in distinguishing the real OBD execution from its simulation. We have

$$\left| \Pr[\mathsf{G}_2^{\mathsf{sec}} = 1] - \Pr[\mathsf{G}_3^{\mathsf{sec}} = 1] \right| \leq \mathsf{Adv}_{\mathsf{OBD}}^{\mathsf{obli}}.$$

$\mathsf{G}_4^{\mathsf{sec}}$. As shown in Fig. 15, in $\mathsf{G}_4^{\mathsf{sec}}$: each $\mathcal{SE}$ encryption of a hash value v is replaced with an encryption of 0^l, where l is the bit length of v (e.g., $\log n$); simultaneously, the encrypted batch is replaced with

$$\mathbb{EX} \leftarrow \{ ex_i \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, 0^{l_x}) \}_{i=0}^{|\mathbb{X}|-1},$$

where l_x is the bit length of elements in $\mathbb{X}$. Since v is no longer required, the decryption of $\mathbb{EX}[i]$ and the subsequent computation of hash values can be omitted. Let $\mathsf{Adv}_{\mathcal{SE}}^{\mathsf{CPA}}$ denote the advantage of any PPT adversary in breaking the

CPA-security of $\mathcal{SE}$. Then,

$$\left|\Pr[\mathsf{G}_3^{\mathsf{sec}} = 1] - \Pr[\mathsf{G}_4^{\mathsf{sec}} = 1]\right| \;\leq\; \mathsf{Adv}^{\mathsf{CPA}}_{\mathcal{SE}}.$$

Finally, note that $\mathsf{G}_4^{\mathsf{sec}}$ is identical to the ideal experiment $\text{OBFII}_{\text{IDEAL}_{\mathcal{A},\mathcal{S}}}(1^\lambda)$: OBFISetup depends only on M, and OBFIInsert depends only on $|\mathbb{X}|$. For clarity, Fig. 16 defines the simulator $\mathcal{S}$ so that $\mathcal{S}(M)$ and $\mathcal{S}(|\mathbb{X}|)$ reproduce the access patterns of OBFISetup and OBFIInsert in $\mathsf{G}_4^{\mathsf{sec}}$. Therefore,

$$\Pr[\mathsf{G}_4^{\mathsf{sec}} = 1] = \Pr[\text{OBFII}_{\text{IDEAL}_{\mathcal{A},\mathcal{S}}}(1^\lambda) = 1].$$

Conclusion. Putting the above together, if each hash function in $\mathcal{H}$ is modeled as a uniformly random function, then

$$\left|\Pr[\text{OBFIR}_{\text{EAL}_{\mathcal{A}}}(1^\lambda) = 1] - \Pr[\text{OBFII}_{\text{IDEAL}_{\mathcal{A},\mathcal{S}}}(1^\lambda) = 1]\right| \;\leq$$
$$\mathsf{Adv}^{\mathsf{sec}}_{\mathcal{BE}} + \mathsf{Adv}^{\mathsf{obli}}_{\mathsf{OBD}} + \mathsf{Adv}^{\mathsf{CPA}}_{\mathcal{SE}}.$$

F An Adapted WS12 Construction

Fig. 17 presents an adapted version of the Oblivious Bloom Filter Creation protocol proposed initially by Williams and Sion [50]. The client inputs a secret key K_e for $\mathcal{SE}$ and the parameters of the bucket Bloom filter (size n' and hash functions $\mathcal{H}'$). The server inputs the bucket $\widehat{\mathcal{B}}_i$.

The client first chooses a small integer parameter α (*e.g.,* $\alpha = 32$), which determines the required client storage. It then scans each element in the bucket, computes its hash values, and labels each as real if it belongs to $\mathcal{W}_i$, or as dum otherwise. Each tagged hash value is encrypted and sent to the server. Next, the client generates $\lceil n'/\alpha \rceil$ *segment delimiters* that partition $[0, n'-1]$ into $\lceil n'/\alpha \rceil$ contiguous segments. Each delimiter is chosen as a fractional value lying between two adjacent segments, *e.g.,*, $\alpha - \frac{1}{2}, 2\alpha - \frac{1}{2}, \ldots$, continuing until the final delimiter exceeds $n' - 1$. Every delimiter is tagged as delim, encrypted, and transmitted to the server.

An OSORT is then executed over the encrypted array of hash values and delimiters. The client initializes an empty set Δ and scans the sorted array. For each hash value: if tagged real, it is inserted into Δ; if tagged dum, it is discarded. In either case, the client produces a dummy bitstring of length α, tags it with $+\infty$ (any value exceeding $\lceil n'/\alpha \rceil$), encrypts it, and sends it to the server. When a delimiter τ is encountered, the client constructs a bitstring corresponding to the segment of $\mathcal{BBF}$ spanning indices $\lfloor \tau/\alpha \rfloor \cdot \alpha$ through $\lfloor \tau \rfloor$, with bits set according to membership in Δ. This bitstring is tagged with $\lfloor \tau/\alpha \rfloor$, encrypted, and sent to the server, after which Δ is cleared.

A second OSORT is then performed over the encrypted bitstrings, ordered by their tags. Dummy strings are moved to the end and subsequently discarded. The client finally encrypts each bit of the remaining bitstrings using F_1 and F_2, sending the resulting ciphertexts to the server, which outputs the encrypted bucket Bloom filter.

The client inputs a secret key K_e for $\mathcal{SE}$, along with the parameters for the bucket Bloom filter: the size n', and a set of d' hash functions $\mathcal{H}' = \{H'_i : \{0,1\}^* \to [0, n' - 1]\}_{i=0}^{d'-1}$. The server inputs a bucket $\widehat{\mathcal{B}}_i$.

Client:
Select a small integer α, *e.g.,* $\alpha = 32$
for $j = 0$ to $(t-1)$ **do**
  Read $\widehat{\mathcal{B}}_i[j]$ from bucket $\widehat{\mathcal{B}}_i$
  $v \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \widehat{\mathcal{B}}_i[j])$
  for $k = 0$ to $d' - 1$ **do**
    $h \leftarrow H'_k(v)$, $eh \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, h)$
    if $v \in \mathcal{W}_i$ **then**
      $ep \leftarrow (eh, \mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{real}))$
    else
      $ep \leftarrow (eh, \mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum}))$
    end if
    Send ep to the server
  end for
end for
$\tau \leftarrow \alpha - \frac{1}{2}$
  ▷ τ can be initialized to an arbitrary non-integer value between $\alpha - 1$ and α.
while $\tau < n' + \alpha - 1$ **do**
  $ep \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, \tau), \mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{delim}))$
  Send ep to the server, $\tau \leftarrow \tau + \alpha$
end while

Server:
$\widehat{\mathcal{P}}_1 \leftarrow$ empty array
while Receive ep **do**
  $\widehat{\mathcal{P}}_1 \leftarrow \widehat{\mathcal{P}}_1 \cup \{ep\}$
end while

The client and the server execute $\mathsf{OS\text{-}ORT}(k_e; \widehat{\mathcal{P}}_1)$, after which the server receives the encrypted array $\widehat{\mathcal{P}}_1^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.

Client:
$\Delta \leftarrow$ empty set
for $j = 0$ to $|\widehat{\mathcal{P}}_1^{\uparrow}| - 1$ **do**
  $p \leftarrow (\mathcal{SE}.\mathsf{Dec}(K_e, \widehat{\mathcal{P}}_1^{\uparrow}[j].\mathsf{first}), \mathcal{SE}.\mathsf{Dec}(K_e, \widehat{\mathcal{P}}_1^{\uparrow}[j].\mathsf{second}))$
  if $p.\mathsf{second} = \mathsf{real}$ **then**
    $\Delta \leftarrow \Delta \cup \{p.\mathsf{first}\}$
    $bs \leftarrow$ dummy string of length α
    $ep' \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, +\infty), \mathcal{SE}.\mathsf{Enc}(K_e, bs))$
  else if $p.\mathsf{second} = \mathsf{dum}$ **then**
    $bs \leftarrow$ dummy string of length α
    $ep' \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, +\infty), \mathcal{SE}.\mathsf{Enc}(K_e, bs))$
  else if $p.\mathsf{second} = \mathsf{delim}$ **then**
    $\tau \leftarrow p.\mathsf{second}$
    $bs \leftarrow$ empty string
    for $k = \lfloor \frac{\tau}{\alpha} \rfloor \cdot \alpha$ to $\lfloor \tau \rfloor$ **do**
      if $k \in \Delta$ **then**
        $bs \leftarrow bs||1$
      else
        $bs \leftarrow bs||0$
      end if
    end for
    $ep' \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, \lfloor \frac{\tau}{\alpha} \rfloor), \mathcal{SE}.\mathsf{Enc}(K_e, bs))$
    Send ep' to the server, Clear Δ
  end if
end for

Server:
$\widehat{\mathcal{P}}_2 \leftarrow$ empty array
while Receive ep' **do**
  $\widehat{\mathcal{P}}_2 \leftarrow \widehat{\mathcal{P}}_2 \cup \{ep'\}$
end while

The client and the server execute $\mathsf{OS\text{-}ORT}(k_e; \widehat{\mathcal{P}}_2)$, after which the server receives the encrypted array $\widehat{\mathcal{P}}_2^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.

Server:
Discard the last $d' \cdot t$ pairs from $\widehat{\mathcal{P}}_2^{\uparrow}$.
Discard the first entry of each pair in $\widehat{\mathcal{P}}_2^{\uparrow}$.

Client
$SK_i \leftarrow F_1(SK, i)$
for $j = 0$ to $|\widehat{\mathcal{P}}_2^{\uparrow}| - 1$ **do**
  $bs \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \widehat{\mathcal{P}}_2^{\uparrow}[j])$
  for $k = 0$ to $\alpha - 1$ **do**
    $\epsilon \leftarrow j \cdot \alpha + k$, $eb \leftarrow F_2(SK_i, k') \oplus bs[k]$
    Send (ϵ, eb) to the server
  end for
end for

Server:
$\widehat{\mathcal{BBF}}_i \leftarrow$ empty array
while Receive (ϵ, eb) **do**
  $\widehat{\mathcal{BBF}}_i[\epsilon] \leftarrow eb$
end while
return $\widehat{\mathcal{BBF}}_i$

Fig. 17. An Adapted Variant of WS12 for Oblivious Bloom Filter Creation

```
1: if T is not empty then
2:     Select an arbitrary entry (x, y) from T,
                                   i.e., T[x] = y.
3:     p ← (x, dum)
4:     if y = 0 then
5:         Remove T[x] from T.
6:     else
7:         Update T[x] to (y − 1).
8:     end if
9: else
10:    p ← (+∞, dum)
11: end if
12: return p
```

Fig. 18. GenPa(T)

G Reducing the Robustness Parameter to Lower Client Storage

In this section, we reduce the client storage required by OBFI (Variant 1) without incurring the additional client storage. Our idea is to set the robustness parameter ρ to a value smaller than the security parameter λ (*e.g.,* $\rho = 20$). This still ensures that the bounds given in Formula (3) of Lemma 1 with high probability, namely $1 - e^{-\rho}$. This observation can be exploited to reduce the sender's storage requirements, at the cost of allocating a small additional amount of storage to handle the rare cases where the bounds are violated.

Modified OBD protocol. To achieve this, we modify the original OBD protocol. The algorithmic description of the modified OBD protocol is provided in Fig. 19. In particular, we note that during the output phase, in addition to the buckets produced by the server, the client also retains any elements that may cause bucket overflow. Specifically, for each i, if $|\mathbb{V} \cap \mathcal{W}_i| > t$, the client stores the surplus $(|\mathbb{V} \cap \mathcal{W}_i| - t)$ elements from $\mathbb{V} \cap \mathcal{W}_i$.

In the algorithm details, after the first OSORT on the input array $\mathbb{V}$, the client creates a set L and a table T. L will be used to store the elements from $\mathbb{V}$ that can cause bucket overflows. T will be used to map the index of each bucket that can be underfilled to the remaining number of elements required by the bucket. These L and T can be constructed after one scan on the sorted $\mathbb{V}$.

To be specific, as shown in Line 5-Line 33 in Fig. 19, the variable i is initialized to -1, and as the client scans the elements, notation ζ_i is used to track the number of elements from $\mathbb{V}$ that falls within subrange $\mathcal{W}_i$ (if $i \geq 0$). For each scanned element v_j, the client first determines the subrange in which v_j resides. We denote by i' the index of the subrange $\mathcal{W}_{i'}$ containing this element. If v_j is the first scanned element that falls within $\mathcal{W}_{i'}$ (as indicated by the absence of $\zeta_{i'}$), this implies that the count ζ_i for the previous subrange $\mathcal{W}_i$ has been fully determined. At this point, the client checks the sum $\zeta_i + \zeta_{i-1}$ (if $i > 1$). If this sum is smaller than t, it indicates that, under the original allocation scheme, bucket $i-1$ would not be filled. In this case, the client sets $T[i-1] = t - (\zeta_i + \zeta_{i-1})$. It is important to note that some subranges may have an empty intersection with $\mathbb{V}$. This occurs when $i' \geq i + 2$, meaning that for all k such that $i + 1 \leq k \leq i' - 1$,

Client:

$\rho \leftarrow$ a small constant, *e.g.*, 20.
Select the integer ω that satisfies Formula 1 in Lemma 1, and value c that satisfies Formula 2.
$z \leftarrow \lceil \frac{n}{\omega} \rceil$, $t \leftarrow \lceil c \rceil$

The client and the server execute $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EV})$, after which the server obtains the sorted array $\mathbb{EV}^{\uparrow}$.

Client:

$L \leftarrow$ empty set, $T \leftarrow$ empty table
for $j = 0$ to $(s-1)$ **do**
  Read $\mathbb{EV}^{\uparrow}[j]$ from $\mathbb{EV}^{\uparrow}$
  $v_j \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EV}^{\uparrow}[j])$
  Find the largest integer i' such that $v_j \geq i' \cdot \omega$.
  if $i' < z-1$ and $\zeta_{i'}$ is absent **then**
    if $i > 1$ and $(\zeta_i + \zeta_{i-1}) < t$ **then**
      $T[i-1] \leftarrow t - (\zeta_i + \zeta_{i-1})$
    end if
    for $k = i+1$ to $i'-1$ **do**
      $\zeta_k \leftarrow 0$
      if $k > 1$ and $(\zeta_k + \zeta_{k-1}) < t$ **then**
        $T[k-1] \leftarrow t - (\zeta_k + \zeta_{k-1})$
      end if
    end for
  end if
  if $\zeta_{i'}$ is absent **then**
    $i \leftarrow i'$, $\zeta_i \leftarrow 1$
  else
    $\zeta_i \leftarrow \zeta_i + 1$
  end if
  if $\zeta_i > t$ **then**
    $L \leftarrow L \cup \{v_j\}$
    $ev_j \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum})$
  else
    $ev_j \leftarrow \mathcal{SE}.\mathsf{Enc}(K_e, v_j)$
  end if
  Send (j, ev_j) to the server
end for

Server:

while Receive (j, ev_j) **do**
  $\mathbb{EV}^{\uparrow}[j] \leftarrow ev_j$
end while

The client sends t dummy ciphertexts, each computed as $\mathcal{SE}.\mathsf{Enc}(K_e, \mathsf{dum})$, to the server, which appends them to $\mathbb{EV}^{\uparrow}$.

Client:

$i \leftarrow -1$
for $j = 0$ to $(s+t-1)$ **do**
  Read $\mathbb{EV}^{\uparrow}[j]$ from $\mathbb{EV}^{\uparrow}$
  $v_j \leftarrow \mathcal{SE}.\mathsf{Dec}(K_e, \mathbb{EV}^{\uparrow}[j])$
  if $v_j \neq \mathsf{dum}$ **then**
    Find the largest integer i' such that $v_j \geq i' \cdot \omega$.
    if $\delta_{i'}$ is absent **then**
      $i \leftarrow i'$, $\delta_i \leftarrow j$
      if δ_{i-1} is absent **then**
        $\delta_{i-1} \leftarrow j$
      end if
    end if
  else if $v_j = \mathsf{dum}$ and $j = s$ **then**
    if δ_{z-2} and δ_{z-1} are both absent **then**
      $\delta_{z-2} \leftarrow j$, $\delta_{z-1} \leftarrow j$
    else if only δ_{z-1} is absent **then**
      $\delta_{z-1} \leftarrow j$
    end if
  end if
  if $v_j = \mathsf{dum}$ and $j < s$ **then**
    $p_0 \leftarrow \mathsf{GenPa}(T)$, $p_1 \leftarrow \mathsf{GenPa}(T)$ ▷ $\mathsf{GenPa}(T)$ is introduced in Fig. 18.
  else if $v_j \in \mathcal{W}_0$ **then**
    $p_0 \leftarrow (0, v_j)$, $p_1 \leftarrow \mathsf{GenPa}(T)$
  else if $v_j \in \mathcal{W}_i$ and $j < \delta_{i-1} + t$ **then**
    $p_0 \leftarrow (i, v_j)$, $p_1 \leftarrow (i-1, v_j)$
  else if $v_j \in \mathcal{W}_i$ and $j \geq \delta_{i-1} + t$ **then**
    $p_0 \leftarrow (i, v_j)$, $p_1 \leftarrow \mathsf{GenPa}(T)$
  else if $v_j = \mathsf{dum}$ and $j < \delta_{z-2} + t$ **then**
    $p_0 \leftarrow (z-2, \mathsf{dum})$, $p_1 \leftarrow (z-1, \mathsf{dum})$
  else if $v_j = \mathsf{dum}$ and $\delta_{z-2} + t \leq j < \delta_{z-1} + t$ **then**
    $p_0 \leftarrow \mathsf{GenPa}(T)$, $p_1 \leftarrow (z-1, \mathsf{dum})$
  else if $v_j = \mathsf{dum}$ and $j \geq \delta_{z-1} + t$ **then**
    $p_0 \leftarrow \mathsf{GenPa}(T)$, $p_1 \leftarrow \mathsf{GenPa}(T)$
  end if
  $ep_0 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, p_0.\mathsf{first}), \mathcal{SE}.\mathsf{Enc}(K_e, p_0.\mathsf{second}))$
  $ep_1 \leftarrow (\mathcal{SE}.\mathsf{Enc}(K_e, p_1.\mathsf{first}), \mathcal{SE}.\mathsf{Enc}(K_e, p_1.\mathsf{second}))$
  Send ep_0 and ep_1 to the server
end for

Server:

$\mathbb{EP} \leftarrow$ empty array
while Receive ep_0 and ep_1 **do**
  $\mathbb{EP} \leftarrow \mathbb{EP} \cup \{ep_0, ep_1\}$
end while

The client and the server run $\mathsf{OSORT}^{\mathcal{SE}}(K_e; \mathbb{EP})$, after which the server obtains the sorted array $\mathbb{EP}^{\uparrow}$. ▷ Ordered primarily by the first entry of each pair.

Client:

return L

Server:

Discard $\mathbb{EP}^{\uparrow}[z \cdot t]$ through the end.
$\widehat{\mathbb{EV}} \leftarrow \mathbb{EP}^{\uparrow}$ with all first entries removed
return $\widehat{\mathbb{EV}}$

Fig. 19. Modified OBD Protocol

we have $\mathbb{V} \cap \mathcal{W}_k = \emptyset$. In such cases, we set $\zeta_k = 0$ and check the sum $\zeta_k + \zeta_{k-1}$ (if $k > 1$). If this sum is smaller than t, then the client sets $T[k-1] = t - (\zeta_k + \zeta_{k-1})$. Through this procedure, we record in T both the indices of underfilled buckets and the number of elements required to fill each of them. It suffices to check buckets only up to index $z - 3$, since the introduction of dummy elements later in the protocol guarantees that buckets $i - 2$ and $i - 1$ will always be filled to capacity.

In addition to constructing T, we maintain a list L to record elements that cause bucket overflows. Specifically, if $\zeta_{i'}$ does not yet exist, the client updates $i \leftarrow i'$ and initializes $\zeta_i = 1$. Otherwise, i is already equal to i', and ζ_i is incremented by 1. If ζ_i exceeds t, this indicates that v_j would cause bucket i to overflow. In that case, we add the element to L, encrypt a dummy element $\approxeq\!\!>$, and write it back. If $\zeta_i \leq t$, the element is simply re-encrypted and written back.

After the completion of the construction for L and T, and the server receives the updated $\mathbb{EV}^{\uparrow}$, t dummy ciphertexts are appended to the end of $\mathbb{EV}^{\uparrow}$, and then the pair generation phase starts. In the pair generation phase, the client scans (and decrypts) elements from $\mathbb{EV}^{\uparrow}$. Similar to the first scan, we initialize $i = -1$ and denote by i' the index of the subrange in which the scanned element v_j resides (when $v_j \neq \mathsf{dum}$). If v_j is the first scanned element from $\mathcal{W}_{i'}$, indicated by the absence of $\delta_{i'}$, we update $i \leftarrow i'$ and initialize $\delta_i = j$. The subsequent pair-generation step also requires the value of δ_{i-1} to fill bucket $i - 1$ correctly. If δ_{i-1} is absent at this point, we assign $\delta_{i-1} = j$. When the scan reaches v_s with $v_s = \mathsf{dum}$, the pair-generation condition depends on both δ_{i-2} and δ_{i-1}. If neither exists, we set both to s; if only δ_{i-1} is absent, we assign $\delta_{i-1} = s$. For each scanned element, two pairs must be generated according to the specified condition. At this stage, our modified protocol differs from the original OBD protocol in two ways:

- **Conditionally replacing $+\infty$.** In the original protocol, if the first entry of a pair is to be set to $+\infty$, this assignment is now performed *only if* the table T is empty. If T is non-empty, the client instead retrieves an entry (x, y) from T and sets the first entry of the pair to x. Furthermore, if $y - 1 > 0$, the client updates $T[x]$ to $(y - 1)$; otherwise, the entry $T[x]$ is removed from T. We denote this modified pair-generation procedure by $\mathsf{GenPa}(T)$ (see Fig. 18).
- **Handling dummy elements.** If the scanned element v_j is dum and $j < s$, then both pairs associated with this element are generated using $\mathsf{GenPa}(T)$.

Each generated pair is then encrypted and sent to the server. All subsequent steps proceed identically to those in the original OBD protocol, except that the client returns the set L.

Guarantee that T will be emptied. A potential concern is whether the table T is guaranteed to be empty by the end of the pair generation phase. The condition that T is emptied is equivalent to requiring that, before the generation of all $2(s + t)$ pairs is completed, each of the z buckets is associated with exactly t pairs. In other words, it suffices to show that

$$2(s + t) \geq z \cdot t,$$

which ensures that T will always be emptied in each execution of the modified OBD protocol. As shown by the inequality below, the condition $2(s+t) \geq z \cdot t$ always holds.

$$2(s+t) \geq 2(s+\frac{s\omega}{n}) = 2\frac{s\omega}{n}(\frac{n}{\omega}+1) > t \cdot z$$

Writing L to the Bloom Filter. In OBFI Variant 1, the OBD component is replaced with the modified OBD protocol described above. This modification enables the distribution of hash values corresponding to the current batch into buckets. During this process, the client locally retains, for each subrange $\mathcal{W}_i$ such that $|\mathbb{V} \cap \mathcal{W}_i| > t$, the largest $(|\mathbb{V} \cap \mathcal{W}_i| - t)$ elements from the set $\mathbb{V} \cap \mathcal{W}_i$. All such retained elements are collected into a list L.

When writing the encrypted Bloom filter, the client reads and decrypts each bucket $\widehat{\mathcal{B}}_i$. At this stage, it is sufficient to insert into the decrypted bucket $\mathcal{B}_i$ all elements from L that fall within the corresponding subrange $\mathcal{W}_i$. Once this insertion is completed, the subsequent steps proceed identically to those in the original OBFI Variant 1 protocol.