InstantOMR: Oblivious Message Retrieval with Low Latency and Optimal Parallelizability

Haofei Liang
Shanghai Jiao Tong University
lianghaofei@sjtu.edu.cn

Zeyu Liu
Yale University
zeyu.liu@yale.edu

Eran Tromer ✉
Boston University
tromer@bu.edu

Xiang Xie ✉
Primus Labs
xiexiangiscas@gmail.com

Yu Yu ✉
Shanghai Jiao Tong University
yyuu@sjtu.edu.cn

Abstract

Oblivious message retrieval (OMR) addresses the expensive secure message retrieval process in anonymous messaging systems and private blockchains: It enables resource-limited recipients to outsource detection and retrieval of their messages, while preserving privacy.

This work introduces InstantOMR, a novel OMR scheme that combines TFHE functional bootstrapping with standard RLWE operations in a hybrid design. InstantOMR is specifically optimized for low latency and high parallelizability. Our implementation, using the Primus-fhe library (and estimates based on TFHE-rs), demonstrates that InstantOMR offers the following key advantages:

- **Low latency:** InstantOMR achieves ∼860× lower latency than SophOMR, the state-of-the-art single-server OMR. This translates directly into reduced recipient waiting time (by the same factor) in the *streaming* setting, where the detector processes incoming messages on-the-fly and returns a digest immediately upon the recipient becoming online.
- **Optimal parallelizability:** InstantOMR scales near-optimally with CPU cores (by processing messages independently), so for high core counts, it is faster than SophOMR (whose parallelism is constrained by its reliance on BFV).

1 Introduction

In privacy-preserving message delivery systems require more than end-to-end encryption that protects the *contents* of messages. Leakage of *metadata*, exploited by traffic analysis can also reveal sensitive information [6, 33]. Thus, it is widely recognized that metadata privacy should be fully protected in applications like private messaging [8, 11, 20, 42] or privacy-preserving blockchains [9, 12, 35]

Protecting the identity of message recipients poses a challenge: from a recipient's perspective, a message addressed to them can appear anywhere in the message sequence (e.g., the public ledger, in blockchain applications). Therefore, to find their *pertinent* messages, they need to scan and trial-decrypt *all* messages. This introduces a significant communication and computation burden for resource-limited recipients (e.g., apps running on mobile devices).

For such recipients, it is desirable to outsource this to a powerful server in a private way. *Oblivious Message Retrieval* (OMR) [28] addresses this problem, using homomorphic encryption in the single-server setting.

Summarized system model. We first briefly summarize the OMR system model. In the system, there exist *senders* who want to send messages to the *recipients* without leaking the identity of the recipients. Each *message* consists of a *payload* (the content the sender would like to send) and a *clue* (usually a ciphertext, indicating who the recipient is in a privacy-preserving way) generated using the *clue key* from the recipient. All these messages are put on a *bulletin board* (or *board* for short). When the recipients want to obtain their messages, they send a *detection key* to an untrusted third party, the *detector*. The detector then uses the board and the detection key to generate a *digest*, which the recipient decodes and obtains all the payloads pertinent to them. This workflow is also visualized in Fig. 1.

Existing OMR constructions. All prior single-server OMR schemes [26, 28–30] utilize the BFV homomorphic encryption scheme [13, 22]. Such a construction achieves high throughput: SophOMR [26] takes only ∼1.83 minutes to do retrieval over 2^{16} messages (∼1.68ms per message), while PerfOMR [30] takes ∼8.0 minutes.

However, a major bottleneck of these constructions is latency: even when retrieving a single message, SophOMR [26] takes ∼76 seconds, and PerfOMR [30] takes ∼89 seconds. This is because the throughput is achieved via amortizing costs across many messages. The high latency impacts user experience in scenarios requiring frequent, small-sized retrievals or real-time streaming of updates.

This also causes these existing schemes to have limited parallelizability: to effectively utilize c cores, [26, 30] need to batch-process $c \cdot N$ messages for large N (e.g., 2^{15} or 2^{16}) to fully utilize these c cores, due to the limitation that BFV can-

not benefit much from multi-threading. Ideally, we would like to be able to process each individual message independently, implying that detector retrieval computations of nontrivial size can be optimally parallelized across any reasonable number of cores (i.e., requires only c messages to fully utilize c cores).

We thus ask the following question:

Can we build OMR with low latency and optimal parallelizability without sacrificing much throughput?

This paper shows such a scheme.

1.1 Our Contribution

We present InstantOMR, a novel OMR construction leveraging TFHE [16], optimized for latency and parallelizability:

- **Low-Latency Per-Message Processing**: InstantOMR operates the homomorphic retrieval circuit over individual messages rather than message batches in prior works. This eliminates high-latency inherent in BFV-based designs.
- **Seamless Parallelism**: Per-message processing also enables optimal parallel scalability. In particular, only c messages are needed to fully utilize c cores.
- **Hybrid Use of TFHE and RLWE Operations**: InstantOMR employs a novel two-layer structure of TFHE functional bootstrapping for homomorphic clue decryption and checking. Regular RLWE operations are then used to complete homomorphic encoding. This hybrid use of TFHE and RLWE enables the above advantages without significantly compromising overall performance.

Implementation. We implemented InstantOMR (available in [2]) using Primus-fhe [36] (and estimated using TFHE-rs [43]), and evaluated it against state-of-the-art BFV-based OMR schemes (PerfOMR,SophOMR). Our results show:

- **Significant Latency Reduction:** For single-message retrieval on a single CPU core, InstantOMR achieves 864× lower latency than SophOMR and 1008× than PerfOMR.
- **Optimal Parallel Scaling:** InstantOMR exhibits near-linear speedup with the number of cores. Using 180 cores, it is approximately ∼3× faster than SophOMR and ∼13× faster than PerfOMR for 2^{16} messages (which maximizes SophOMR's and PerfOMR's efficiency). Performance can approach single-message latency given 2^{16} cores.
- **Streaming:** In real-time applications with on-the-fly message processing, InstantOMR reduces recipient wait time to < 0.1s, compared to $\gtrsim$76 seconds in prior schemes.

Note that these improvements are estimated when the implementation uses TFHE-rs [43] (see § 6.4), while the runtime of our implementation using Primus-fhe is roughly 3× slower than this estimation (see § 6).

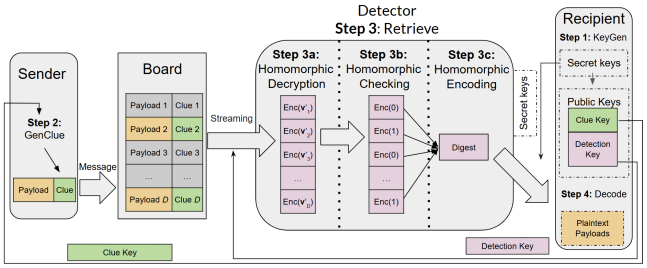

Figure 1: Summary of system model and main paradigm.

1.2 Related Work

Oblivious message retrieval. Oblivious message retrieval (OMR) was first introduced in [28] as a solution to the problem of recipient privacy in anonymous messaging systems. Group OMR [29] extends it to the group setting, where each message may have multiple recipients. PerfOMR [30] provides a more efficient OMR construction by using RLWE instead of LWE for clue generation. SophOMR [26] further improves the efficiency by fully exploiting the native homomorphic SIMD structure of the underlying HE scheme. [27] focuses security against spamming/DoS attacks from malicious senders, proposing a provably DoS-resistant OMR built upon PerfOMR. All these works rely on BFV leveled homomorphic encryption scheme [13, 22] and work with a single server as our paper. We discuss multi-server-based OMR protocols in Appx A.

Fuzzy message detection and Private Signaling. Fuzzy message detection (FMD) [8] and Private Signaling (PS) [24, 32] are two related primitives. However, the former provides weaker security and the latter assumes very strong environment assumptions. See Appx A for details.

2 Technical Overview

This section provides a high-level overview of the InstantOMR construction (with simplifications). We assume readers are familiar with the OMR model and its interface definitions (introduced in [28] and recalled in § 4), as well as the general OMR paradigm used in prior works (summarized in § 4). These are also illustrated in Fig. 1. Parameters are summarized in Table 7 for reference.

Why prior works have high latency. We first discuss the issues in prior constructions. Recall that the detector processes all clues to detect pertinent messages (i.e., the messages addressed to the recipient), by performing decryption of the clue ciphertext using the recipient's decryption key, and then checking whether the decrypted vector is $(1, 1, \ldots, 1)$. To achieve privacy, all of this is done by homomorphic encrypted evaluation, under a transient detection key provided by the recipient. These homomorphic decryption and homomorphic checking

steps (3a and 3b in Fig. 1 and § 4) are the main source of latency. Specifically, to maximize throughput, prior works realized these steps using BFV encryption with a ring dimension of 2^{16} (for [26], and 2^{15} for [30]). These steps take the same amount of time whether processing a single message or a batch of 2^{16} messages, and they account for a large portion of the total runtime (approximately two-thirds). Thus, while the per-message cost is as low as $\sim$1.5ms, the latency for a single message can still be (at least) $\sim$76 seconds.

Using FHEW/TFHE instead of BFV. Since the issue arises from BFV requiring a batch of 2^{16} messages (i.e., the ring dimension) to maximize efficiency, a natural idea is to instead use FHEW [21] or TFHE [16], since these work on individual values. However, as noted by [28], using TFHE naively (as a black-box binary gate operator) takes tens of seconds per message. This is because, although TFHE supports fast gate-level operations [16], Step 3c (see Fig. 1 and § 4) requires approximately $\sim$5000 gates.[1] Compared to BFV-based OMR, this approach has comparable latency, and hugely reduced throughput. Our approach tailors the use of TFHE to vastly improve both, and we summarize our techniques below.

2.1 OMR Setup

Before diving into details, we first describe the initial setup needed to use TFHE.

Setup of our solution. We follow a paradigm similar to prior works, where the setup phase corresponds to Step 1 and Step 2, as visualized in Fig. 1 and summarized in § 4.

When joining the system (Step 1), the recipient generates a key pair $(\mathsf{PKE.pk}, \mathsf{PKE.sk})$ for a lattice-based public-key encryption (PKE) scheme and publishes $\mathsf{pk}_{\mathsf{clue}} := \mathsf{PKE.pk}$. For simplicity, let this PKE be Regev05 LWE-based encryption [38] (and this naturally extends to Ring-LWE-based schemes, as shown in [30]). For detection, the recipient generates an FHE key pair $(\mathsf{FHE.pk}, \mathsf{FHE.sk})$, computes $\mathsf{ct}_{\mathsf{sk}} \leftarrow \mathsf{FHE.Enc}(\mathsf{PKE.sk})$ (i.e., $\mathsf{PKE.sk}$ encrypted under FHE), and sends $\mathsf{pk}_{\mathsf{detect}} := (\mathsf{FHE.pk}, \mathsf{ct}_{\mathsf{sk}})$ to the detector.

In Step 2, the sender uses $\mathsf{PKE.pk}$ to encrypt ℓ ones, i.e., $\mathsf{PKE.Enc}((1,\ldots,1))$, as the clue[2]. This encryption serves to identify the intended recipient.

We now move on to the Retrieve process (Step 3), where the detector helps the recipient retrieve its messages homomorphically: Step 3a, homomorphic decryption; Step 3b, homomorphic checking; and Step 3c, homomorphic encoding.

[1] For a 612-byte payload as in prior works, there are at least 4896 bits to encode into a digest per message.

[2] For simplicity, think of Enc as encrypting each 1 into a separate ciphertext, and subsequently Dec on a vector of ciphertexts outputting a vector of plaintexts. See § 5.1 for how to make it more efficient.

2.2 Homomorphic Decryption and Checking via Tailored TFHE

As mentioned earlier, a natural idea to reduce latency is to use TFHE instead of BFV for homomorphic decryption. However, using TFHE naively as a black-box gate evaluator is far too inefficient. To make it practical, we first revisit TFHE bootstrapping from a slightly different perspective, and then show how to leverage it in our construction.

TFHE bootstrapping. Let the TFHE parameters be: secret key dimension n, ciphertext moduli q_1, q_2, and plaintext moduli p_1, p_2 (where $p_1 \mid q_1$ and $p_2 \mid q_2$).

TFHE ciphertexts are standard LWE ciphertexts. TFHE bootstrapping is usually described as standard FHE bootstrapping, which takes a valid ciphertext as input and outputs another valid ciphertext encrypting the same plaintext with reduced noise; or as gate bootstrapping, which, given two ciphertexts encrypting bits m_1, m_2, outputs a ciphertext of $\mathsf{NAND}(m_1, m_2)$. However, we utilize a more general version, functional bootstrapping, as follows:

Given a vector $(\boldsymbol{a}, b) \in \mathbb{Z}_{q_1}^{n+1}$, an LWE secret key $\mathsf{sk} \in \mathbb{Z}^n$, and any negacyclic function $f : \mathbb{Z}_{p_1} \to \mathbb{Z}_{p_2}$ (where "negacyclic" means $f(x) = -f(x + p_1/2) \in \mathbb{Z}_{p_2}$ for all $x \in \mathbb{Z}_{p_1}$), TFHE functional bootstrapping outputs a ciphertext $\mathsf{ct} = \mathsf{TFHE.Enc}(f(m)) \in \mathbb{Z}_{q_2}^{n+1}$, where $m \leftarrow \mathsf{Dec}(\mathsf{sk}, (\boldsymbol{a}, b)) \in \mathbb{Z}_{p_1}$.

Note that the decryption input $(\boldsymbol{a}, b)$ can be *any* vector in $\mathbb{Z}_{q_1}^{n+1}$ — not necessarily a valid LWE ciphertext under sk — and still produces a valid ciphertext encrypting a plaintext in the message space under sk. This is crucial because, for impertinent messages in OMR, the ciphertexts are computationally indistinguishable from random vectors in $\mathbb{Z}_{q_1}^{n+1}$ (with respect to sk), and hence are not strictly valid LWE ciphertexts. Additionally, the ability to apply *any* negacyclic function over two potentially different moduli p_1, p_2 also enables significant performance improvements in our construction.

Homomorphic decryption (Step 3a). Our first step is to homomorphically decrypt all the clues: for pertinent messages, the ciphertexts should decrypt to $\boldsymbol{v} = (1, 1, \ldots, 1) \in \mathbb{Z}_{p_2}^{\ell}$; for impertinent messages, the ciphertexts should decrypt to some $\boldsymbol{v}' \neq \boldsymbol{v} \in \mathbb{Z}_{p_2}^{\ell}$. This is achieved by bootstrapping using a specially designed f below.

For the clue of a pertinent message, $\mathsf{Dec}(\mathsf{sk}, (\boldsymbol{a}, b)) = 1$. Thus, the minimum requirement for f is that $f(x) = 1$ for $x = 1$. Recall that f is negacyclic, and thus we also need $f(1 + p_1/2) = -1 = p_2 - 1$.

Conversely, for an impertinent message clue $(\boldsymbol{a}', b')$, $\mathsf{Dec}(\mathsf{sk}, (\boldsymbol{a}', b'))$ is indistinguishable from uniform over $\mathbb{Z}_{p_1}$ (by the hardness of LWE). Thus, to distinguish pertinent from impertinent messages, we can set $f(x') = 0$ for any $x' \notin \{1, 1 + p_1/2\}$ (which still satisfies the negacyclic condition since $f(x') = 0 = -0 = f(x' + p_1/2)$).

To summarize, given a clue $\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell = \mathsf{PKE.Enc}(\boldsymbol{v})$, the detector performs $\mathsf{ct}'_i \leftarrow \mathsf{TFHE.Boot}(\mathsf{ct}_i, f)$. In this case:

- For pertinent messages, $\mathsf{TFHE.Dec}(\mathsf{ct}'_i) = 1$ (secret key implicitly taken) for all $i \in [\ell]$.
- For impertinent messages, for all $i \in [\ell]$:
  - $\mathsf{TFHE.Dec}(\mathsf{ct}'_i) = 0$ with probability $1 - 2/p_1$
  - $\mathsf{TFHE.Dec}(\mathsf{ct}'_i) = 1$ with probability $1/p_1$
  - $\mathsf{TFHE.Dec}(\mathsf{ct}'_i) = -1$ with probability $1/p_1$

Homomorphic checking (Step 3b). After homomorphic decryption, the goal is to check if the decrypted values are $\boldsymbol{v}$, i.e., whether $\mathsf{TFHE.Dec}(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_\ell) = (1, 1, \ldots, 1) \in \mathbb{Z}_{p_2}^\ell$.

A naive way is to perform a homomorphic AND gate over all the ciphertexts in the vector (for simplicity, assume the AND gate work for -1 by treating it as 1): if the result after these $\ell - 1$ AND gates is still an encryption of 1, the vector encrypts $\boldsymbol{v}$; otherwise, the vector encrypts some $\boldsymbol{v}' \neq \boldsymbol{v}$. However, this introduces another $\ell - 1$ gate bootstrapping.

To reduce the cost, instead, we first sum up all the ciphertexts and then perform another *layer* of bootstrapping (different parameters and function). More formally, we first compute $\mathsf{ct}'' \leftarrow \sum_i \mathsf{ct}'_i$. If the message is pertinent, ct'' encrypts $\ell \in \mathbb{Z}_{p_2}$. Otherwise, ct'' encrypts $m'' \in [-\ell, \ell - 1]$ except with probability $(1/p_1)^\ell$ (i.e., unless all ℓ clue ciphertexts were mapped to 1 in the previous step). For reasons that will become clear shortly, we set $p_2 = 4(\ell + 1)$.[3]

Given the second-layer bootstrapping parameters (for the homomorphic check) n_2, q_2, p_2, we define the negacyclic function $f_2 : \mathbb{Z}_{p_2} \to \mathbb{Z}_{p_2}$ for the homomorphic checking as follows[4]: $f_2(x) = \begin{cases} 1 & \text{if } x = \ell \\ -1 & \text{if } x = \ell + p_2/2 = 3\ell + 2 \\ 0 & \text{otherwise} \end{cases}$.

Observe that any $x \in [-\ell, \ell - 1] = [3\ell + 4, 4\ell + 3] \cup [0, \ell - 1]$ is not in $\{\ell, 3\ell + 2\}$. Thus, such an f_2 allows us to map all pertinent messages to 1 and impertinent messages to 0 (except with probability $1/{p_2}^\ell$), which is exactly the homomorphic check we desire. With this, we compute $\mathsf{ct}''' \leftarrow \mathsf{TFHE.Boot}(\mathsf{ct}'', f_2)$ to complete the homomorphic checking.

2.3 Homomorphic encoding via RLWE

After the previous steps, we obtain ct''', which encrypts 1 if and only if the message is pertinent (except for small probability). The next step is to homomorphically encode the payloads of the pertinent messages into a digest (step 3c).

At this point, continuing to use TFHE here is wasteful, since the encoding step can take thousands of gate bootstrappings, and functional bootstrapping does not help here. Instead, we observe that the homomorphic encoding used in prior works only involves plaintext-by-ciphertext multiplications, which can be performed efficiently without bootstrapping since LWE ciphertexts are linearly homomorphic.

In other words, given ciphertexts $\mathsf{PV} := (\mathsf{ct}'''_1, \ldots, \mathsf{ct}'''_\mathsf{D})$ indicating whether the D payloads $\mathsf{PLD} := (\mathsf{pld}_1, \ldots, \mathsf{pld}_\mathsf{D})$ are pertinent, the homomorphic encoding can be realized as: $(\mathsf{PV} \circ \mathsf{PLD}) \times A$, where $\circ$ denotes element-wise multiplication, and A is a matrix of size $\mathsf{D} \times O(\bar{k})$, where $\bar{k}$ is the (expected) upper bound on the number of pertinent messages.[5] This eliminates the dependency on bootstrapping calls.

Converting LWE ciphertexts to RLWE ciphertexts. Naively using LWE ciphertexts for homomorphic multiplication remains costly. Each LWE ciphertext consists of $n_2 + 1$ elements in $\mathbb{Z}_{q_2}$ but encrypts a single value in $\mathbb{Z}_{p_2}$. Therefore, encoding a payload of, say, 100 $\mathbb{Z}_{p_2}$ elements would require $100 \cdot (n_2 + 1)$ multiplications in $\mathbb{Z}_{q_2}$ for just one plaintext-by-ciphertext multiplication.

To improve efficiency, we convert each LWE ciphertext into an RLWE [31] ciphertext before multiplications. One may observe that an RLWE ciphertext is essentially a BFV ciphertext. We use RLWE/BFV instead of TFHE here is because for the step of homomorphic encoding, even for one message, we can leverage the SIMD packing (unlike for the homomorphic decryption or checking steps), thus providing significantly better efficiency than directly using TFHE bootstrappings.

This conversion from LWE to RLWE has been systematically studied in [15], so we defer the details to § 5.[6]

In short, given an LWE ciphertext encrypting $m \in \mathbb{Z}_p$, we convert it into an RLWE ciphertext, encrypting $m' \in \mathcal{R}_p$ (see the definition of m' below), where $\mathcal{R}$ is a polynomial ring and $\mathcal{R}_p := \mathcal{R}/p\mathcal{R}$ (see § 3 for definition).

Homomorphic encoding. We now describe the homomorphic encoding process. We adopt the core idea as in [28], but apply it in a slightly different way: we perform all encoding using polynomial coefficients.

Recall that in the RLWE setting, ciphertexts have the form $(a, b) \in \mathcal{R}_q^2$ satisfying $b = as + e + \lfloor (q/p) \cdot m_\mathcal{R} \rceil_q$, where $m_\mathcal{R} \in \mathcal{R}_p$ is represented as a polynomial $m_\mathcal{R}(X) = \sum_{i \in [0,N)} m_i X^{i-1}$ for ring dimension N. In the LWE-to-RLWE conversion discussed earlier, we have output RLWE ciphertext encrypting m' being a constant polynomial: $m'(X) := m \in \mathcal{R}_p$.

With this, we now detail the encoding process. This step follows the encoding scheme in [28], with the key difference that we encode messages in *coefficients* rather than *slots*.[7]

Encode the indices. The first step is to enable the recipient to identify which messages are pertinent, i.e., to recover the indices of pertinent messages. To achieve this: assuming there are k pertinent messages, the detector initializes $N_s > k$ "buckets", each consisting of an "accumulator" and a "counter", both of which are initialized to 0.

Each message i (with a corresponding $\mathsf{PV}[i]$ ciphertext, encrypting a bit 1 or 0 to indicate pertinency, output from

[3] In practice, we choose $p_2 > 4\ell$ as a power of 2 for ease of implementation. Here, we use $p_2 = 4(\ell + 1)$ for clarity of exposition.

[4] Actually, we will set the output of f_2 to be in $\mathbb{Z}_{p_3}$ for $p_3 \gg p_2$ for better efficiency in later steps (see §§ 5.2.2 to 5.2.4). For simplicity, we use p_2 here.

[5] The matrix A represents a linear encoding method, detailed later.

[6] Looking ahead, we let the second-layer to output an RLWE ciphertext whose constant term encodes the desired value to avoid redundant steps.

[7] To fully utilize the "slots", a special plaintext modulus is needed. This is not necessary for "coefficients", thus enabling more flexible parameters.

previous steps) is assigned uniformly at random to one of the $N_s > k$ buckets. The accumulator for that bucket is updated as $\mathsf{Acc} \leftarrow \mathsf{Acc} + i \cdot \mathsf{PV}[i]$, and the counter for that bucket is updated as $\mathsf{Ctr} \leftarrow \mathsf{Ctr} + \mathsf{PV}[i]$ (both updated homomorphically).

Upon receiving all the encrypted buckets, the recipient decrypts each one. If a bucket has counter $\mathsf{Ctr} = 1$, then the accumulator value Acc reveals a single pertinent message index. Otherwise, the bucket contains a collision and is discarded. With some probability, the recipient recovers a subset of the k pertinent indices. To recover all k indices, the detector repeats this process independently for ℓ_{max} trials. The choices of N_s, ℓ_{max} are detailed in § 5.4.2.

We encode each accumulator with $\lceil \frac{\log(\mathsf{D})}{\log(p)} \rceil$ coefficients, where p is the RLWE plaintext modulus. As prior works, we use an additional coefficient to encode the counter.

Encode the payloads. With the pertinent indices identified, encoding the payloads becomes straightforward. The detector first samples a uniformly random matrix $M \leftarrow_\$ \mathbb{Z}_p^{\mathsf{D} \times m}$, and then computes the product $(\mathsf{PV} \circ \mathsf{PLD}) \times M$, resulting in $m > k$ linear combinations of the pertinent payloads.

Since the recipient knows which indices are pertinent, it can use these linear combinations, along with knowledge of M (represented as a seed sent to the recipient), to recover all pertinent payloads via, e.g., Gaussian elimination.

As in the previous step, we encode both the payloads PLD and the matrix M in the polynomial coefficients. The choice of m is discussed in more detail in § 5.4.2.

2.4 Limitations

Reduced throughput. While InstantOMR improves latency (i.e., the time taken to process a single message) compared to prior works [26, 30], its throughput (number of processed messages per second) is lower when using a single thread. To match the throughput of single-core SophOMR [26], our implementation of InstantOMR requires 180 cores (and the estimation using the faster TFHE-rs library [43] would require ∼64 cores). Note that SophOMR itself would not benefit much from multiple cores ($< 1.5\times$ throughput using 180 cores; see § 6.3). Thus, when optimizing for throughput per core regardless of latency, prior works such as SophOMR remain preferable, as discussed below.

Use-case scope. InstantOMR shines in applications where streaming or real-time updates (§ 5.5) are required, or where only a small number of messages (e.g., $100c$ messages, for c being the number of CPU cores available to the server) need to be processed. For large-scale applications (e.g., $10000c$ messages), SophOMR [26] may still be preferred.

3 Preliminaries

Notation. We will use bold lower-case letters $\boldsymbol{a}$, $\boldsymbol{b}$, ... to denote column vectors over $\mathbb{Z}$. For $q \in \mathbb{Z}$, we identify $\mathbb{Z}_q$ with the set $[-q/2, q/2) \cap \mathbb{Z}$ by default. The i-th element of $\boldsymbol{a}$ is denoted as $a[i]$. Let $[n]$ denote the set $\{1, \ldots, n\}$, $[a,b]$ denote the set $\{a, a+1, \ldots, b-1, b\}$, and $[a,b)$ denote the set $\{a, a+1, \ldots, b-1\}$.

We denote the cyclotomic ring as $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$, where N is a power of 2. $\mathcal{R}_Q = \mathcal{R}/(Q\mathcal{R})$ represents the polynomial from $\mathcal{R}$ with coefficients modulo Q. Elements of $\mathcal{R}_Q$ are represented in the form of $a(X)$, $b(X)$, and $m(X)$. When the context is clear, the polynomial notation (X) may be omitted, and we only use a, b, or m. The i-th coefficient of $a(X)$ is denoted as a_i.

We use $\boldsymbol{x} \leftarrow \mathcal{D}^N$ to denote the sampling of (vector) $\boldsymbol{x}$ from the distribution $\mathcal{D}$ of length $N \geq 1$. For a set S, we use $x \leftarrow_\$ S$ to denote the sampling of x uniformly from all elements of S. The discrete Gaussian distribution is denoted by χ_σ, with mean being 0 and standard deviation being σ.

3.1 Homomorphic Operations

3.1.1 TFHE Cryptosystem

Here, we recall TFHE [16] cryptosystem in a way that is sufficient to understand our construction without going into much detail about the detailed realization of TFHE. Before recalling TFHE, we first define the LWE and RLWE ciphertexts.

LWE. The LWE encryption of the message $m \in \mathbb{Z}_t$ has form: $\mathbf{LWE}_{\boldsymbol{s}}^{t/q}(m) = (\boldsymbol{a}, b) = (\boldsymbol{a}, \langle \boldsymbol{a}, \boldsymbol{s} \rangle + \frac{q}{t} m + e \mod q)$, where $\boldsymbol{s} \in \mathbb{Z}^n$ is the LWE secret key, $\boldsymbol{a} \leftarrow_\$ \mathbb{Z}_q^n$ and $e \leftarrow \chi_\sigma$.

RLWE. The RLWE encryption of the message $m \in \mathcal{R}_t$ is of the form: $\mathbf{RLWE}_z^{t/Q}(m) = (a, b) = (a, az + \frac{Q}{t} m + e)$, where $z \in \mathcal{R}$ is the RLWE secret key, $a \leftarrow_\$ \mathcal{R}_Q$ and $e \leftarrow \chi_\sigma^N$.

Note that a $\mathcal{R}$ element is represented by a polynomial. For example, $m \in \mathcal{R}_t$ is a polynomial $m(X) = \sum_i m_i X^i$ for $i \in [0, N)$ where N is the ring dimension of $\mathcal{R}$, and $m_i \in \mathbb{Z}_t$.

Remark 3.1. For brevity, we may sometimes omit t/q and t/Q from $\mathbf{LWE}_{\boldsymbol{s}}^{t/q}(m)$, $\mathbf{RLWE}_z^{t/Q}(m)$ when context is clear.

Remark 3.2. The RLWE assumption says that $\mathsf{RLWE}(0)$ and $u_1, u_2 \leftarrow_\$ \mathcal{R}_Q$ are computationally indistinguishable. Since the RLWE assumption is standard, we omit the details due to space reasons and refer the readers to [30, 31] for details. Additionally, for TFHE to be secure, we need RLWE to hold even when $(a_i, b_i \mathsf{sk} + \phi_i(\mathsf{sk}) + e_i)$ is given, where $\phi_i : \mathcal{R}_Q \to \mathcal{R}_Q$ is a function determined prior to a_i, e_i and $a_i \leftarrow_\$ \mathcal{R}_Q, e_i \leftarrow_\$ \chi_\sigma$ as above. This is also standard and detailed in [34].

Next, we briefly introduce several TFHE building blocks: "Functional Bootstrapping", "Key Switching", "Modulus Switching" and "Extract". They together are used to build the entire TFHE scheme, but for concrete efficiency, we use them in a modular way and thus we introduce all of them separately instead of introducing the entire TFHE as a black-box.

Functional Bootstrapping. Functional bootstrapping, the key component of TFHE, is defined as TFHE.Boot :

$\mathsf{LWE}_{\boldsymbol{s}}^{p_1/q} \times f \times \mathsf{BSK} \to \mathsf{RLWE}_z^{p_2/Q}$ where $f : \mathbb{Z}_{p_1} \to \mathbb{Z}_{p_2}$ is a negacyclic function (i.e., $f(x) = -f(x + p_1/2)$ for all $x \in \mathbb{Z}_{p_1}$), BSK is the bootstrapping key, $\boldsymbol{s}$, z, p_0, p_1 and q, Q represent the secret key, plaintext modulus and ciphertext modulus, respectively. Let us assume the input ciphertext be $\mathsf{ct} \in \mathbf{LWE}_{\boldsymbol{s}}^{p_1/q}(m)$, then the output ciphertext $\mathsf{TFHE.Boot}(\mathsf{ct}, f, \mathsf{BSK})$ would be $\mathsf{ct}' \in \mathbf{RLWE}_z^{p_2/Q}(f(m) + \sum_{i=1}^{N-1} r_i X^i)$ where r_i is some dummy value. The generation of bootstrapping key BSK is defined as $\mathsf{TFHE.GenBootKey}(\boldsymbol{s}, z)$ (see [16] for its details, which are not needed for our discussion).

Remark 3.3. We define functional bootstrapping to output a RLWE ciphertext, instead of LWE as in [16]. We describe it in this alternative way since for some of our TFHE bootstrapping calls we do not need to extract LWE from RLWE (see below).

Key Switching. Given key switching key $\mathsf{KSK}_{s_1 \to s_2}$, the algorithm $\mathsf{FHE.KS} : (\mathsf{R})\mathsf{LWE}_{s_1} \times \mathsf{KSK}_{s_1 \to s_2} \to (\mathsf{R})\mathsf{LWE}_{s_2}$ converts an input (R)LWE ciphertext encrypted under the s_1 to a ciphertext (R)LWE ciphertext encrypted under s_2 without altering the message. $\mathsf{FHE.GenKeySwitchingKey}(s_1, s_2)$ denote the generation for $\mathsf{KSK}_{s_1 \to s_2}$.

Modulus Switching. The modulus switching algorithm interface is defined as $\mathsf{FHE.MS}_{q_1 \to q_2} : \mathsf{LWE}^{q_1} \to \mathsf{LWE}^{q_2}$ for ciphertext modulus q_1, q_2.

Extract. Let's assume the RLWE dimension is N, then at most N messages can be encrypted in an RLWE ciphertext. Given a ciphertext $\mathsf{ct} = \mathbf{RLWE}(\sum_{i=0}^{N-1} m_i X^i) := (a, b) \in \mathcal{R}^2$, the algorithm $\mathsf{FHE.Extract}(\mathsf{ct}, i)$ will return an LWE ciphertext which encrypts m_i. Note that if $i \leq \ell$, then only the first ℓ coefficients of b are needed.

3.1.2 Homomorphic Trace

$\mathsf{FHE.HomTrace}$: given a ciphertext $\mathbf{RLWE}(m + \sum_{i=1}^{N-1} m_i X^i)$, outputs $\mathbf{RLWE}(m)$ (using trace keys $\mathsf{TraceKey} \leftarrow \mathsf{FHE.GenTraceKey}(z)$ with secret key z). The realization is deferred to our full version due to space

3.1.3 RLWE Plaintext Multiplication

Given a plaintext polynomial $p(X) \in \mathcal{R}_t$ and a ciphertext $\mathsf{ct} := (a, b) \in \mathbf{RLWE}_z^{t/Q}(m)$, $\mathsf{RLWE.ConstMul}(\mathsf{ct}, p)$ yields $(a \cdot p, b \cdot p) \in \mathbf{RLWE}_z^{t/Q}(m \cdot p)$.

This property enables efficient multiplication in a Single-Instruction-Multiple-Data (SIMD) manner: We can express a vector $(p_1, \ldots, p_{N-1}) \in \mathbb{Z}_t^N$ in a single polynomial as $p(X) = \sum_{i=0}^{N-1} p_i \cdot X^i \in \mathcal{R}_t$ and multiply it by an RLWE ciphertext encrypting a constant $m \in \mathbb{Z}_t$. The resulting ciphertext encrypts $p_i \cdot m$ for each $i \in [0, N)$ when m is a constant polynomial. Notably, this SIMD-like multiplication is far more efficient than performing N separate LWE multiplications.

For InstantOMR, the message m is in $\{0, 1\}$, product resulting in either $\mathbf{RLWE}_z^Q(0)$ or $\mathbf{RLWE}_z^Q(p)$.

4 Oblivious Message Retrieval (OMR)

Since the system model of OMR is summarized in § 1, in this section, we recall the threat model, interfaces, and the paradigm for existing solutions.

Threat model. We assume a computationally-bounded adversary that can read all public information, including all board messages, all public keys, and all communication between the detector and the recipient. It can also generate new messages and post them on the board, as well as honestly new clue keys and induce other parties to generate messages addressed to those keys. For soundness and completeness, we require the detectors, senders, and recipients to be honest but curious; they may collude by sharing information (defer the discussion on correctness/integrity against malicious detectors to the full version). In regard to privacy, we let *all* parties in the systems be *malicious* and colluding, except for the sender and recipient whose privacy is the protection target.

In [28], the authors have also discussed a DoS threat model. In short, it allows the senders and recipients to behave arbitrarily while ensuring that correctness and soundness hold. In [27], the authors propose a general way to make a non-DoS-resistant OMR secure in this DoS threat model, which also applies to our construction (with moderate overhead, roughly $20\% - 30\%$). Therefore, similar to [26], we focus on the standard model, which is sufficient to achieve the DoS-resistant OMR using techniques in [27].

Definition. OMR has the following PPT algorithms:

- $\mathsf{pp} \leftarrow \mathsf{GenParam}(1^\lambda, \varepsilon_\mathrm{p}, \varepsilon_\mathrm{n})$: takes a security parameter λ, a false positive rate ε_p, a false negative rate ε_n (see our full version for terms), and outputs a public parameter pp.
- $(\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}})) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$: takes the public parameter pp; outputs a secret key sk and a public key pk consisting of a clue key $\mathsf{pk}_{\mathsf{clue}}$ and a detection key $\mathsf{pk}_{\mathsf{detect}}$.
- $c \leftarrow \mathsf{GenClue}(\mathsf{pp}, \mathsf{pk}_{\mathsf{clue}}, x)$: takes the public parameter pp, a clue key $\mathsf{pk}_{\mathsf{clue}}$, and a payload $x \in \mathcal{P}$ where $\mathcal{P} := \{0,1\}^P$ for some $P > 0$; outputs a clue $c \in \mathcal{C}$.
- $M \leftarrow \mathsf{Retrieve}(\mathsf{pp}, \mathsf{BB}, \mathsf{pk}_{\mathsf{detect}}, \bar{k})$: takes public parameter pp, a bulletin board $\mathsf{BB} = \{(x_1, c_1), \ldots, (x_\mathsf{D}, c_\mathsf{D})\}$ for size D, a detection key $\mathsf{pk}_{\mathsf{detect}}$, and a bound $\bar{k}$ on the number of pertinent messages to that recipient; outputs digest M.
- $\mathsf{PL} \leftarrow \mathsf{Decode}(\mathsf{pp}, M, \mathsf{sk})$: takes the public parameter pp, the digest M and the secret key sk; outputs either a decoded payload list $\mathsf{PL} \subset \mathcal{P}^k$ or an overflow indication $\mathsf{PL} = \mathsf{overflow}$.

Formal OMR definition also includes completeness, soundness, privacy, and compactness. Since our scheme focuses on concrete realization and efficiency improvement, we defer the formal definition to the full version.

Prior FHE-based OMR Constructions. We now briefly recap existing constructions of OMR using the model outlined in above. All existing single-server OMR schemes [26,28,30] follow this paradigm (also visualized in Fig. 1):

1. KeyGen**:** The recipient generates a key pair $(\mathsf{PKE.pk}, \mathsf{PKE.sk})$ using a lattice-based PKE scheme PKE, and sets its clue key as $\mathsf{pk}_{\mathsf{clue}} := \mathsf{PKE.pk}$. It also generates an FHE key pair $(\mathsf{FHE.pk}, \mathsf{FHE.sk})$ using some FHE scheme FHE, and sets its detection key as $\mathsf{pk}_{\mathsf{detect}} := (\mathsf{FHE.pk}, \mathsf{FHE.Enc}(\mathsf{PKE.sk}))$.
2. GenClue**:** The sender uses $\mathsf{pk}_{\mathsf{clue}} = \mathsf{PKE.pk}$ to encrypt a vector of ℓ ones: $\boldsymbol{v} := (1,1,\ldots,1) \in \{0,1\}^{\ell}$. It computes the clue ciphertext as $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{clue}}, \boldsymbol{v})$. At a high level, this ciphertext decrypts to $\boldsymbol{v}$ under PKE.sk if the message is pertinent to the recipient, and to some $\boldsymbol{v}' \in \{0,1\}^{\ell}$, where $\boldsymbol{v}' \neq \boldsymbol{v}$, with overwhelming probability if it is not. This is because an honestly generated secret key not matching $\mathsf{pk}_{\mathsf{clue}}$ will fail to correctly decrypt the ciphertext. In other words, this ciphertext serves to indicate whether the message is intended for the recipient: the recipient could decrypt each clue and compare the result with $\boldsymbol{v}$ to determine pertinency.
3. Retrieve**:** The goal of this phase is to let the detector check message pertinency on behalf of the recipient. This step is the primary efficiency bottleneck and our main focus:
   (a) The detector uses $\mathsf{pk}_{\mathsf{detect}}$ to homomorphically decrypt each clue ciphertext ct_i, producing $\mathsf{decRes}_i = \mathsf{FHE.Enc}(\boldsymbol{v}'_i)$.
   (b) It then homomorphically tests whether $\boldsymbol{v}'_i = \boldsymbol{v}$, outputting $\mathsf{PV}[i] = \mathsf{FHE.Enc}(1)$ if so, and $\mathsf{FHE.Enc}(0)$ otherwise. The resulting vector PV of length D (called the *pertinency vector*) indicates which messages are relevant to this recipient.
   (c) Finally, using PV and the payloads from the database BB, the detector homomorphically generates a digest M (encrypted under FHE.sk), such that only the payloads of $\mathsf{BB}[i]$ with $\mathsf{PV}[i] = \mathsf{FHE.Enc}(1)$ are included (and encoded). This is done through homomorphic encoding, which we describe in detail later.
4. Decode**:** In this final step, the recipient simply decrypts M using FHE.sk to recover the relevant payloads.

5 Our Construction, InstantOMR

We now dive directly into InstantOMR and expand the high-level summary explained in § 2 (parameters summarized in Table 7 as reference).

5.1 Setup

We start with the setup. In particular, we discuss the keys generated by the recipient and specify the public parameters.

Secret key generation. As mentioned in the technical overview § 2, InstantOMR employs a two-layer bootstrapping framework, where each layer employs different parameters to perform different functions. The TFHE scheme, which utilizes both LWE and RLWE simultaneously (see § 3.1.1), typically requires an LWE secret key and an RLWE secret key. Thus, during the setup, we generate four keys, an LWE secret key and an RLWE secret key per layer.

The recipient executes KeyGen with the PKE parameter pp, generating a secret key sk and a public key pk. The secret key sk comprises the LWE secret keys $\boldsymbol{s}_1 \in \mathbb{Z}^{n_1}$ and $\boldsymbol{s}_2 \in \mathbb{Z}^{n_2}$, along with the RLWE secret keys $z_1 \in \mathcal{R}/(X^{N_1}+1)$ and $z_2 \in \mathcal{R}/(X^{N_2}+1)$, used to generate the public keys below.[8]

Clue key generation. The public key pk includes a clue key $\mathsf{pk}_{\mathsf{clue}}$ and a detection key $\mathsf{pk}_{\mathsf{detect}}$, where $\mathsf{pk}_{\mathsf{clue}}$ is used by the sender, and $\mathsf{pk}_{\mathsf{detect}}$ is used by the detector.

We start with the clue key $\mathsf{pk}_{\mathsf{clue}}$. For better performance, similar to [30], $\mathsf{pk}_{\mathsf{clue}}$ is a RLWE public key generated by treating the LWE secret key $\boldsymbol{s}_1$ in $\mathcal{R}$, as shown in Alg 1.

Algorithm 1 Generate Clue Key GenClueKey

Input: secret key $\boldsymbol{s}_1 \in \mathbb{Z}^{n_1}$ (viewed as $s_1 \in \mathcal{R}_{q_1,n_1}$)
Input: parameters $(n_1, q_1, \sigma_{ck}) \in \mathsf{pp}$
Output: clue key $\mathsf{pk}_{\mathsf{clue}}$

1: Sample $a \leftarrow \mathcal{R}_{q_1,n_1}$ randomly
2: Sample $e \in \mathcal{R}_{q_1,n_1}$ according to $\mathcal{N}(0, \sigma_{ck}^2)$
3: **return** $\mathsf{pk}_{\mathsf{clue}} = (a, b = as_1 + e) \in (\mathcal{R}_{q_1,n_1}, \mathcal{R}_{q_1,n_1})$

Detection key generation. The *detection key* $\mathsf{pk}_{\mathsf{detect}}$ is slightly more involved. It comprises two bootstrapping keys for our two-layers of bootstrapping. In addition, it includes one key-switching key for the first layer and one trace key for the second layer. The goal of the key-switching key and trace key will become clearer as we proceed, but in summary, they are used to improve our concrete efficiency. These keys enable the detector to perform retrieval operations on behalf of the recipient and to generate a digest composed of messages pertinent to the recipient. The detection key generation algorithm is detailed in Alg 2.

Algorithm 2 Generate Detection Key GenDetectKey

Input: public parameters pp
Input: secret key $\boldsymbol{s}_1 \in \mathbb{Z}^{n_1}, \boldsymbol{s}_2 \in \mathbb{Z}^{n_2}, z_1 \in \mathcal{R}/(X^{N_1}+1), z_2 \in \mathcal{R}/(X^{N_2}+1)$
Output: detection key $\mathsf{pk}_{\mathsf{detect}}$

1: $\mathsf{BSK}_1 \leftarrow \mathsf{TFHE.GenBootKey}(\boldsymbol{s}_1, z_1)$
2: $\mathsf{KSK}_{z_1 \to \boldsymbol{s}_2} \leftarrow \mathsf{FHE.GenKeySwitchingKey}(\boldsymbol{z}_1, \boldsymbol{s}_2)$
3: $\mathsf{BSK}_2 \leftarrow \mathsf{TFHE.GenBootKey}(\boldsymbol{s}_2, z_2)$
4: $\mathsf{TraceKey} \leftarrow \mathsf{FHE.GenTraceKey}(z_2)$
5: Let $\mathsf{pk}_{\mathsf{detect}} = (\mathsf{BSK}_1, \mathsf{KSK}_{z_1 \to \boldsymbol{s}_2}, \mathsf{BSK}_2, \mathsf{TraceKey})$
6: **return** $\mathsf{pk}_{\mathsf{detect}}$ ▷ See § 3.1 for details about each key

[8] Once the public key has been generated, the recipient need only retain $z_2 \in \mathcal{R}/(X^{N_2}+1)$ to decode the digest for Step 4.

When the recipient needs to retrieve the messages addressed to them on the board, they first transmit the detection key to the detector. Looking ahead, as we will discuss in § 5.2, the detector performs a series of homomorphic operations with the detection key to generate a digest of the pertinent messages, which is then sent back to the recipient. Upon receiving the digest, the recipient can decode it and get the payloads of the pertinent messages.

The complete InstantOMR.KeyGen protocol, encompassing the generation of secret keys, clue keys, and detection keys, is detailed in line 4 of Alg 8.

Now that the algorithms for generating all necessary keys have been introduced, we will proceed to explain how the sender utilizes the clue key to generate clues for their messages, and how the detector employs the detect key to assist the recipient in retrieving their pertinent messages.

Clue generation. Now we discuss the sender's clue generation after obtaining $\mathsf{pk}_{\mathsf{clue}}$ from the intended recipient. At a high-level, similar to [30], instead of using ℓ LWE ciphertexts each encrypting a single 1 (as described in § 2 for simplicity), we use RLWE to encrypt ℓ 1's in a single RLWE ciphertext to reduce the clue size. We formalize this process in Alg 3.

Algorithm 3 Generate Clue GenClue

Input: clue key $\mathsf{pk}_{\mathsf{clue}} = (a,b) \in (\mathcal{R}_{q_1,n_1}, \mathcal{R}_{q_1,n_1})$
Input: number of encrypted bits per clue $\ell \in \mathsf{pp}$
Output: clue $\mathbf{RLWE}_{\boldsymbol{s}_1}^{p_1/q_1}(\sum_{i=0}^{\ell-1} X^i) = (a',\ b'')$ with the length of b'' is ℓ

1: Sample $r \leftarrow \mathcal{R}_{q_1,n_1}$ with binary coefficients randomly
2: Sample $e_0, e_1 \in \mathcal{R}_{q_1,n_1}$ according to $\mathcal{N}(0, \sigma_{ck}^2)$
3: Let $(a', b') = \left(ra + e_0,\ rb + e_1 + \frac{q_1}{p_1}(\sum_{i=0}^{\ell-1} X^i)\right)$ ▷ Take only the first ℓ coefficients of b'.
4: Let $b'' = \sum_{i=0}^{\ell-1} b'_i X^i$
5: **return** (a', b'')

5.2 Retrieval Algorithms Run by the Detector

As mentioned in the § 2, InstantOMR performs the "first-layer bootstrapping" (Step 3a in § 4) for "homomorphic decryption" and the "second-layer bootstrapping" for "homomorphic checking" (Step 3b in § 4). Then, it performs "homomorphic encoding" via standard RLWE operations to leverage the power of SIMD.

5.2.1 The First-layer Bootstrapping

The focus now shifts to the first-layer bootstrapping, which is used to "homomorphically decrypt" the clue string. In § 5.1, we see that the sender generates a ciphertext $\mathsf{ct} = \mathbf{RLWE}(m(X) = \sum_{i=0}^{\ell-1} X^i)$ which encrypts ℓ 1's as clue string (since the clue only contains the first ℓ coefficients of the ring element b''). The detector performs $\mathsf{FHE.Extract}(\mathsf{ct}, i)_{i\in[0,\ell)}$ (recalled in § 3.1.1) to get ℓ LWE ciphertexts $c_0, \ldots, c_{\ell-1}$ encrypt 1. Then we execute the "first-layer bootstrapping" on every LWE ciphertext respectively.

The TFHE homomorphic encryption scheme supports functional bootstrapping (see § 3.1.1). Recall that the ciphertexts $c_0, \ldots, c_{\ell-1}$ extracted from the clue ciphertext, all of which decrypt to 1 for pertinent messages, and decrypt to $[0, 2, \ldots, p_1 - 1]$ for impertinent messages. Thus, ideally, the following function distinguishes pertinent and impertinent messages: $f(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{if } x = 0 \text{ or } x \in [2,\ p_1) \end{cases}$

Intuitively, all the pertinent messages, after bootstrapping using this function, have the corresponding output ciphertexts all encrypting 1's, while all the impertinent messages have ciphertexts encrypting all 0's. This then allows us to distinguish these two kinds of messages. However, this function cannot be straightforwardly implemented: as recalled in § 3.1.1, the functions that can be executed in TFHE's functional bootstrapping must satisfy the condition: $f(m + p_1/2) = -f(m)$.[9] If we desire the output to be $\mathbf{LWE}(1)$ when the input is $\mathbf{LWE}(1)$, the function actually being executed is as follows:

$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ -1 & \text{if } x = 1 + p_1/2\ . \\ 0 & \text{otherwise} \end{cases}$$

Then, if the message is pertinent to the recipient, the output of line 5 Alg 4 should yield ℓ ciphertexts in $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(1)$. Conversely, if the message is impertinent, the ciphertexts $c_0, \ldots, c_{\ell-1}$ are encrypted under another recipient's public clue key. Then $c_0, \ldots, c_{\ell-1}$ are computationally indistinguishable from random ciphertexts (i.e., uniform over $\mathcal{R}_{q_1} \times \mathbb{Z}_{q_1}^{\ell}$), which can thus be seen as a ciphertext encrypting a random value in $\mathbb{Z}_{p_1}$. Therefore, the output of the bootstrapping will be $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(1)$ with probability $\frac{1}{p_1}$, $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(-1)$ with probability $\frac{1}{p_1}$, and $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(0)$ with probability $1 - \frac{2}{p_1}$.

Next, as intuitively explained in § 2, we aggregate all ℓ ciphertexts. For a pertinent message, this results in a ciphertext of $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(\ell)$. For an impertinent message, we obtain a ciphertext of $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(r)$, where $-\ell \le r \le \ell - 1$, with probability $1 - (\frac{1}{p_1})^{\ell}$, or a ciphertext of $\mathbf{LWE}_{\boldsymbol{z}_1}^{p_2/Q_1}(\ell)$ with probability $(\frac{1}{p_1})^{\ell}$. By selecting appropriate values for p_1 and ℓ, $(\frac{1}{p_1})^{\ell}$ can be made negligible.

The last step in Alg 4 transforms the resulting ciphertext to use the secret key $\boldsymbol{s}_2$ and modulus q_2 instead. The reduced dimension of $\boldsymbol{s}_2$ is to accelerate the second-layer bootstrapping.

5.2.2 The Second-layer Bootstrapping

Now, we obtain $\mathbf{LWE}_{\boldsymbol{s}_2}^{p_2/q_2}(\ell)$ for the pertinent message with overwhelming probability, but only negligible probability for

[9] We defer the discussion on full-domain functional bootstrapping to the full version.

Algorithm 4 FirstLayerBoot

Input: clue: $\mathsf{ct} = (a, b)$ where b's dimension is ℓ
Input: $\mathsf{BSK}_1, \mathsf{KSK}_{z_1 \to s_2} \in \mathsf{pk}_{\mathsf{detect}}$
Input: $p_2 > 4\ell$ being a power-of-2
Output: $\mathbf{LWE}_{\boldsymbol{s}_2}^{p_2/q_2}(\ell)$ for pertinent message, $\mathbf{LWE}_{\boldsymbol{s}_2}^{p_2/q_2}(r)_{r\in[-\ell,\,\ell-1)}$ for impertinent message

1: $t \leftarrow \mathbf{LWE}_{z_1}^{p_2/Q_1}(0)$ ▷ A trivial encryption of 0
2: **for** $i = 0$ to $\ell - 1$ **do**
3: $\quad c_i = \mathsf{FHE.Extract}(\mathsf{ct}, i)$
4: $\quad t_i \leftarrow \mathsf{TFHE.Boot}(c_i, f, \mathsf{BSK}_1)$ ▷ $\mathbf{RLWE}_{z_1}^{p_2/Q_1}$
5: $\quad t_i' \leftarrow \mathsf{FHE.Extract}(t_i, 0)$ ▷ $\mathbf{LWE}_{z_1}^{p_2/Q_1}$
6: $\quad t \leftarrow t + t_i'$
7: $t \leftarrow \mathsf{FHE.KS}(t, \mathsf{KSK}_{z_1 \to s_2})$ ▷ $\mathbf{LWE}_{\boldsymbol{s}_2}^{p_2/Q_1}$
8: **return** $\mathsf{FHE.MS}_{Q_1 \to q_2}(t)$ ▷ $\mathbf{LWE}_{\boldsymbol{s}_2}^{p_2/q_2}$

the impertinent message. Subsequently, we need to perform a "homomorphic checking" to determine whether the ciphertext encrypts ℓ as mentioned in § 2. To avoid the constraints imposed by the negacyclic function, the plaintext modulus p_2 must satisfy the condition $p_2/2 > 2\ell$.

We use the following function to bootstrap this ciphertext (again, negacyclic): $g(x) = \begin{cases} 1 & \text{if } x = \ell \\ -1 & \text{if } x = \ell + p_2/2 \\ 0 & \text{otherwise} \end{cases}$.

It's easy to see that $p_2 - \ell = p_2/2 + p_2/2 - \ell > p_2/2 + \ell$. Hence, the input ciphertext encrypts the message $x \in [-\ell, \ell] = [p_2 - \ell, p_2) \cup [0, \ell]$ is not in $\{\ell + p_2/2\}$, which means that the functional bootstrapping over g would never output $\mathsf{Enc}(-1)$.

After this bootstrapping procedure, we obtain a ciphertext $\mathbf{RLWE}_{z_2}^{p_3/Q_2}(m(x) = m_0 + \sum_{i=1}^{N_2-1} r_i X^i)$ whose constant-term m_0 represents whether the message is pertinent to the recipient while r_i's are irrelevant values. To further compute, we need to obtain $\mathbf{RLWE}_{z_2}^{p_3/Q_2}(m_0)$ (i.e., the constant polynomial, zeroing out all the other coefficients), allowing better performance in the homomorphic encoding step as discussed in § 2 (and detailed § 5.2.3).

The classic approach in [16, Section 6.2] achieves what we need with a large sized key and more ring multiplications. To minimize key size and runtime, we employ the $\mathsf{FHE.HomTrace}$ function, which uses only $(\log N_2)\ell_{auto}$ RLWE ciphertexts as $\mathsf{TraceKey}$. This greatly reduced the size of the key. Let the output of the $\mathsf{TFHE.Boot}$ be $\mathbf{RLWE}(m + \sum_{i=1}^{N_2-1} m_i X^i)$. As discussed in § 3.1.2, $\mathsf{FHE.HomTrace}$ will then yield $\mathbf{RLWE}(m)$, where m equals either 0 or 1, 1 for the pertinent message and 0 for the impertinent message. In the Alg 5, we show our second layer of bootstrapping.

Algorithm 5 SecondLayerBoot

Input: First-layer result: $c = \mathbf{LWE}_{\boldsymbol{s}_2}^{p_2/q_2}(r)_{r\in[-\ell,\,\ell]}$
Input: $\mathsf{BSK}_2, \mathsf{TraceKey} \in \mathsf{pk}_{\mathsf{detect}}$
Output: $\mathbf{RLWE}_{z_2}^{p_3/Q_2}(1)$ for pertinent message, $\mathbf{RLWE}_{z_2}^{p_3/Q_2}(0)$ for impertinent message

1: $t \leftarrow \mathsf{TFHE.Boot}(c, g, \mathsf{BSK}_2)$ ▷ $\mathbf{RLWE}_{z_2}^{p_3/Q_2}$
2: $t' \leftarrow \mathsf{FHE.HomTrace}(t, \mathsf{TraceKey})$ ▷ $\mathbf{RLWE}_{z_2}^{p_3/Q_2}$
3: **return** t'

5.2.3 Encode the pertinent message indices

After performing "homomorphic decryption" and "homomorphic checking" on clue string of each message on the board, the detector subsequently performs "homomorphic encoding" (Step 3c in § 4) to generate a digest, which includes *encoded pertinent indices* and *encoded pertinent payloads*. In this section, we will elaborate on how to encode the pertinent message indices for the recipient.

In contrast to single-message LWE encryption, RLWE enables multiplications in a SIMD manner (see § 3.1.3) when we encrypt a constant polynomial. This is why we output an RLWE ciphertext encrypting a constant polynomial in Alg 5.

For D messages, let the output ciphertexts of the two-layer bootstrapping be $\mathsf{PV} := (\mathsf{ct}_0, \ldots, \mathsf{ct}_{\mathsf{D}-1})$ which indicates whether the D payloads $\mathsf{PLD} := (\mathsf{pld}_0, \ldots, \mathsf{pld}_{\mathsf{D}-1})$ are pertinent. Recall that each ct_i has plaintext modulus p_3 (e.g., $\mathsf{ct}_i = \mathbf{RLWE}(m^{(i)})$ where $m^{(i)} \in \{0, 1\} \subset \mathcal{R}_{p_3}$).

As discussed in § 2, to encode the pertinent indices, we initialize many *buckets* consisting of the so-called *accumulators* and *counters*, and then randomly assign each message to one of these buckets. Now we discuss how we encode them using coefficients and how the random assignment works.

First, we decompose each message index k ($k \in [0, \mathsf{D})$) into its base-p_3 digits: $k = \sum_{i=0}^{d-1} k_i \cdot {p_3}^i$, where d is the smallest integer satisfying ${p_3}^d \geq \mathsf{D}$. For each index k, we construct a corresponding polynomial: $\mathsf{Poly}(k) = k_0 + k_1 \cdot X + \cdots + k_{d-1} \cdot X^{d-1} + 1 \cdot X^d$, where the final term, 1, acts as a counter.[10]

In total, we initialize N_s buckets (N_s choice discussed in § 5.4.2). We call these N_s buckets a *segment*. Each segment takes $(d+1) \cdot N_s$ coefficients to encode. WLOG, assume $(d+1) \cdot N_s < N_2$, the number of coefficients available per ciphertext (otherwise, simply extend to use more ciphertexts).

Next, for the k-th message, we generate a random shift r_k ensuring $r_k < N_s$. We then multiply the polynomial by $X^{(d+1)\cdot r_k}$, producing: $\mathsf{Shifted}(k) = k_0 \cdot X^{(d+1)\cdot r_k} + k_1 \cdot X^{(d+1)\cdot r_k + 1} + \cdots + k_{d-1} \cdot X^{(d+1)\cdot r_k + d - 1} + 1 \cdot X^{(d+1)\cdot r_k + d}$. This means that the k-th index is rotated to the r_k-th bucket.

For encryption, we multiply $\mathsf{Shifted}(k)$ by the ciphertext ct_k. If the message is pertinent, this results in:

[10] Following prior works, we assume the total number of pertinent messages $k < p_3$, which is true for the parameter choices in prior works [26, 28, 30].

$\mathbf{RLWE}\begin{pmatrix} k_0 \cdot X^{(d+1)\cdot r_k} + k_1 \cdot X^{(d+1)\cdot r_k+1} + \dots \\ + k_{d-1} \cdot X^{(d+1)\cdot r_k+d-1} + 1 \cdot X^{(d+1)\cdot r_k+d} \end{pmatrix}$. Otherwise, the output is simply $\mathbf{RLWE}(0)$.

Aggregating all resulting RLWE ciphertexts, we obtain:

$$\mathbf{RLWE}\begin{pmatrix} \cdots \\ + a_0 \cdot X^{(d+1)\cdot r_a} + a_1 \cdot X^{(d+1)\cdot r_a+1} + \dots \\ + a_{d-1} \cdot X^{(d+1)\cdot r_a+d-1} + a_d \cdot X^{(d+1)\cdot r_a+d} \\ + \dots \end{pmatrix}.$$

Here shows the r_a-th bucket with coefficients a_0 to a_d.

- If $a_d = 0$ (the bucket counter is 0), this indicates that no index was added to positions a_0 through a_{d-1}.
- If $a_d = 1$, this signifies that exactly one unique index was accumulated across a_0 to a_{d-1}.
- If $a_d \geq 2$, multiple indices collide in this bucket, in which case we discard this bucket and check subsequent buckets.

The original index a (detected as pertinent) can be recovered via: $a \leftarrow \sum_{i=0}^{d-1} a_i \cdot {p_3}^i \quad (\text{if } a_d = 1)$.

Due to possible collision, the detector needs to repeat the aforementioned steps multiple times independently for the recipients to recover all indices. The number of repetitions is denoted by ℓ_{max} (see § 5.4.2 for how ℓ_{max} is determined).

The aforementioned process describes the method of encoding a single pertinent indices vector using RLWE. In practice, we can partition the polynomial space into multiple segments, each of which encodes a group of buckets, since $(d+1)N_s \ll N_2$ (which is indeed the case for parameters in § 6). The algorithm of index encoding is presented in Alg 6.

Algorithm 6 EncodePertinentIndices

Input: Homomorphic checking result: $\mathsf{PV} := (\mathsf{ct}_0, \dots, \mathsf{ct}_{\mathsf{D}-1}) \in \mathbf{RLWE}_{z_2}^{p_3/Q_2}(0 \text{ or } 1)$
Input: Segment count ℓ_s
Input: Buckets count per segment N_s, satisfying $(d+1)\cdot N_s \cdot \ell_s < N_2$ (otherwise, view $\alpha > 1$ ciphertexts as a single ciphertext and have $(d+1)\cdot N_s \cdot \ell_s < \alpha \cdot N_2$)
Input: Round bound ℓ_{max}
Output: Encoded pertinent indices res

for $j = 1$ to $\lceil \ell_{max}/\ell_s \rceil$ **do**
  $\mathsf{Acc} \leftarrow \mathbf{RLWE}_{z_2}^{p_3/Q_2}(0)$ ▷ A trivial encryption of 0
  for $k = 0$ to $\mathsf{D}-1$ **do**
    Parse $k = \sum_{i=0}^{d-1} k_i \cdot {p_3}^i$
    $\mathsf{Poly}(k) = k_0 + k_1 \cdot X + \cdots + k_{d-1} \cdot X^{d-1} + 1 \cdot X^d$
    $\forall i \in [0, \ell_s)$, sample r_i randomly such that $r_i < N_s$
    $\mathsf{Shifted}(k) = \sum_{i=0}^{\ell_s-1} \mathsf{Poly}(k) \cdot X^{(d+1)\cdot(iN_s+r_i)}$
    $\mathsf{Acc} += \mathsf{ConstMul}(\mathsf{ct}_k, \mathsf{Shifted}(k))$ ▷ See § 3.1.3
  Push Acc into res (empty before the loop)
return res

5.2.4 Encode the pertinent message payloads

With previous step, the recipient can recover all pertinent messages' indices. Now, we will discuss encoding payloads.

Recall that $\bar{k}$ is the (expected) bound on the number of pertinent messages, and let $m > \bar{k}$ (see § 5.4.2 for m). We generate a weight matrix $\mathsf{W} \in \mathbb{Z}_{p_3}^{m\times D}$ with a random seed.

Subsequently, we decompose each payload into digits with p_3 as the basis, $t_0 + t_1 \cdot p_3 + t_2 \cdot {p_3}^2 + \cdots$. This is encoded into a polynomial $\mathsf{pld}_i = t_0 + t_1 \cdot X + t_2 \cdot X^2 + \cdots \in \mathcal{R}_{p_3,N_2}$. We obtain a column vector $\mathsf{PLD} = [\mathsf{pld}_0, \cdots, \mathsf{pld}_{D-1}] \in \mathcal{R}_{p_3,N_2}^{\mathsf{D}}$ encoding all payloads.

We then compute the weighted payload:

$$\begin{aligned} \mathsf{Wp} &= \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,D-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m-1,0} & w_{m-1,1} & \cdots & w_{m-1,D-1} \end{bmatrix} \otimes \mathsf{PLD} \\ &= \begin{bmatrix} w_{0,0}\cdot\mathsf{pld}_0 & w_{0,1}\cdot\mathsf{pld}_1 & \cdots & w_{0,D-1}\cdot\mathsf{pld}_{D-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m-1,0}\cdot\mathsf{pld}_0 & w_{m-1,1}\cdot\mathsf{pld}_1 & \cdots & w_{m-1,D-1}\cdot\mathsf{pld}_{D-1} \end{bmatrix} \\ &\in (\mathcal{R}_{p_3,N_2})^{m\times D} \end{aligned}$$

Next, we multiply Wp and $\mathsf{PV} := (\mathsf{ct}_0, \dots, \mathsf{ct}_{\mathsf{D}-1})$ obtained from Alg 5, yielding the encoded pertinent message payloads, denoted as $\mathsf{Cp} \in \mathsf{RLWE}^m$ (a column vector of m RLWE's).

$$\begin{aligned} \mathsf{Cp} &= \begin{bmatrix} wp_{0,0} & wp_{0,1} & \cdots & wp_{0,D-1} \\ \vdots & \vdots & \ddots & \vdots \\ wp_{m-1,0} & wp_{m-1,1} & \cdots & wp_{m-1,D-1} \end{bmatrix} \cdot \mathsf{PV} \\ &= \begin{bmatrix} \sum_{i=0}^{D-1} \mathsf{RLWE.ConstMul}(\mathsf{ct}_i, wp_{0,i}) \\ \vdots \\ \sum_{i=0}^{D-1} \mathsf{RLWE.ConstMul}(\mathsf{ct}_i, wp_{m-1,i}) \end{bmatrix} \in \mathsf{RLWE}^m \end{aligned}$$

The seed and Cp are then both sent to the recipient. Recall that the recipient has already ascertained which indices are pertinent. With k ($k \leq \bar{k}$) such relevant messages, the recipient removes all columns from matrix W (recoverable by a random seed) that do not correspond to these indices, resulting in a new weight matrix $\mathsf{W}' \in \mathbb{Z}_{p_3}^{m\times k}$. It now suffices to solve the system of equations $\mathsf{W}' \cdot x = \mathsf{FHE.Dec}(\mathsf{Cp})$ to obtain x, thereby retrieving the payloads of the messages on the board that are pertinent to the recipient. A formal description of payload encoding is given in Alg 7.

Algorithm 7 EncodePertinentPayloads

Input: Homomorphic checking result: $\mathsf{PV} := (\mathsf{ct}_0, \dots, \mathsf{ct}_{\mathsf{D}-1}) \in \mathbf{RLWE}_{z_2}^{p_3/Q_2}(0 \text{ or } 1)$
Input: Payloads list $\mathsf{PLD} = [\mathsf{pld}_0, \mathsf{pld}_1, \cdots, \mathsf{pld}_{D-1}] \in \mathcal{R}_{p_3,N_2}^{\mathsf{D}}$
Input: Parameter $(p_3, m) \in \mathsf{pp}$
Output: seed and payloads combinations Cp

Sample a seed and a weight matrix $\mathsf{W} \in \mathbb{Z}_{p_3}^{m\times D}$ with seed
Construct diagonal matrix $\mathsf{diag}(\mathsf{PLD})$ from PLD, where $\mathsf{diag}(\mathsf{PLD})[i,i] = \mathsf{PLD}[i]$.
$\mathsf{Wp} \leftarrow \mathsf{W} \cdot \mathsf{diag}(\mathsf{PLD}) \in (\mathcal{R}_{p_3,N_2})^{m\times D}$
$\mathsf{Cp} \leftarrow \mathsf{Wp} \cdot \mathsf{PV} \in \mathsf{RLWE}^m$
return (seed, Cp)

5.3 Our Full Construction

The full construction of InstantOMR is shown in Alg 8.

Theorem 5.1. *Let $\lambda > 0, \mathsf{D} > 0, 1 > \varepsilon_n, \varepsilon_p > 0, k < p_3$, Alg 8 is an Oblivious Message Retrieval scheme, assuming the hardness of RLWE assumption (with circular security).*

Proof. Completeness: A pertinent message is successfully retrieved if

1. The clue RLWE ciphertext decrypts correctly using TFHE bootstrapping.
2. The second layer of bootstrapping correctly maps all pertinent clues to 1 and the rest to 0.
3. The homomorphic encoding is done correctly without error.
4. No accumulator counter overflow.
5. The pertinent indices are retrieved correctly via decoding, given no counter overflow.
6. All the linear combinations are linearly independent.

The first condition is guaranteed by condition 5 in § 5.4.2, except for false negative rate ε_n. For the second and the third conditions, we need that TFHE and the following RLWE homomorphic operations are done currently, which happens unless the noise overflow, which happens with negligible probability, given the noise analysis in § 5.4, as also guaranteed by condition 2 in § 5.4.2. The fourth condition is guaranteed by $k < p_3$. The fifth condition is guaranteed by $1 - \prod_{i=1}^{\bar{k}} (1 - (\frac{1}{N_s})^{\ell_{max}}) = \mathsf{negl}(\lambda)$. The sixth condition is guaranteed by $\prod_{i=m-\bar{k}+1}^{m} (1 - 1/{p_3}^i) = 1 - \mathsf{negl}(\lambda)$. The last two are set in § 5.4.2.

Soundness: Soundness follows directly from (1) the correctness (as guaranteed by condition 2 in § 5.4.2) of TFHE bootstrapping, and (2) ${1/p_1}^{\ell} \leq \varepsilon_p$ (as guaranteed by condition 4 in § 5.4.2).

Privacy: We argue privacy via a hybrid argument.

- Hyb_0 : our construction.
- Hyb_1 : $\mathsf{BSK}_1, \mathsf{BSK}_2$ (Alg 2) are replaced with random ring elements in ring $\mathcal{R}_{N_1,Q_1}$ and $\mathcal{R}_{N_2,Q_2}$ respectively.
- Hyb_2 : $\mathsf{pk}_{\mathsf{clue}}$ (Alg 1) is replaced with two uniformly random $\mathcal{R}_{q_1,n_1}$ elements.
- Hyb_3 : clue $c = (a', b'')$ (Alg 3) is replaced with a uniformly random $\mathcal{R}_{q_1,n_1}$ element and a uniformly random $\mathbb{Z}_{q_1}^{\ell}$ vector.

It is easy to see that Hyb_3 is trivially secure because the clue does not include any information about the recipient. Hyb_2 and Hyb_3 are indistinguishable by the RLWE assumption, since the clue is simply a (truncated) RLWE sample given that the clue keys are two uniformly random $\mathcal{R}_q$ elements. Hyb_2 and Hyb_1 are indistinguishable by the RLWE assumption, since the clue is exactly an RLWE sample. Lastly, Hyb_1 and Hyb_0 are indistinguishable by the security guarantee of TFHE (which is guaranteed by RLWE with circular security [16]).

Compactness: As shown in § 5.4, the noise growth is $O(\mathsf{D})$, and thus the digest size is only linear in $\log(\mathsf{D})$. Therefore, the digest size for a single pertinent message is payload size times N (the ring dimension, which is polynomial in λ the security parameter) times ciphertext modulus Q plus index size times N times Q. Then, for k pertinent messages, there are at most $\tilde{O}(k + \varepsilon_p \cdot \mathsf{D})$ messages, and by [28], the encoding blowup is $\mathsf{poly}(k + \varepsilon_p \cdot \mathsf{D}, \lambda)$. Thus, in total, for k messages, the digest size is $\mathsf{poly}(\lambda, (k + \varepsilon_p \cdot \mathsf{D}, \log(\mathsf{D})))$.

□

Algorithm 8 InstantOMR: Oblivious Message Retrieval

procedure InstantOMR.GenParam(1^{λ}, ε_p, ε_n)
  Choose pp according to strategy in § 5.4.2
  return pp
procedure InstantOMR.KeyGen(pp)
  Sample LWE secret key $\boldsymbol{s}_1 \in \mathbb{Z}^{n_1}$ and $\boldsymbol{s}_2 \in \mathbb{Z}^{n_2}$ according to pp
  Sample RLWE secret key $z_1 \in \mathcal{R}/(X^{N_1}+1)$ and $z_2 \in \mathcal{R}/(X^{N_2}+1)$ according to pp
  $\mathsf{pk}_{\mathsf{clue}} \leftarrow \mathsf{GenClueKey}(\mathsf{pp}, \boldsymbol{s}_1)$ ▷ Alg 1
  $\mathsf{pk}_{\mathsf{detect}} \leftarrow \mathsf{GenDetectKey}(\mathsf{pp}, \boldsymbol{s}_1, \boldsymbol{s}_2, z_1, z_2)$ ▷ Alg 2
  return $(\mathsf{sk} = z_2, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}}))$
procedure InstantOMR.GenClue(pp, $\mathsf{pk}_{\mathsf{clue}}$)
  return $\mathsf{GenClue}(\mathsf{pp}, \mathsf{pk}_{\mathsf{clue}})$ ▷ Alg 3
procedure InstantOMR.Retrieve(pp, BB, $\mathsf{pk}_{\mathsf{detect}}$, $\bar{k}$)
  Parse $\mathsf{BB} = \{(\mathsf{pld}_0, \mathsf{ct}_0), \cdots, (\mathsf{pld}_{\mathsf{D}-1}, \mathsf{ct}_{\mathsf{D}-1})\}$
  Let payloads list $\mathsf{PLD} = \{\mathsf{pld}_0, \cdots, \mathsf{pld}_{\mathsf{D}-1}\}$
  $\mathsf{PV} = []$
  for $i = 0$ to $\mathsf{D}-1$ **do**
    $\mathsf{ct}'_i \leftarrow \mathsf{FirstLayerBoot}(\mathsf{pp}, \mathsf{ct}_i, \mathsf{pk}_{\mathsf{detect}})$ ▷ Alg 4
    $\mathsf{ct}''_i \leftarrow \mathsf{SecondLayerBoot}(\mathsf{ct}'_i, \mathsf{pk}_{\mathsf{detect}})$ ▷ Alg 5
    Push ct''_i into PV
  $\mathsf{digest}_{\mathsf{indices}} \leftarrow \mathsf{EncodePertinentIndices}(\mathsf{PV}, \mathsf{pp})$ ▷ Alg 6
  $(\mathsf{seed}, \mathsf{digest}_{\mathsf{payloads}}) \leftarrow \mathsf{EncodePertinentPayloads}(\mathsf{PV}, \mathsf{PLD}, \mathsf{pp})$ ▷ Alg 7
  return $\mathsf{digest} = (\mathsf{seed}, \mathsf{digest}_{\mathsf{indices}}, \mathsf{digest}_{\mathsf{payloads}})$
procedure InstantOMR.Decode(digest, $\mathsf{sk} = z_2$)
  Generate $\mathsf{W} \in \mathbb{Z}_{p_3}^{m \times D}$ with seed
  Decode $\mathsf{FHE.Dec}(z_2, \mathsf{digest}_{\mathsf{indices}})$, to obtain the k pertinent indices
  Extract $\mathsf{W}' \in \mathbb{Z}_{p_3}^{m \times k}$ from $\mathsf{W} \in \mathbb{Z}_{p_3}^{m \times D}$ with k pertinent indices columns
  Solve the equations of $\mathsf{W}' \cdot x = \mathsf{FHE.Dec}(z_2, \mathsf{digest}_{\mathsf{payloads}})$
  return x or if any of the step fail, **return** overflow

5.4 Parameter Analysis

There remains to discuss the parameter analysis, including noise analysis and the choice of parameters for encoding.

5.4.1 FHE Noise Analysis

We start with the former. We need to make sure that the noise budget is enough to complete the entire FHE circuit evaluation. This differs greatly from prior works since they use BFV, which can be more easily (heuristically) bounded using the multiplicative depth. When using TFHE, we need to bound the noise more carefully, as follows. The proof for why the following noise analysis holds is deferred to the full version for space reasons, and since the analysis follows [17, 41].

The noise of the clues. Let the noise variance of the clue key $\mathsf{pk}_{\mathsf{clue}}$ be σ_{ck}^2. The noise variance of the clues is $(n_1+1)\sigma_{ck}^2$.

First layer bootstrapping. Let the input RLWE ciphertext be parametrized by dimension n_1, plaintext modulus p_1, ciphertext modulus q_1 and the *binary* secret key $\boldsymbol{s}_1$. Let the bootstrapping key be parametrized by ring dimension N_1, ciphertext modulus Q_1, ternary secret key z_1, noise variance σ_{bsk1}^2, gadget basis B_1 and gadget length $d_1 \leq \lfloor \log_{B_1} Q_1 \rfloor$. Let the key switching key be parameterized by dimension n_2, ciphertext modulus Q_1, secret key $\boldsymbol{s}_2$, noise variance σ_{ks}^2, gadget basis B_{ks} and gadget length $d_{ks} \leq \lfloor \log_{B_{ks}} Q_1 \rfloor$. Then the noise variance of the output ciphertext of the first-layer bootstrapping is $\sigma_{out1}^2 = \frac{{q_2}^2}{{Q_1}^2}(\ell \cdot \sigma_{br1}^2 + \sigma_{ks}^2) + \sigma_{ms}^2$ where the key switching noise σ_{ks}^2 is $d_{ks} N_1 \frac{B_{ks}^2+2}{12}\sigma_{ksk}^2$ and the modulus switching noise σ_{ms}^2 is $\frac{n_2+2}{24}$. The total noise σ_{br1}^2 is shown below.

$$\sigma_{br1}^2 = d_1 n_1 N_1 \frac{B_1^2+2}{6}\sigma_{bsk1}^2 + n_1 \frac{{Q_1}^2 - B_1^{2d_1}}{24 B_1^{2d_1}}\left(1+\frac{N_1}{2}\right) + \frac{n_1 N_1}{16} + \frac{n_1}{16}$$

Second layer bootstrapping. Let the input LWE ciphertext be parametrized by dimension n_2, plaintext modulus p_2, ciphertext modulus q_2 and the *binary* secret key $\boldsymbol{s}_2$. Let the bootstrapping key be parametrized by ring dimension N_2, ciphertext modulus Q_2, ternary secret key z_2, noise variance σ_{bsk2}^2, gadget basis B_2 and gadget length $d_2 \leq \lfloor \log_{B_2} Q_2 \rfloor$. Let the trace key be parameterized by N_2, Q_2, z_2, noise variance σ_{ak}^2, gadget basis B_{ak} and gadget length $d_{ak} \leq \lfloor \log_{B_{ak}} Q_2 \rfloor$. Then the noise variance of the second layer bootstrapping is $\sigma_{out2}^2 = \sigma_{br2}^2 + \sigma_{tr}^2$. The σ_{br2}^2 bears similarity to the σ_{br1}^2 from the first layer, as illustrated below: $\sigma_{br2}^2 = d_2 n_2 N_2 \frac{B_2^2+2}{6}\sigma_{bsk2}^2 + n_2 \frac{{Q_2}^2 - B_2^{2d_2}}{24 B_2^{2d_2}}\left(1+\frac{N_2}{2}\right) + \frac{n_2 N_2}{16} + \frac{n_2}{16}$, where σ_{tr}^2 satisfy $\sigma_{tr}^2 \leq \frac{{N_2}^2-1}{3}\sigma_{auto}^2$ where $\sigma_{auto}^2 = d_{ak} N_2 \frac{B_{ak}^2+2}{12}\sigma_{ak}^2$.

Encode Pertinent Message Indices. Let the pertinent vector PV output by the second layer bootstrapping(see Alg 5) be parameterized by ring dimension N_2, ciphertext modulus Q_2, secret key z_2, noise variance σ_{out2}^2. Let the payloads count be D. Let the "Encode Pertinent Message Indices" be parameterized by segment count ℓ_s. Then the noise variance of the encoded pertinent message indices vector is $\mathsf{D}\ell_s\left(\lceil \log_{p_3} D \rceil + 1\right) {p_3}^2 \sigma_{out2}^2$.

Encode Pertinent Message Payloads. Let the pertinent vector PV output by the second layer bootstrapping(see Alg 5) be parameterized by ring dimension N_2, ciphertext modulus Q_2, secret key z_2, noise variance σ_{out2}^2. Let the payloads count be D and the payload size is $\tilde{n}$-bits. Let the "Encode Pertinent Message Payloads" be parameterized by combined payloads count per ciphertext ℓ_{cmb}. Then the noise variance of the encoded pertinent message payloads is $\mathsf{D}\ell_{cmb}\lceil \log(\tilde{n})/\log(p_3) \rceil {p_3}^2 \sigma_{out2}^2$.

5.4.2 Parameter-choosing Strategy

Now, we discuss how $\mathsf{InstantOMR}$ parameters are chosen.

Parameters for bootstrapping. We select the parameters satisfying the following property: (1) security has λ bits; (2) the final noise does not exceed $Q_2/2p_3$ except for $\mathsf{negl}(\lambda)$ probability (i.e., the noise does not exceed the noise bound after all steps using the analysis above) and the noise bound holds for each step (i.e., TFHE bootstrapping succeeds except with $\mathsf{negl}(\lambda)$ probability); (3) n_1, n_2, N_1, N_2 are selected to minimize the bootstrapping runtime; (4) $1/{p_1}^{\ell} \leq \varepsilon_{\mathrm{p}}$ for false positive rate ε_{p}; (5) the first level of bootstrapping maps a pertinent LWE ciphertext to 1 with probability $\varepsilon_{\mathrm{n}}/\ell$ for false negative rate ε_{n}. Concrete selection is shown in Table 1.

Parameters for homomorphic encoding. Lastly, we briefly discuss how we choose parameters for homomorphic encoding, specifically segment size N_s, number of segments ℓ_{max} (see § 5.2.3) and the number of linear combinations m (see § 5.2.4). For indices, we require $1 - \prod_{i=1}^{\bar{k}}(1 - (\frac{1}{N_s})^{\ell_{max}}) = \mathsf{negl}(\lambda)$ ([28, pp. 21]). For payloads, we require $\prod_{i=m-\bar{k}+1}^{m}(1 - 1/{p_3}^i) = 1 - \mathsf{negl}(\lambda)$ ([28, pp. 25]) for p_3 being a prime. The selection of ℓ_{max}, N_s, and m must ensure that the recipient can decode the digest with overwhelming probability. We select the largest ℓ_s, ℓ_{cmb} to minimize digest size. Similarly, they are summarized in Table 1.

5.5 Streaming Updates

As discussed in [28, Section 7.5], in practice, the applications may prefer streaming updates. In the streaming setting, a recipient uploads the detection key to the detector in advance (before making any retrieval requests). The detector processes messages on-the-fly as they arrive, and is ready to serve the digest at low additional cost when the recipient shows up.

While this setting is already discussed in [28], unfortunately, all the existing BFV-based OMR constructions *do not* fit in this setting well. Specifically, the existing OMR schemes accumulate N messages before performing the homomorphic retrieval process (for N being the ring dimension, either 32768 or 65536), since this maximizes their overall efficiency. However, if the recipient becomes online with only $t < N$ messages have accumulated, the detector then needs to perform the homomorphic retrieval with these t messages. In this case, no matter $t = 1$ or $t = N-1$, the recipient needs to wait for a long time (at least ∼52 seconds as shown in § 6).

InstantOMR, on the other hand, avoids the latency induced by large batches in this streaming setting, since it processes exactly one message at a time. Thus, no message accumulates. When the recipient comes online, the digest is ready since all messages are processed. In some cases, during the very second that the recipient is online, $t' \geq 1$ new messages arrive. Recipient wait time is $t' \cdot T$ where T is the time InstantOMR takes to process a single message. Concretely, for Bitcoin-scale application, $t' < 7$ [1], and the maximum recipient wait time is < 1 second for a single-core server, and < 100ms for a t'-core server (see § 6 for more details), compared to at least $\sim$52 seconds for prior works. Thus, our scheme is highly preferable for streaming update setting.[11]

For large D, a hybrid solution can be used (see § 6.2).

Remark 5.2. For all the discussion in § 5.5, we assume the recipient also sends $\bar{k}$ (i.e., the expected number of pertinent messages) in advance in addition to the detection key. Otherwise, since the homomorphic encoding is dependent on $\bar{k}$, the digest generation in the streaming process cannot be achieved (similar to all prior single-server OMR constructions [26, 28, 30]).

As discussed in [28], this seems to be a reasonable assumption that a client can estimate the upper bound on the number of messages they receive (and this bound can be conservative, only influencing the communication cost but not the computation cost).

6 Evaluation

6.1 Methodology

We implement the InstantOMR schemes with Primus-fhe [36] library. Then we compare its performance against PerfOMR and SophOMR, the state-of-the-art FHE-based OMR constructions. Our focus lies in optimizing scenarios where the total number of messages is relatively small (or the detector processes incoming messages on-the-fly), aiming to reduce the overall latency. We present comprehensive benchmark data for our scheme.

For PerfOMR [30] and SophOMR [26], we run their code on our instance. Note that for these prior works, if the total number of messages D is smaller than the ring dimension N of the underlying BFV scheme, their range-check component (i.e., Steps 3a and 3b in § 4) remains the same as for N messages, so we list range-check-time + $\mathsf{D}/N\times$ (total-time − range-check-time). Note that this is estimated to their favor, since there are other components that remain the same as for N messages, but sufficient to demonstrate our advantages.

Parameters. Table 1 presents a set of parameters chosen according to § 5.4.2 satisfying 128-bit security using the standard LWE-estimator [3]; and let $\bar{k} = k = 50$, payload size be 612 bytes, false negative probability be 2^{-30}, and false positive probability be 2^{-20} as the prior works [26,28,30]. For pertinent indices encoding, we partition the ring polynomial with dimension N_2 into ℓ_s groups of buckets, every group takes N_s buckets and every bucket takes $d+1$ coefficients, and ℓ_{max} groups of buckets will be encoded for correctness. These buckets can be encoded in $\lceil \ell_{max}/\ell_s \rceil$ RLWE ciphertexts. The linear combination count (for pertinent payloads encoding) is m, and we can encode ℓ_{cmb} combinations in one RLWE ciphertext at the same time.

Parameters							
Clue key	n_1	q_1	s_1	p_1	ℓ	σ_{ck}	
	512	2048	Binary	2^3	7	0.8293	
First bootstrapping	N_1	Q_1	z_1	B_1	d_1	σ_{bsk1}	p_2
	1024	134215681	Ternary	2^5	4	3.1859	2^5
Key switching	n_2	Q_{ks}	s_2	B_{ks}	d_{ks}	σ_{ksk}	
	670	134215681	Binary	2	26	2.0329×2^{10}	
Modulus switching	q_2						
	4096						
Second bootstrapping	N_2	Q_2	z_2	B_2	d_2	σ_{bsk2}	p_3
	2048	1125899906826241	Ternary	2^7	6	0.3908	257
HomTrace	N_2	Q_2	z_2	B_{ak}	d_{ak}	σ_{ak}	
	2048	1125899906826241	Ternary	2^2	24	0.3908	
Encode pertinent indices	ℓ_{max}	N_s	ℓ_s	d			
	25	120	5	2			
Encode pertinent payloads	m	ℓ_{cmb}					
	55	2					

Table 1: Parameters for InstantOMR.

6.2 Benchmark

We benchmark InstantOMR scheme using Google Compute Cloud "c3d-highcpu-360", equipped with 180 cores and 708 GB of memory.[12] We conducted comprehensive experiments evaluating the runtime of the detector under varying thread count and message count. For improved readability and analytical clarity, we have selected and presented a representative subset of the data in the following tables and figures. For Table 2 and Figs. 3 and 4, we use $\mathsf{D} = 2^{16}$ messages as [26], and for Fig. 2, we vary D, the total number of messages.

Table 2 presents the runtime data of InstantOMR (Primus-fhe), InstantOMR (TFHE-rs), PerfOMR and SophOMR, where InstantOMR (TFHE-rs) is the estimated runtime of InstantOMR if it were implemented using TFHE-rs , which will be discussed in more detail in § 6.4. It is easy to see that our implementation of InstantOMR has a much better latency ($\sim$277×), but worse in terms of the total runtime for a single core ($\sim$163×). This disadvantage of throughput decreases with sufficient number of cores, which we discuss in more detail in § 6.3. A prior work Homerun [25] focuses on two-server OMR, so we do not directly compare to it in this section. See the full version for a more detailed discussion.

Table 2 also shows the key size, clue size and digest size of InstantOMR, PerfOMR and SophOMR. Compared to prior

[11] For all the discussion above (including for prior works), we assume the recipient also sends $\bar{k}$ in advance. See the full version for details on why and what happens if not.

[12] InstantOMR requires only $\leq$3GB of memory even when processing 2^{16} messages even with 180 cores.

	Latency (ms)	Detector Time (ms/msg)	Recipient Time (ms)	Clue Key Size (Bytes)	Clue Size (Bytes)	Detection Key Size (MB)	Digest Size (KB)
PerfOMR [30]	88,742	7.34	12	2197	2191	171	567
SophOMR [26] (1 core)	76,065	1.68	14	2576	2570	114	263
SophOMR (180 cores)	52,846	1.38	14	2576	2570	114	263
InstantOMR (Primus-fhe, 180 cores)	274	1.69	201	720	714	113	825
InstantOMR (estimated TFHE-rs, 180 cores)	88	0.55	201	720	714	113	825
InstantOMR (Primus-fhe, 1 core)	274	274.51	201	720	714	113	825
InstantOMR (estimated TFHE-rs, 1 core)	88	88.39	201	720	714	113	825

Table 2: Performance of OMR Schemes ($\mathsf{D} = 2^{16}$, $k = 50$). Latency is measured for one message. Detecttor time is ms-per-msg (total-time/D) following PerfOMR [26], the state-of-the-art OMR construction. , Latency is defeind to be the detector runtime taken to process a single (new) message, which is equivalent to the runtime taken to run $\mathsf{D} = 1$ message. Detector time, following prior works [26, 30], is defined to be the total detector runtime divided by D messages, which is equivalently the *inverse* of throughput. Recipient time is the recipient runtime for the digest generated from all these D messages and k pertinent messages. We discuss multi-threading for prior works and our construction in § 6.3.

	Total (ms)	First-layer Boot (ms)	Second-layer Boot (ms)	Encode (ms)
InstantOMR (Primus-fhe, 1 core)	274	164	110	0.127
InstantOMR (TFHE-rs, 1 core)	88	69	19	0.127

Table 3: Runtime/msg Breakdown for InstantOMR

works, we show smaller clue key and clue sizes, comparable detection key size, and larger digest size. Note that in terms of sizes, clue size could be considered as a practical concern in some applications. For example, in Zcash Sapling protocol [23], a transaction itself is around 1.39 KB, and the clue sizes in prior works are even larger than the transaction itself, thus inefficient. Thus, InstantOMR has additional advantages for schemes that have small payload sizes.

Fig. 2 illustrates the detector time of InstantOMR, PerfOMR and SophOMR across varying message payload count and thread count.

Runtime breakdown. Lastly, we briefly discuss how our detector runtime breaks down. The runtime breakdown for detecting a single message in InstantOMR is presented in Table 3. The "Total" column is the sum runtime of last three columns. The "First-layer Boot" column represents the sum of 7 times the values in the "First bootstrapping" column of Table 4 plus the FHE.KS time. The "Second-layer Boot" column corresponds to the sum of the values in the "Second bootstrapping" column of Table 4 and the FHE.HomTrace time. It is evident that the most time-consuming operation is the two-layers bootstrapping (taking $> 99.85\%$ of the total runtime). The runtime of "Encode", which generates the digest, is very small compared to two-layers bootstrapping (only $< 0.15\%$ of the total time). Thus, any future improvement on functional bootstrapping (§ 3.1.1) could significantly improve our scheme as well.

6.3 Multithreading

Additionally, InstantOMR supports *optimal multi-threading*, further improving detection efficiency—an advantage unattainable in previous BFV-based schemes.

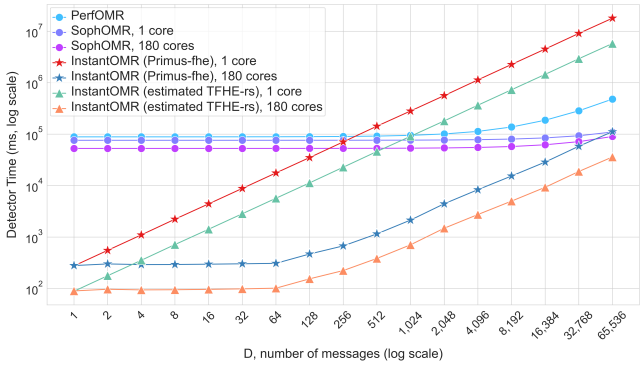

Figure 2: Benchmark of detector total runtime.

Parallelizability for prior works. Prior works do not benefit much from multi-threading for the following two reasons: (1) The OMR schemes in [26, 30] only naturally support multi-threads when there are $> N$ messages (for N being the ring dimension, either 32768 or 65536), which means that for smaller N, their runtime is close to that of running N messages even for multi-threading. Under such a multi-threading method, to fully utilize a machine with 180-cores as ours, more than ten million messages are needed to be processed at a time. (2) The BFV scheme [13, 22] itself does not benefit much from multi-threading. In particular, the only effective way known is to parallelize BFV operations over the RNS limb, which is essentially the multiplicative depth (or half of it for the parameters of [30]) [7]. However, the depth is only < 20 even for the full circuit of [26, 30]. Furthermore, when the homomorphic circuit proceeds, the depth becomes smaller. Specifically, for [26], about 1/2 of the runtime is over ciphertexts with multiplicative depth 1 or 2. Thus, even for 180 cores, the most optimistic improvement one can hope for is about 2-4x.

In fact, the implementation of SophOMR [26] does support multi-threading (since the library it builds on, OpenFHE [7] natively supports it). With the same machine, their runtime is only $< 1.5\times$ faster in terms of latency and about $1.2\times$ faster

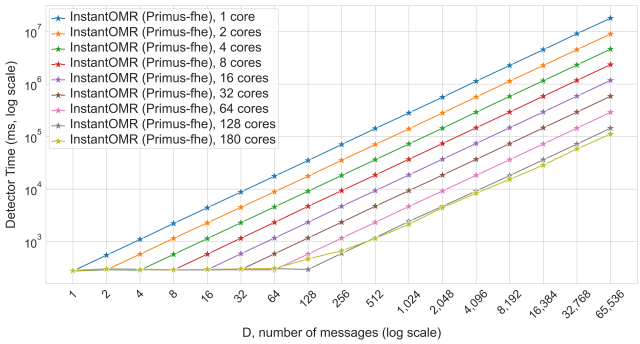

Figure 3: Benchmarked detector total time of InstantOMR, with Primus-fhe

	First bootstrapping (ms)	Second bootstrapping (ms)
Primus-fhe	22.83	106.07
TFHE-rs	9.82	15.18

Table 4: Bootstrapping Performance: Primus-fhe [36] vs TFHE-rs [43]

in terms of total runtime, which matches the intuition above.

Our throughput. With this in mind, Figs. 2 and 3 demonstrate that by increasing the number of threads, InstantOMR achieves significantly higher throughput. With 180 cores, our implementation has similar throughput as [26] with the same number of cores. If the server is strong enough (with D cores), our total runtime can be as low as retrieving a single messages, thus showing essentially optimal parallelizability. The minor anomaly of 180 cores in Fig. 3 is due to that we benchmark with D always being a power-of-two. Note that multi-threading is *only* used to improve throughput not latency of InstantOMR.

6.4 Estimation with TFHE-rs

InstantOMR **using TFHE-rs.** The TFHE-rs FHE library [43] is generally faster than the Primus-fhe library we use in our implementation, but TFHE-rs could not be directly used since it is not amenable to our non-black-box TFHE usage (see Rmk 6.2 for details). Nonetheless, we obtain estimates of performance for our scheme if it were implemented using a suitable extension of TFHE-rs, by the following methodology: To replace Primus-fhe with TFHE-rs, the only difference is

LWE	n_1	q_1	$\boldsymbol{s}_1$	p_1	σ_{ck}	
	630	2^{32}	Binary	2^3	3.05×10^{-5}	
RLWE	N_1	Q_1	z_1	B_1	d_1	σ_{bsk1}
	1024	2^{32}	Binary	2^7	3	2.98×10^{-8}
Keyswitching	n_1	q_1	$\boldsymbol{s}_1$	B_{ks}	d_{ks}	σ_{ksk}
	630	2^{32}	Binary	2^2	8	3.05×10^{-5}

Table 5: TFHE-rs parameters for the first bootstrapping.

LWE	n_2	q_2	$\boldsymbol{s}_2$	p_2	σ	
	863	2^{64}	Binary	2^4	2.15×10^{-6}	
RLWE	N_2	Q_2	z_2	B_2	d_2	σ_{bsk2}
	2048	2^{64}	Binary	2^{23}	1	2.85×10^{-15}
Keyswitching	n_2	q_2	$\boldsymbol{s}_2$	B_{ks}	d_{ks}	σ_{ksk}
	863	2^{64}	Binary	2^3	5	2.15×10^{-6}

Table 6: TFHE-rs parameters for the second bootstrapping.

that we need to use TFHE-rs for our two layers of bootstrapping (i.e., line 4 of Alg 4 and line 1 of Alg 5). Thus, we benchmark TFHE-rs with parameters (Tables 5 and 6) similar to the parameters (see Rmk 6.1 below) we use for our two layers of bootstrapping. Given the parameter $\ell = 7$ (i.e., the number of bootstrapping calls for the first layer), when using TFHE-rs, we obtain the speed difference for the two layers of bootstrapping is $\frac{22.83\times7+106.07}{9.82\times7+15.18} \approx 3.17$ (see also Table 4 for how this calculation is obtained). Consequently, using TFHE-rs, we estimate that our scheme can earn $> 3\times$ speed up (since these two layers take $> 99.85\%$ of total runtime as shown in Table 3).

Remark 6.1. Table 5 and Table 6 present the parameters used to estimate the bootstrapping of TFHE-rs scheme. These two parameter sets are used in the TFHE-rs library. The parameter N_i (where $i \in \{1,2\}$) corresponds to those in Table 1, while n_i (where $i \in \{1,2\}$) is larger than the respective parameter specified in Table 1. Additionally, the benchmark results for TFHE-rs in Table 4 comprises both bootstrapping and key-switching times, whereas those for Primus-fhe doesn't contain key-switching times.

Even under these two adverse conditions ((1) larger n_i and (2) TFHE-rs' benchmark result contains key-switching times), TFHE-rs still outperforms Primus-fhe by this factor. Thus, we believe that it is safe to conclude that when implemented using TFHE-rs, InstantOMR can *at least* be $3\times$ faster than the current implementation.

Remark 6.2. We use Primus-fhe [36] instead of TFHE-rs [43] since Primus-fhe is easier to modify for our non-black-box TFHE usage. TFHE-rs is very suitable for black-box use by application developers, and has better efficiency optimizations, but is not amenable to the requisite structural changes, such as having input vs. output ciphertexts with different parameters, including secret key dimension, ciphertext modulus, and plaintext modulus.

Performance. As shown in Table 2 and Fig. 2, with this estimation using TFHE-rs, InstantOMR has even smaller latency ($\sim$865$\times$ compared to single-core SophOMR and $\sim$600$\times$ compared to 180-core SophOMR). Our throughput is also improved, now $\sim$53$\times$ smaller than SophOMR with single-thread, but $\sim$2.5$\times$ larger than SophOMR when 180 cores are used. We also show in Fig. 4 on how InstantOMR with TFHE-rs estimation scales with the number of cores.

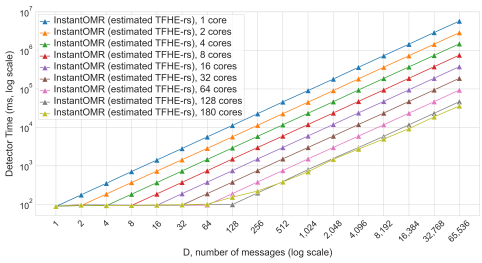

Figure 4: Estimated detector total time of InstantOMR with TFHE-rs

Remark 6.3. While not a fair comparison due to different environment settings, we briefly compare with [25] in terms of concrete efficiency for completeness (where the numbers are directly taken from the paper since they have a similar instance setup). Specifically, for 2^{15} messages, their runtime per message is ≈ 0.01ms per message online time, including $\sim$580MB of server-to-server communication (numbers taken from [19]). This is indeed orders of magnitude more efficient than our construction and also a lot faster than the existing state-of-the-art single-server OMR SophOMR. However, this is achieved under two non-colluding and semi-honest servers (where semi-honesty is required even for privacy, not only correctness). A very recent, concurrent and independent work [19] improves it to have security against malicious servers, but the runtime becomes similar to SophOMR (about 2-3$\times$ after), while still assuming two non-colluding servers.

6.5 Applications and Trade-offs

Latency and small D**.** InstantOMR demonstrates significantly lower latency, achieving orders of magnitude improvement over prior works [26, 30]. Specifically, for a single message, the latency of our scheme can be around 864 times smaller than using [26], the fastest existing single-server OMR scheme, and roughly 1008 times smaller than using [30], the existing single-server OMR with the smallest latency, if implemented using TFHE-rs.

Additionally, our scheme shines when the number of messages to-be-processed is small: This can be either that the recipient checks its inbox regularly, so a small amount of messages accumulates, or in the streaming setting (see below). With a single-thread, Fig. 2 shows that InstantOMR (implemented with Primus-fhe) outperforms PerfOMR and SophOMR for message counts below $\sim$256 in terms of *both* latency and runtime. When implemented with TFHE-rs, InstantOMR maintains superior performance for message counts below $\sim$900 (again, single thread; multi-threading discussed below).

Large D**.** For a large (e.g., $\mathsf{D} > 900 \cdot c$ for c cores) number of accumulated messages, directly using InstantOMR is less efficient. However, instead, SophOMR [26] and InstantOMR can be used in a hybrid way. Specifically, SophOMR can process these accumulated messages, an InstantOMR can process the new incoming messages. This guarantees both throughput efficiency and low-latency in real-time retrievals. Furthermore, since the two schemes both use the RLWE-based constructions, the digest provided by the two constructions can potentially be merged. Of course, this may require additional techniques such as ring-switching. We leave it for future work to explore such use in more detail.

Streaming setting. In the streaming setting (§ 5.5), if there are t' messages arrives the second that the recipient is online, the recipient's wait time for InstantOMR is $t' \cdot 88$ (for TFHE-rs). Note that even for Bitcoin-scale applications, $t' \leq 7$ (§ 5.5), and thus the waiting time is at most $\approx$ 600ms. Furthermore, for a t'-core detector, the waiting time is as low as a single message (i.e., 88ms). However, for [26, 30], the waiting time in the streaming setting is at least around 76 seconds, Thus, InstantOMR (using TFHE-rs) has a 123$\times$ advantage (or 864$\times$ advantage for a t'-core detector) compared to prior works. In general, even for a single-core detector, our scheme outperforms both prior works for $t' \lessapprox 900$ as discussed above.

Another example application is Zcash [39]: in Zcash, if a client remained offline on 11/07/2025, they would have 20,408 transactions to process when they reconnect.[13] In prior constructions, these transactions can be processed only after the client reconnects and issues a retrieval request (since 20,408 is smaller than the ring dimension), so the client would wait roughly 2 minutes before receiving any messages. Using InstantOMR, the client can obtain the updates of these messages essentially instantaneously (since the detector has already processed all the arrived messages). Furthermore, even if there are more than ring-dimension number of messages (e.g., on 11/13/2025, with 73,826 Zcash transactions), a similar issue remains: a batch of 65,536 transactions can be processed by SophOMR [26] in advance, However, to receive the remaining 8290, the client still needs to wait for roughly 2 minutes. Again, such wait time can be eliminated by using InstantOMR. As discussed above, one could potentially use SophOMR to process the large batch and use InstantOMR to process the rest.

Trade-off between the latency and throughput. In short, InstantOMR offers a trade-off throughput with latency compared to prior works [26]. As discussed above, InstantOMR provides $>$ 600 latency improvement (with estimated runtime with TFHE-rs) compared to SophOMR and $<$ 0.1 second of waiting time for recipients who need real-time updates. On the other hand, it also puts burden to the detector: it takes 88ms to process a message instead of only 1.68ms as in SophOMR ($\sim$53$\times$ slower). Additionally, with a super powerful server (e.g., 180 cores), InstantOMR is preferred in most applications, since the throughput can be improved to be larger than

[13] https://bitinfocharts.com/comparison/zcash-transactions.html.

SophOMR as well.

Acknoledgement

Zeyu Liu is supported by Yale CADMY. Yu Yu is supported by the National Natural Science Foundation of China (Grant Nos. 92270201 and 62125204). This paper was edited for grammar using Grammarly, ChatGPT, and Google Gemini.

Ethical Considerations

Ethics Summary.

Our work introduces InstantOMR, an oblivious message retrieval (OMR) scheme with optimal parallelizability and low latency. In general, we believe that OMR does not introduce additional risks, since it merely provides a more efficient mechanism for outsourcing message retrieval. In principle, the same retrieval task could always be performed locally by the recipient, without relying on a detector or third party. Thus, OMR does not provide stronger privacy guarantees than those already present in privacy-preserving messaging systems, but instead improves performance and usability.

It is important to note that, as with all privacy-preserving communication tools, there exists the possibility of misuse by adversaries who seek to hide their activities. However, this risk stems from the general availability of anonymous messaging, not from the existence of OMR specifically. The improvements provided by our scheme are unlikely to cause additional harm in nature: they make the system more practical for legitimate users but do not fundamentally change the adversarial threat landscape (who is more likely to have enough resources for messaging retrieval). For these reasons, we believe that studying and publishing more efficient OMR protocols raises no new ethical concerns.

Detailed Stakeholder-Based Ethics Analysis

Stakeholders. The primary stakeholders include: (1) recipients of private messages who may benefit from efficient retrieval mechanisms, (2) senders of private messages whose privacy is indirectly preserved, (3) service providers that operate private messaging systems, (4) the research community that may build upon our proposed construction or work on topics that could benefit from our techniques, and (5) society at large, which benefits from secure and efficient communication technologies.

Impacts. Our contribution is the introduction of InstantOMR, an Oblivious Message Retrieval (OMR) scheme with low latency and optimal parallelizability. The core impact is improved efficiency in private message retrieval systems. Importantly, OMR does not expand the scope of privacy guarantees: it simply provides an efficient means to perform a task that a recipient could otherwise execute locally. Thus, while the work may improve system scalability and performance, it does not alter the underlying security or privacy model.

Ethical Principles. We considered the Menlo Report principles:

- **Beneficence:** The research benefits recipients and messaging platforms by enabling faster and more resource-efficient communication, without introducing new risks.
- **Respect for Persons:** No human subjects are directly involved, and the scheme does not undermine user autonomy or informed consent.
- **Justice:** The efficiency gains are broadly applicable and not restricted to a privileged subset of users or providers. As demonstrated in the Open Science section below, we will open-source our code, so it will be publicly available to all.
- **Respect for Law and Public Interest:** The scheme does not enable violations of law; it merely improves existing cryptographic protocols. However, as noted, private messaging apps may be misused by adversaries, but as argued, such a risk is unlikely to be amplified by InstantOMR or OMR in general.

Harms. We identify minimal risks. One hypothetical concern is that privacy-preserving messaging systems, in general, could be misused by malicious parties to conceal unlawful activities. However, such risks are independent of the efficiency of the OMR component. OMR itself does not expand the adversarial capabilities—it only reduces computational and communication overhead. However, note that such a reduction could potentially make an attacker's malicious behaviors cheaper in such a system. Specifically, in the case of our work, it allows the attacker to use the anonymous messaging system with streaming updates on a lightweight device (and outsource the message retrieval process to a server). We would like to additionally note that an adversary can already use anonymous messaging systems with a more powerful machine, such that they could process message retrieval without outsourcing.

No tangible harms (financial, psychological, or physical) are introduced by our methodology or its publication, nor violation of human rights, to the best of our knowledge.

Mitigations. Since OMR does not alter the threat model of existing private messaging systems, no special mitigations seem to be required for OMR particularly. However, it is certainly worth posing additional regulations to such anonymous systems as it becomes more efficient and affordable as OMR advances.

Decision. We proceeded with this research because it offers clear technical benefits (efficiency and scalability for regular lightweight clients) with minimal new ethical risks. Publication is justified because the societal benefits of disseminating an efficient primitive outweigh the small risk (if any) that efficiency gains might indirectly facilitate malicious use of messaging systems.

Open Science

An anonymous repository is provided at [2]. The repository contains the full implementation of our system and enables reviewers to reproduce the evaluation results in § 6. Following the USENIX Security open-science requirements, the repository will remain accessible throughout the review period. Upon acceptance, we will release a non-anonymized public version and submit the artifact for evaluation of availability, functionality, and reproducibility.

References

[1] YCharts. https://ycharts.com/indicators/bitcoin_transactions_per_day. Accessed: 2025-08-05. At most 573,275 transactions in July 2025.

[2] Instantomr. GitHub, 2025. https://github.com/xiangxiecrypto/tfhe-omr.

[3] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology (2015).*, GitHub:https://github.com/malb/lattice-estimator.

[4] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication-computation trade-offs in PIR. In M. Bailey and R. Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 1811–1828. USENIX Association, Aug. 11–13, 2021.

[5] S. Angel, H. Chen, K. Laine, and S. T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[6] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of the 4th International Workshop on Information Hiding*, IHW '01, Berlin, Heidelberg, 2001.

[7] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca. OpenFHE: Open-source fully homomorphic encryption library. WAHC'2022, 2022.

[8] G. Beck, J. Len, I. Miers, and M. Green. Fuzzy message detection. In G. Vigna and E. Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 1507–1528, Virtual Event, Republic of Korea, Nov. 15–19, 2021. ACM Press.

[9] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[10] O. Biçer and C. Tschudin. Oblivious homomorphic encryption. Cryptology ePrint Archive, Report 2023/1699, 2023.

[11] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, 2017.

[12] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.

[13] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, Aug. 19–23, 2012. Springer Berlin Heidelberg, Germany.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3), July 2014.

[15] H. Chen, W. Dai, M. Kim, and Y. Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In K. Sako and N. O. Tippenhauer, editors, *ACNS 21: 19th International Conference on Applied Cryptography and Network Security, Part I*, volume 12726 of *Lecture Notes in Computer Science*, pages 460–479, Kamakura, Japan, June 21–24, 2021. Springer, Cham, Switzerland.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, Jan. 2020.

[17] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In M. Tibouchi and H. Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699, Singapore, Dec. 6–10, 2021. Springer, Cham, Switzerland.

[18] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science*, pages 41–50, Milwaukee, Wisconsin, Oct. 23–25, 1995. IEEE Computer Society Press.

[19] H. Chu, X. Wang, and Y. Jia. Private signaling secure against actively corrupted servers. Cryptology ePrint Archive, Paper 2025/1056, 2025.

[20] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[21] L. Ducas and D. Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640, Sofia, Bulgaria, Apr. 26–30, 2015. Springer Berlin Heidelberg, Germany.

[22] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[23] D.-E. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. https://zips.z.cash/protocol/sapling.pdf, 2024.

[24] S. Jakkamsetti, Z. Liu, and V. Madathil. Scalable private signaling. Cryptology ePrint Archive, Report 2023/572, 2023.

[25] Y. Jia, V. Madathil, and A. Kate. HomeRun: High-efficiency oblivious message retrieval, unrestricted. In B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, editors, *ACM CCS 2024: 31st Conference on Computer and Communications Security*, pages 2012–2026, Salt Lake City, UT, USA, Oct. 14–18, 2024. ACM Press.

[26] K. Lee and Y. Yeo. SophOMR: Improved oblivious message retrieval from SIMD-aware homomorphic compression. Cryptology ePrint Archive, Report 2024/1814, 2024.

[27] Z. Liu, K. Sotiraki, E. Tromer, and Y. Wang. Snake-eye resistant PKE from LWE for oblivious message retrieval and robust encryption. Eurocrypt'25, 2024.

[28] Z. Liu and E. Tromer. Oblivious message retrieval. In Y. Dodis and T. Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 753–783, Santa Barbara, CA, USA, Aug. 15–18, 2022. Springer, Cham, Switzerland.

[29] Z. Liu, E. Tromer, and Y. Wang. Group oblivious message retrieval. In *2024 IEEE Symposium on Security and Privacy*, pages 4367–4385, San Francisco, CA, USA, May 19–23, 2024. IEEE Computer Society Press.

[30] Z. Liu, E. Tromer, and Y. Wang. PerfOMR: Oblivious message retrieval with reduced communication and computation. In D. Balzarotti and W. Xu, editors, *USENIX Security 2024: 33rd USENIX Security Symposium*, Philadelphia, PA, USA, Aug. 14–16, 2024. USENIX Association.

[31] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer Berlin Heidelberg, Germany.

[32] V. Madathil, A. Scafuro, I. A. Seres, O. Shlomovits, and D. Varlakov. Private signaling. In K. R. B. Butler and K. Thomas, editors, *USENIX Security 2022: 31st USENIX Security Symposium*, pages 3309–3326, Boston, MA, USA, Aug. 10–12, 2022. USENIX Association.

[33] N. Mathewson and R. Dingledine. Practical traffic analysis: extending and resisting statistical disclosure. In *Proceedings of the 4th International Conference on Privacy Enhancing Technologies*, PET'04, page 17–34, Berlin, Heidelberg, 2004.

[34] D. Micciancio and V. Vaikuntanathan. SoK: Learning with errors, circular security, and fully homomorphic encryption. In Q. Tang and V. Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part IV*, volume 14604 of *Lecture Notes in Computer Science*, pages 291–321, Sydney, NSW, Australia, Apr. 15–17, 2024. Springer, Cham, Switzerland.

[35] S. Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098, 2015.

[36] primus labs. primus-fhe, 2024. `https://github.com/primus-labs/primus-fhe`.

[37] S. Pu, S. A. K. Thyagarajan, N. Döttling, and L. Hanzlik. Post quantum fuzzy stealth signatures and applications. In W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 371–385, Copenhagen, Denmark, Nov. 26–30, 2023. ACM Press.

[38] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), Sept. 2009.

[39] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, page 459–474, USA, 2014. IEEE Computer Society.

[40] J. Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. Cryptology ePrint Archive, Report 2016/479, 2016.

[41] R. Wang, J. Ha, X. Shen, X. Lu, C. Chen, K. Wang, and J. Lee. FHEW-like leveled homomorphic evaluation: Refined workflow and polished building blocks. Cryptology ePrint Archive, Paper 2024/1318, 2024.

[42] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, Oct. 2012.

[43] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. `https://github.com/zama-ai/tfhe-rs`.

Table 7: Parameter Table

Notation	Meaning
D	Message count on the board
k	Pertinent messages count
$\bar{k}$	Upper bound of pertinent messages count
t,p,p_1,p_2,p_3	Plaintext moduli
n,n_1,n_2	LWE ciphertext dimension
N,N_1,N_2	RLWE ciphertext dimension
q,q_1,q_2	LWE ciphertext moduli
Q,Q_1,Q_2	RLWE ciphertext moduli
σ,σ_{ck},σ_{bsk1},σ_{bsk2},σ_{ak}	Noise standard deviation
B_1,B_{ks},B_2,B_{ak}	Gadget basis
d_1,d_{ks},d_2,d_{ak}	Gadget length
ℓ	The count of ones in clue
d	Number of coefficients per accumulator
ℓ_{max}	Max repeat count of encoding indices
N_s	Buckets count of one segment
m	Matrix row count of encoding payloads
ℓ_{cmb}	Combinations count encode in one ciphertext
t	Number of messages accumulated in the streaming setting
t'	Number of messages arrived the recipient is online

A Additional Related Works

Oblivious message retrieval. There are two additional works in the OMR line of work [10] and [19, 25].

[10] provides an alternative way to achieve DoS-resistance (alternative to [27]). However, unfortunately, as demonstrated in [27], this scheme is only of theoretical interest. As noted in § 4, [27] is sufficient to allow InstantOMR to possess DoS-resistance with only moderate overhead.

Homerun [25] on the other hand provides an OMR construction in two-communicating-but-non-colluding servers. However, while Homerun provides an efficient OMR construction, it (1) relies on a stronger environment assumption than the focus of this paper, and (2) assumes both servers to be semi-honest even for privacy, thus requiring an even stronger assumption. It also introduces the concept of "deletion" for OMR, which is out of the scope of this work. Thus, we only compare it briefly in the full version for completeness. A very recent work [19] introduces constructions to add malicious efficiency and to improve its communication cost.

Fuzzy message detection. [37] introduces a post-quantum secure FMD. However, similarly, it focuses on a weaker decoy-based security guarantee.

Private signaling. As mentioned, PS [24,32] rely on Trusted Execution Environments (TEEs). TEE-based approaches impose significant environmental assumptions, as numerous studies demonstrate their vulnerability to side-channel attacks that compromise secrecy [40]. While the construction in [24] offers exceptional scalability—with runtime scaling poly-logarithmically relative to message volume—we exclude direct comparisons due to its reliance on this nonstandard setting. Another construction of PS in [32] relies on the exact same environment assumption as Homerun [25].

Private information retrieval. Private information retrieval (PIR) [18] is a cryptographic primitive that allows the recipient to query specific data from the remote databases while completely concealing the indices or identifiers of the retrieved information from the database server. PIR performs a query with an index or label that is hidden from the server, whereas OMR performs retrieval with the recipient's evaluation key, which does not reveal the information of the messages that would be retrieved. Unlike PIR [4, 5], where the recipient must know the indices of messages to retrieve, OMR operates under the assumption that users lack prior knowledge of message indices. The detector scans all messages across the board, subsequently returning every pertinent message.

Homomorphic encryption. BFV [13, 22] or BGV [14] is a leveled homomorphic encryption scheme that enables arithmetic circuit evaluations of bounded depth. [13] first proposed an efficient LWE-based construction and [22] ports it to Ring-LWE setting, significantly improving computational efficiency. A key strength of BFV lies in its SIMD (Single Instruction, Multiple Data) batching capability: A single ciphertext encodes a vector of N plaintext "slots" (typically $N = 2^{14}$ to 2^{16}), enabling vectorized homomorphic operations with throughput scaling as $O(N)$. However, this also limits its latency and parallelizability, which we discuss in more detail in our full version.