

SophOMR: Improved Oblivious Message Retrieval from SIMD-Aware Homomorphic Compression

Keewoo Lee*
Ethereum Foundation

Yongdong Yeo*
Seoul National University

Abstract

Privacy-preserving blockchains and private messaging services that ensure receiver-privacy face a significant UX challenge: each client must scan every payload posted on the public bulletin board to avoid missing messages intended for them. Oblivious Message Retrieval (OMR) addresses this issue by securely outsourcing this expensive scanning process to a service provider using Homomorphic Encryption (HE).

In this work, we propose a new OMR scheme that substantially improves upon the previous state-of-the-art, PerfOMR (USENIX Security'24). Our implementation demonstrates reductions of 3.4x in runtime, 2.2x in digest size, and 1.5x in key size, in a scenario with 65536 payloads (each 612 bytes), of which up to 50 are pertinent. At the core of these improvements is a new homomorphic compression mechanism, where ciphertexts of length proportional to the number of total payloads are compressed into a digest whose length is proportional to the upper bound on the number of pertinent payloads. Unlike previous approaches, our scheme fully exploits the native homomorphic SIMD structure of the underlying HE scheme, significantly enhancing efficiency. In the setting described above, our compression scheme achieves 7.5x speedup compared to PerfOMR.

1 Introduction

Consider a classic scenario where a sender wants to deliver a message to a receiver without revealing any information to eavesdroppers. While end-to-end encryption ensures *message-privacy*, it alone does not guarantee *metadata-privacy*. In particular, eavesdroppers may still gather details such as who is communicating, when, and how often, enabling traffic analysis [6,52] and compromising privacy.[1][2] As a former NSA general counsel said:[3] "Metadata absolutely tells you everything about somebody's life. If you have enough metadata you don't really need content." In this regard, mechanisms for metadata-privacy are critical in private messaging systems [4,24,61,63] and privacy-preserving cryptocurrencies [5,10,53].

In this work, we specifically focus on *receiver-privacy*, which aims to hide the receiver's identity. A plausible solution to receiver-privacy is the *public bulletin board* approach. In this framework, the sender encrypts the message using the receiver's public key and posts it on a public bulletin board (e.g., blockchain). Over time, the bulletin board accumulates messages from various senders to various receivers. To retrieve their messages, a receiver scans the entire bulletin board and attempts to decrypt each message one by one. Messages that decrypt correctly are those intended for the receiver, while unsuccessful decryptions are ignored. This framework ensures that eavesdroppers without access to the decryption key cannot link messages to their receivers.

In fact, many privacy-preserving blockchain projects that aim to provide receiver-privacy are essentially following this approach (e.g., Zcash [10,35], Aztec [5], Monero [53], ERC-5564 [62] for Stealth Address). However, this approach poses a significant UX challenge: clients must individually scan every payload posted on a public bulletin board to avoid missing messages intended for them. In the context of privacy-preserving blockchains, this means that even light clients are required to scan all transactions.

One approach to addressing this issue is to *securely outsource* the expensive scan to a powerful server. *Homomorphic Encryption* (HE) [27,57], a cryptosystem that allows computation on encrypted data, is a natural cryptographic primitive for such a secure outsourcing scenario [16,17]. With HE, we can delegate the scan to an untrusted server without compromising the privacy of receivers. Liu and Tromer [43] formalized this HE-based secure scan outsourcing solution as a primitive called *Oblivious Message Retrieval (OMR)*. They also explored various design choices and optimization

*Equal contribution.

[1] https://www.propublica.org/article/how-facebook-undermines-privacy-protections-for-its-2-billion-whatsapp-users

[2] https://www.forbes.com/sites/thomasbrewster/2022/02/23/meet-the-secretive-surveillance-wizards-helping-the-fbi-and-ice-wiretap-facebook-and-google-users/

[3] https://www.wired.com/2015/03/data-and-goliath-nsa-metadata-spying-your-secrets/

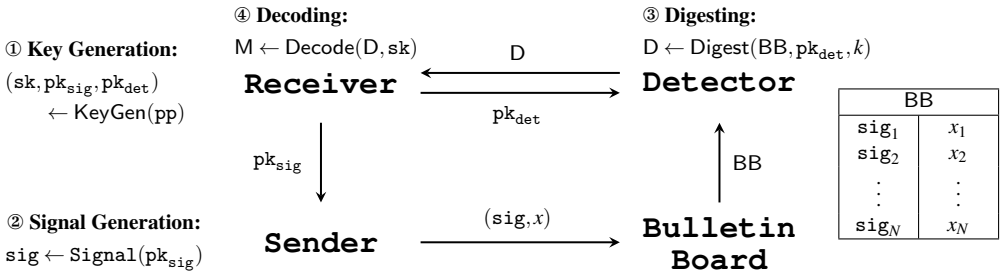


Figure 1: Oblivious Message Retrieval

techniques and provided a proof-of-concept implementation of OMR. However, this initial scheme required over 20 hours to process 2^{19} payloads (155 ms/payload), making it impractical for real-world use. A follow-up work, PerfOMR [46] by Liu-Tromer-Wang, advanced the practicality of OMR through further optimizations, achieving significantly improved efficiency. Specifically, PerfOMR reported 15× faster runtime compared to the original OMR construction [43].

1.1 Our Contribution

In this work, we propose a new OMR scheme, SophOMR, that substantially improves upon PerfOMR [46]. Specifically, it takes less than 3 minutes to process 2^{16} payloads in a single thread (2.5 ms/payload).

- While SophOMR largely follows the high-level framework of prior works [43, 46], our key conceptual contribution is in viewing OMR through the lens of *homomorphic compression* and in identifying compression as the primary bottleneck in prior works. At the heart of our improvement is our new homomorphic compression mechanism based on [21]. The main advantage of our approach lies in its compatibility with the homomorphic SIMD (Single Instruction, Multiple Data) structure of the underlying HE scheme. This SIMD-awareness avoids complexities found in prior works [43, 46], resulting in a more efficient and arguably simpler scheme. Notably, SophOMR requires only one level of multiplicative depth and $O(\sqrt{N})$ homomorphic rotations to compress N payloads, in contrast to PerfOMR, which requires two levels of depth and $O(N)$ rotations. In our primary setting, this yields a 7.5× speedup in homomorphic compression. See also Sec. 1.3 and Tab.1.

- We also suggest several other optimization techniques to further improve the performance of the scheme: ring-switching (Sec. 4.2), hybrid key-switching (Sec. 4.3), and BSGS-style homomorphic matrix multiplication (Sec. 4.4).

- We implement SophOMR in a C++ library and compare its concrete performance to that of PerfOMR. Our implementation demonstrates reductions of 3.4× in runtime, 2.2× in digest size, and 1.5× in key size, in a scenario with 65536 payloads (each 612 bytes), of which up to 50 are pertinent. We highlight that the comparison was conducted in a manner that favors PerfOMR, whose parameters achieve 94-bit security and 18-bit completeness, whereas our parameters offer 128-bit security and 30-bit completeness. (See Sec. 5.)

- We also consider OMD (Oblivious Message Detection), where clients retrieve only pertinent indices rather than the payloads (Sec. 4.6). OMD can be combined with PIR (Private Information Retrieval) schemes [23] to achieve OMR-like functionality. Although this approach adds an extra communication round, it provides a generic way to achieve OMR-like functionality while leveraging the specific advantages of the chosen PIR scheme. In the same setting, our OMD variant achieves an overall 4.6× speedup and a specific 479× speedup in homomorphic compression compared to PerfOMR (Sec. 5).

1.2 System Model

More precisely, OMR operates as follows (Fig. 1). For more details, refer to Sec. 2.5 and Sec. 3.

① Each user runs KeyGen, publishes their signal key $\mathtt{pk}_{\mathtt{sig}}$, and send the detection key $\mathtt{pk}_{\mathtt{det}}$ to Detector.

② Sender, who wants to deliver a payload[4] x to Receiver, posts the pair $(\mathtt{sig}, x)$ on the bulletin board BB, tagging the payload with a signal $\mathtt{sig}$ generated using the Receiver's signal key $\mathtt{pk}_{\mathtt{sig}}$. The signal allows only the intended receiver to detect the payload.

③ Upon Receiver's requests, Detector uses the Receiver's detection key $\mathtt{pk}_{\mathtt{det}}$ to summarize the bulletin board BB into a digest D and sends it to Receiver.

④ Finally, Receiver decodes the digest D to recover its pertinent payloads.

[4]Note that we treat the payload x simply as a bitstring. In practice, these payloads are typically end-to-end encrypted, possibly with accompanying authentication. Since OMR focuses on receiver privacy, we assume message privacy and integrity are handled by the payload format itself and therefore abstract them away as a bitstring.

At a high level, Detector (③) performs *homomorphic detection* of the signals, given homomorphically encrypted Receiver's secret as a part of the detection key $\mathtt{pk}_{\mathtt{det}}$. This yields an encrypted sparse binary vector, where the ones indicate the indices pertinent to Receiver. The encrypted binary vector is then used to mask the payloads on Bulletin Board, preserving only the pertinent payloads while zeroizing the rest. However, notice that the result is as large as the entire Bulletin Board; the Detector cannot simply discard irrelevant payloads since the detection result is encrypted. Therefore, homomorphic detection alone does not yield a *compact* digest.

This is where *homomorphic compression* comes into play. It is a technique that obliviously compresses a lengthy, homomorphically encrypted sparse vector into a digest whose size depends only on *an upper bound for the number of non-zero components* in the input ciphertexts. Applied to the output of homomorphic detection, it condenses the sparse vector of pertinent payloads into a compact digest. However, note that this requires the client to know an a priori upper bound on the number of pertinent payloads.

Remark 1 (A Priori Upper Bound)**.** *The upper bound for the number of pertinent payloads can be predicted from application scenarios and past statistics. If the number of relevant items exceeds this bound, it can be easily detected (e.g., through a checksum). In such cases, the query can simply be rerun with a larger bound. We leave the bound as a client-side configuration, which can be easily updated over time. While activity volume can be inferred through the bound, this leakage is unavoidable if the client's communication cost is to remain independent of the Bulletin Board size.*

Threat Model. The security of OMR (Sec. 2.5) considers a passive adversary attempting to link specific messages to their receivers. The adversary has access to the public bulletin board, public keys, and any communication among the parties in the system. It may also collude with any party except the sender and receiver of the target messages.

1.3 Technical Overview

SIMD-Aware Homomorphic Compression. Our main technical contribution is identifying that the bottleneck of prior works [43, 46] is at homomorphic compression and devising a new mechanism based on the SIMD-aware homomorphic compression scheme of [21]. Prior approaches rely on a *hashing-based* compression scheme (Sec. 3.4) in which each index is *hashed* into *buckets*, making the scheme incompatible with the homomorphic SIMD structure. As a result, before the actual compression, each slot in the detection result must be *unpacked* into separate ciphertexts, using costly subroutines like SlotToCoeff [30, 47] and *oblivious expansion* [3]. The latter step involves $O(N)$ homomorphic rotations, where N is the size of the bulletin board, creating a significant bottleneck in the digesting process.

Table 1: Cost Comparison for Digest Compression

	PerfOMR	SophOMR
#(PMul)	$\left(\frac{kt}{n} + 1 + \frac{\nu t}{n \log p}\right) N$	$\left(\frac{kt}{n} + \frac{k}{n}\right) N$
#(Rot)	$\left(4 + \frac{2}{\sqrt{n}} - \frac{4}{n}\right) N$	$\approx \sqrt{\frac{4k(t+1)}{n}} \sqrt{N}$
Depth	2	1

- N: # of payloads
- n: BFV ring dimension
- t: # of BFV slots per payload
- k: # of pertinent payloads
- p: BFV plaintext modulus
- $2^{-\nu}$: false negative probability

In contrast, our SIMD-aware homomorphic compression mechanism based on [21] eliminates the need for a complex and expensive unpacking process, enabling a seamless transition between the detection and compression phase (Sec. 4.1). Furthermore, our compression scheme ultimately reduces to a single homomorphic matrix-vector multiplication, a well-studied operation in the HE literature [18, 32, 33, 38, 48]. As a result, our scheme is arguably simpler to understand and implement, while also being significantly more efficient (see Tab. 1): the number of required homomorphic rotation is reduced from $O(N)$ to $O(\sqrt{N})$, and the number of required plaintext-ciphertext multiplications is also substantially reduced under realistic parameter settings ($kt < n$), compared to the previous hashing-based approach of PerfOMR. Furthermore, SophOMR achieves this compression using only a single multiplicative depth, whereas PerfOMR requires two—an additional depth is needed for the *unpacking* step—which directly impacts the overall efficiency of OMR. Another advantage of our compression scheme is that, unlike the previous hashing-based approach, it is *deterministic* and does not introduce any additional failure probability, which substantially simplifies the analysis for parameter setting.

A limitation of the SIMD-aware homomorphic compression of [21] is that it only considers small payloads that fit within a single message *slot* of the underlying HE (e.g., 16-bit). However, in many applications, including OMR, payloads are significantly larger and require multiple slots. To address this, we introduce *precomputation* and *stacking* techniques to accelerate compression of multi-slot payloads, achieving a quadratic reduction in the required number of homomorphic rotations, relative to the number of slots, compared to naively compressing each slot individually.

Optimizations. We also suggest several other optimization techniques to further improve the performance of our scheme.

- *Ring-Switching* (Sec. 4.2): We adopt *ring-switching* [14, 28, 29], which reduces the HE parameter after completing the desired homomorphic computations, to achieve further compression of the digest.

- *Hybrid Key-Switching Optimization* (Sec. 4.3): We optimize the parameter for *hybrid key-switching* [31, 34, 39] in the HE scheme to reduce the key size.

- *BSGS Optimization for Affine Transform* (Sec. 4.4): We apply *BSGS (Baby-Step-Giant-Step)* optimization to speed up homomorphic matrix-vector multiplications.

1.4 Related Work

Oblivious Message Retrieval. Besides the two works [43, 46] previously mentioned, OMR has been extended further by several studies. Group OMR [45] extends OMR to group messaging scenarios, offering enhanced efficiency than naive use of OMR, where the detector's workload scales linearly with the group size. DoS-resistant OMR [42] considers an extended security notion to safeguard OMR schemes against spamming attacks by malicious senders.

Receiver Privacy. Several non-HE-based solutions have been proposed that offer receiver privacy along with OMR/OMD-like functionality. Fuzzy Message Detection (FMD) [9] is a *decoy-based* solution that employs a detection algorithm specifically designed to misidentify impertinent messages at a predetermined false positive rate, effectively hiding the exact set of pertinent messages. However, the decoy-based security notion of FMD is indeed relatively complex and weak, as analyzed in [59]. In contrast to FMD, some works achieve full privacy [36, 37, 51], but they rely on stronger trust assumptions than OMR. HomeRun [37] and Scalable Private Signaling [36] are roughly two-orders-of-magnitude faster than SophOMR, assuming non-colluding servers and trusted hardware, respectively.

SIMD-Aware Homomorphic Compression. Several homomorphic compression methods have been proposed [11, 21, 22, 26, 43]. However, many of these methods [22, 26, 43] are not particularly efficient when implemented, as they are not SIMD-friendly due to their *hashing-based* approach. (See Sec. 3.4.2.) Although Cheon-Lee-Park-Yeo [21] is the first to investigate SIMD-friendliness, some earlier works feature structures that can be implemented in a SIMD-friendly manner. However, these approaches have their own limitations: The FFT-based method by Fleischhacker-Larsen-Simkin [26] has a costly decompression that requires solving small instances of discrete logarithms, and the method by Bienstock-Patel-Seo-Yeo [11] assumes that the entity performing the decompression already knows the indices of interest, which is not applicable in the context of OMR. Cheon et al. [21] proposed a SIMD-aware homomorphic compression scheme by reinterpreting and extending PS-COIE of Choi et. al. [22], which only considered compressing a binary vector and disregarded its SIMD-friendly structure. This new scheme achieves both an asymptotically optimal compression rate and asymptotically good decompression complexity.

2 Preliminaries

2.1 Notations

We use $[n]$ to denote the set of integers $\{0, \cdots, n-1\}$, and log refers to the base-2 logarithm. Throughout the paper, we follow the column vector notation. For a vector $\boldsymbol{v}$, we use $\boldsymbol{v}[i]$ to denote its i-th element, and for a matrix $\boldsymbol{M}$, we use $\boldsymbol{M}[i][j]$ to denote its (i,j)-th element. The Hadamard product (a.k.a. element-wise product) is denoted by $\odot$, and $\langle \cdot, \cdot \rangle$ denotes the inner product. For $i < n$, we use $\mathtt{Rot}^i$ to denote the left rotation of a vector by i positions, i.e., $\mathtt{Rot}^i(v_0, \cdots, v_{n-1}) = (v_i, \cdots, v_{n-1}, v_0, \cdots, v_{i-1})$. The vector $\mathbf{1}$ refers to a vector of ones $(1, \cdots, 1)$. We use λ for the security parameter and $\mathsf{negl}(\lambda)$ to denote a negligible function with respect to λ. The notation $\mathtt{ct}(m)$ denotes a ciphertext encrypting a message m, and $\mathtt{ct}(\boldsymbol{m})$ denotes possibly multiple ciphertexts encrypting the entries of a vector $\boldsymbol{m}$.

2.2 Homomorphic Encryption

A homomorphic encryption (HE) scheme allows us to compute an encryption of $f(m)$ when only given a circuit f and a ciphertext encrypting m without decryption. Let $\mathcal{M}$ be the message space and $\mathcal{C}$ be the ciphertext space. An HE scheme is a tuple of probabilistic polynomial time algorithms $(\mathtt{KeyGen}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Eval})$ described below:

- $\mathtt{KeyGen}(1^\lambda)$: The key generation algorithm takes a security parameter λ and returns a secret key $\mathtt{sk}$ and public key $\mathtt{pk}$.

- $\mathtt{Enc}_{\mathtt{pk}}(m)$: The encryption algorithm takes the public key $\mathtt{pk}$ and a message $m \in \mathcal{M}$ as inputs. Then, it returns a ciphertext $\mathtt{ct}(m) \in \mathcal{C}$ encrypting m.

- $\mathtt{Dec}_{\mathtt{sk}}(\mathtt{ct}(m))$: The decryption algorithm takes the secret key $\mathtt{sk}$ and a ciphertext $\mathtt{ct}(m) \in \mathcal{C}$ encrypting a message $m \in \mathcal{M}$. Then, it returns the message m.

- $\mathtt{Eval}_{\mathtt{pk}}(f, \mathtt{ct}(\boldsymbol{m}))$: The evaluation algorithm takes the public key $\mathtt{pk}$, a circuit f, and ciphertext $\mathtt{ct}(\boldsymbol{m}) \in \mathcal{C}^*$ as inputs. Then, it returns ciphertext $\mathtt{ct}(f(\boldsymbol{m})) \in \mathcal{C}^*$ that encrypts $f(\boldsymbol{m}) \in \mathcal{M}^*$ as the message.

Definition 1 (Security of HE). *Let* $(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathtt{KeyGen}(1^\lambda)$.

- ***Correctness:*** *For any circuit f (in some class $\mathcal{F}$) and input $\boldsymbol{m} \in \mathcal{M}^*$, the following holds with overwhelming probability for* $\mathtt{ct}(\boldsymbol{m}) \leftarrow \mathtt{Enc}_{\mathtt{pk}}(\boldsymbol{m})$*:*

$$\mathtt{Dec}_{\mathtt{sk}}\left(\mathtt{Eval}_{\mathtt{pk}}(f, \mathtt{ct}(\boldsymbol{m}))\right) = f(\boldsymbol{m})$$

- ***Semantic Security:*** *For any probabilistic polynomial time adversary $\mathcal{A}$ and $m_1, m_2 \in \mathcal{M}$, the following holds for $\mathtt{ct}_i \leftarrow \mathtt{Enc}_{\mathtt{pk}}(m_i)$:*

$$|\Pr[\mathcal{A}(\mathtt{pk}, \mathtt{ct}_1) = 1] - \Pr[\mathcal{A}(\mathtt{pk}, \mathtt{ct}_2) = 1]| \leq \mathsf{negl}(\lambda)$$

2.2.1 BFV Scheme

As in the previous works [43, 46], we employ the BFV scheme [13, 25] in our SophOMR construction. The BFV scheme supports homomorphic SIMD arithmetic in $\mathbb{Z}_p$ for a prime p, which is critical to our SophOMR construction, especially in our homomorphic compression scheme.

More precisely, we consider the BFV scheme with the ring dimension n that is a power of 2, along with the plaintext modulus p satisfying $p = 1 \pmod{2n}$. In this case, the plaintext space is given by $\mathcal{R}_p := \mathbb{Z}_p[X]/(X^n+1) \cong \mathbb{Z}_p^{2\times(n/2)}$, while the ciphertext space is $\mathcal{R}_Q^2$, where $\mathcal{R}_Q := \mathbb{Z}_Q[X]/(X^n+1)$ for a ciphertext modulus Q. For a plaintext $\boldsymbol{m} \in \mathbb{Z}_p^{2\times(n/2)}$, we often refer to each plaintext entry as a *slot*.

The BFV scheme supports (1) SIMD arithmetic in $\mathbb{Z}_p$ and (2) homomorphic rotations (horizontal and vertical). That is, we can homomorphically add and multiply a ciphertext encrypting $\boldsymbol{v} \in \mathbb{Z}_p^{2\times(n/2)}$ with a ciphertext (or plaintext) of $\boldsymbol{w} \in \mathbb{Z}_p^{2\times(n/2)}$ to obtain a ciphertext of $\boldsymbol{v}+\boldsymbol{w}$ and $\boldsymbol{v} \odot \boldsymbol{w}$, respectively. For homomorphic rotations, to be precise, let $\mathtt{Rot}^r(\cdot)$ denote the rotation operation which maps $[v_1, \cdots, v_{n/2}] \mapsto [v_{1+r}, \cdots, v_{n/2}, v_1, \cdots, v_r]$. Then, the BFV scheme supports homomorphic rotations computing $\mathtt{Rot}_{\mathrm{row}}$ and $\mathtt{Rot}_{\mathrm{col}}$, defined as follows for $\boldsymbol{m} = [\boldsymbol{m}_1 | \boldsymbol{m}_2]^T \in \mathbb{Z}_p^{2\times(n/2)}$, where $\boldsymbol{m}_1, \boldsymbol{m}_2 \in \mathbb{Z}_p^{n/2}$.

$$\mathtt{Rot}_{\mathrm{row}}^r\left(\begin{bmatrix}\boldsymbol{m}_1^T \\ \boldsymbol{m}_2^T\end{bmatrix}\right) = \begin{bmatrix}\mathtt{Rot}^r(\boldsymbol{m}_1^T) \\ \mathtt{Rot}^r(\boldsymbol{m}_2^T)\end{bmatrix}$$
$$\mathtt{Rot}_{\mathrm{col}}^1\left(\begin{bmatrix}\boldsymbol{m}_1^T \\ \boldsymbol{m}_2^T\end{bmatrix}\right) = \begin{bmatrix}\boldsymbol{m}_2^T \\ \boldsymbol{m}_1^T\end{bmatrix}$$

In this work, we will often abuse notation and identify a BFV plaintext $[\boldsymbol{m}_1|\boldsymbol{m}_2]^T \in \mathbb{Z}_p^{2\times(n/2)}$ with the vector $[\boldsymbol{m}_1^T|\boldsymbol{m}_2^T]^T \in \mathbb{Z}_p^n$.

2.2.2 Sparse Packing

When working with a vector $\boldsymbol{v}$ of length k that is smaller than the HE ring dimension n, we often employ *Sparse Packing* (a.k.a. Repetitive Slot Packing), where multiple copies of $\boldsymbol{v}$ are packed into a plaintext. More precisely, when $n = 0 \pmod{k}$, we pack $\boldsymbol{v}$ (n/k) times, resulting in $[\boldsymbol{v}^T|\cdots|\boldsymbol{v}^T]^T$. The rationale for using sparse packing is that it preserves the rotation structure. For example, $\mathtt{Rot}_{\mathrm{row}}^k([\boldsymbol{v}^T|\cdots|\boldsymbol{v}^T]^T) = [\boldsymbol{v}^T|\cdots|\boldsymbol{v}^T]^T$, which would not hold if we did not use sparse packing, such as when padding the remaining slots naively with zeros. Throughout this paper, whenever $n = 0 \pmod{\dim \boldsymbol{v}}$, $\mathtt{ct}(\boldsymbol{v})$ should be interpreted as a sparsely packed ciphertext.

2.3 Homomorphic Matrix Multiplication

At the heart of our SophOMR scheme is homomorphic matrix-vector multiplication (MatMul) [18,32,33,38,48]. The goal of MatMul is to multiply a plaintext matrix $\boldsymbol{M}$ with an encryption of vector $\boldsymbol{v}$ and obtain an encryption of $\boldsymbol{Mv}$. Here, we review an efficient MatMul algorithm that we use in SophOMR.

Diagonal Packing. The idea of *diagonal packing*, first introduced by Halevi and Shoup [32], is fundamental to MatMul algorithms. Given a matrix $\boldsymbol{M} \in \mathbb{Z}_p^{m\times k}$ and a vector $\boldsymbol{v} \in \mathbb{Z}_p^k$, let $\mathtt{diag}_i(\boldsymbol{M})$ denote the i-th diagonal packing of $\boldsymbol{M}$ (w.r.t. dimension n), which is defined as follows (for $0 \leq j < n$).

$$\mathtt{diag}_i(\boldsymbol{M})[j] = \boldsymbol{M}[j \bmod m][(i+j) \bmod k]$$

The rationale behind diagonal packing is that it enables a concise representation of a matrix-vector multiplication with native HE operations, specifically Hadamard products and rotations, as follows (for $n = m = k$).

$$\boldsymbol{Mv} = \sum_{i=0}^{k-1} \mathtt{diag}_i(\boldsymbol{M}) \odot \mathtt{Rot}^i(\boldsymbol{v}) \tag{1}$$

BSGS-Style MatMul. Halevi and Shoup [33] introduced the baby-step-giant-step (BSGS) style optimization to reduce the number of homomorphic rotations required in a MatMul algorithm. The core idea is captured by the following equalities, where $k = \tilde{g} \cdot \tilde{b}$ and $\boldsymbol{m}_{g\tilde{b}+b}$ denotes $\mathtt{Rot}^{-g\tilde{b}}\left(\mathtt{diag}_{g\tilde{b}+b}(\boldsymbol{M})\right)$. Notice that, by reusing the $\mathtt{Rot}^b(\boldsymbol{v})$ terms, we can reduce the number of rotations to $\tilde{g}+\tilde{b}$, rather than $k = \tilde{g} \cdot \tilde{b}$ as in Eq. 1.

$$\begin{aligned}\boldsymbol{Mv} &= \sum_{i=0}^{k-1} \mathtt{diag}_i(\boldsymbol{M}) \odot \mathtt{Rot}^i(\boldsymbol{v}) \\ &= \sum_{g=0}^{\tilde{g}-1}\sum_{b=0}^{\tilde{b}-1} \mathtt{diag}_{g\tilde{b}+b}(\boldsymbol{M}) \odot \mathtt{Rot}^{g\tilde{b}+b}(\boldsymbol{v}) \\ &= \sum_{g=0}^{\tilde{g}-1} \mathtt{Rot}^{g\tilde{b}}\left(\sum_{b=0}^{\tilde{b}-1} \boldsymbol{m}_{g\tilde{b}+b} \odot \mathtt{Rot}^b(\boldsymbol{v})\right)\end{aligned}$$

Among several follow-ups on [33], we adopt the approach of PEGASUS [48], which exhibits the best performance in the case of non-square matrices. The original MatMul algorithm of PEGASUS is designed for the CKKS scheme [20] and its rotation structure. In this work, we use its straightforward adaptation to our BFV setting [13, 25]. Additionally, we extend the approach to handle cases where the matrix dimension exceeds the underlying HE ring dimension, requiring the encrypted vectors to span multiple ciphertexts.

To be precise, let n be the ring dimension of the underlying BFV scheme. Let us consider the MatMul of $k \times N$ matrix $\boldsymbol{M}$ with a length-N vector $\boldsymbol{v}$. Here, we focus on the *wide* case,

Table 2: Cost Analysis for Alg. 1

#(Ptxt-Mult)	#(Rotation)	#(Rot. Key)
$s \cdot k$	$s \cdot \tilde{b} + \tilde{g} + \log(n/k)$	$\leq 2 + \log(n/k)$

where $N = s \cdot n$ for some positive integer s and $k < n$. For simplicity, we assume k is a power of two. Alg. 1 describes the BFV-adapted PEGASUS MatMul algorithm for the specified setting. We use the Horner-style [33] variant, which optimizes the number of required rotation keys. The costs for Alg. 1 are summarized in Tab. 2. Note that the number of rotations is minimized when $\tilde{g} \approx \sqrt{s \cdot k}$. For more detailed discussions, refer to PEGASUS [48] and references therein, such as [33].

Algorithm 1 MatMul (Wide)

Input: $\boldsymbol{M} \in \mathbb{Z}_p^{k \times N}$, $(\mathtt{ct}(\boldsymbol{v}_i))_{i=0}^{s-1}$ ▷ $k < n, N = s \cdot n, \boldsymbol{v}_i \in \mathbb{Z}_p^n$, k: power-of-two

Step 0: Parameter Setting

$[\boldsymbol{M}_0 | \cdots | \boldsymbol{M}_{s-1}] \leftarrow \boldsymbol{M}$ and $k = \tilde{g} \cdot \tilde{b}$

Step 1: Plaintext Packing

for $(i, j) \in [s] \times [k]$ **do**

$\boldsymbol{m}_{i,j} \leftarrow \mathtt{Rot}_{\mathrm{row}}^{-g\tilde{b}}\left(\mathtt{diag}_j(\boldsymbol{M}_i)\right)$ for $g = \lfloor j/\tilde{b} \rfloor$

Step 2: Baby-Step-Giant-Step (BSGS)

for $i = 0, \cdots, s-1$ **do** ▷ Baby-Step

$\mathtt{ct}_{i,0} \leftarrow \mathtt{ct}(\boldsymbol{v}_i)$

for $b = 1, \cdots, \tilde{b}-1$ **do**

$\mathtt{ct}_{i,b} \leftarrow \mathtt{Rot}_{\mathrm{row}}^{1}(\mathtt{ct}_{i,b-1})$

for $g = \tilde{g}-1, \cdots, 0$ **do** ▷ Giant-Step

$\mathtt{ct}_{\mathrm{sum}} \leftarrow \sum_{i=0}^{s-1} \sum_{b=0}^{\tilde{b}-1} (\boldsymbol{m}_{i,g\tilde{b}+b} \odot \mathtt{ct}_{i,b})$

if $g = \tilde{g}-1$ **then**

$\mathtt{ct}_{\mathrm{out}} \leftarrow \mathtt{ct}_{\mathrm{sum}}$

else

$\mathtt{ct}_{\mathrm{out}} \leftarrow \mathtt{Rot}_{\mathrm{row}}^{\tilde{b}}(\mathtt{ct}_{\mathrm{out}}) + \mathtt{ct}_{\mathrm{sum}}$

Step 3: Block Summation

for $j = 0, \cdots, \log(n/k) - 2$ **do**

$\mathtt{ct}_{\mathrm{out}} \leftarrow \mathtt{ct}_{\mathrm{out}} + \mathtt{Rot}_{\mathrm{row}}^{k \cdot 2^j}(\mathtt{ct}_{\mathrm{out}})$

$\mathtt{ct}_{\mathrm{out}} \leftarrow \mathtt{ct}_{\mathrm{out}} + \mathtt{Rot}_{\mathrm{col}}^{1}(\mathtt{ct}_{\mathrm{out}})$

return $\mathtt{ct}_{\mathrm{out}}$

2.4 Homomorphic Compression

The goal of homomorphic compression [11, 21, 22, 26, 43] is to homomorphically compress an encryption of a *sparse* vector, which has only a few non-zero components, into an encryption of a shorter vector, which can later be efficiently decompressed after decryption. In this section, we provide a brief overview of the homomorphic compression scheme from [21], which offers high efficiency due to its compatibility with homomorphic SIMD operations. For a more in-depth discussion, please refer to [21].

2.4.1 Index Compression

We first consider the task of compressing a sparse *binary* vector (a.k.a. index compression) as a stepping stone for compressing an arbitrary sparse vector (a.k.a. payload compression). The index compression of [21] leverages so-called *Newton's identity*, which provides conversions between *power sum polynomials* and *elementary symmetric polynomials*. Although the idea of using Newton's identity for index compression has been previously explored in the literature [22], the novelty of [21] lies in (1) recognizing that the power sum encoding can be interpreted as a matrix-vector multiplication and (2) implementing it with efficient SIMD-aware homomorphic algorithms for MatMul (e.g., Alg. 1).

More precisely, for $p > N$, consider a sparse binary vector $\boldsymbol{v} \in \mathbb{Z}_p^N$ with at most k number of non-zero entries and define a Vandermonde-like $k \times N$ matrix $\boldsymbol{C}$ defined as follows. The homomorphic compression scheme can then be concisely described as in Alg. 2, with decompression outlined in Alg. 3.

$$\boldsymbol{C} := \begin{bmatrix} 1 & 2 & \cdots & N \\ 1^2 & 2^2 & \cdots & N^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1^k & 2^k & \cdots & N^k \end{bmatrix} \pmod{p} \tag{2}$$

Algorithm 2 CompIdx

Input: $\mathtt{ct}(\boldsymbol{v})$ ▷ ciphertexts of $\boldsymbol{v} \in \mathbb{Z}_p^N$

$\mathtt{ct}(\boldsymbol{w}) \leftarrow \mathsf{MatMul}(\boldsymbol{C}, \mathtt{ct}(\boldsymbol{v}))$ ▷ see Sec. 2.3 and Alg. 1

return $\mathtt{ct}(\boldsymbol{w})$

Algorithm 3 DecompIdx

Input: $\boldsymbol{w} = (w_1, \cdots, w_k) \in \mathbb{Z}_p^k$ ▷ decryption of $\mathtt{ct}(\boldsymbol{w})$

Step 1: Build a polynomial from $\boldsymbol{w}$ using Newton's identity.

$a_0 \leftarrow 1$

for $j = 1, \cdots, k$ **do**

$a_j \leftarrow j^{-1} \cdot \sum_{i=1}^{j} (-1)^{i-1} a_{j-i} \cdot w_i$

$f_{\boldsymbol{v}}^k(X) \leftarrow \sum_{i=0}^{k} (-1)^i a_i X^{k-i}$

Step 2: Reconstruct non-zero indices from the polynomial.

while X divides $f_{\boldsymbol{v}}^k(X)$ **do**

$f_{\boldsymbol{v}}^k(X) \leftarrow f_{\boldsymbol{v}}^k(X)/X$

$I_{\boldsymbol{v}} \leftarrow$ Find zeros of $f_{\boldsymbol{v}}^k(X)$ ▷ by Cantor-Zassenhaus

return $I_{\boldsymbol{v}}$

2.4.2 Payload Compression

With index compression at hand, we can now compress an arbitrary sparse vector. Consider a sparse vector $\boldsymbol{d} \in \mathbb{Z}_p^N$ with at most k non-zero entries. To compress $\mathtt{ct}(\boldsymbol{d})$, we first obtain an encryption of the binary vector $\boldsymbol{v}$, whose indices of non-zero entries match those in $\boldsymbol{d}$. For example, Fermat's little theorem can be employed for this purpose (as outlined in Alg. 4). However, in certain scenarios (e.g., private database query [21]), it may not be necessary to derive $\mathtt{ct}(\boldsymbol{v})$ from $\mathtt{ct}(\boldsymbol{d})$, as $\mathtt{ct}(\boldsymbol{v})$ may be already available from other source. In fact, $\mathtt{ct}(\boldsymbol{d})$ is often computed *from* $\mathtt{ct}(\boldsymbol{v})$. Looking ahead, this is the case also for OMR.

Then, we can compress $\mathtt{ct}(\boldsymbol{v})$ via Alg. 2 and use $\boldsymbol{v}$ as a *hint* in the decompression phase. Note that any k distinct columns of the matrix $\boldsymbol{C}$ are linearly independent. Therefore, we can also compress $\mathtt{ct}(\boldsymbol{d})$ by performing a MatMul with $\boldsymbol{C}$ and decompress it by solving a linear equation for a submatrix of $\boldsymbol{C}$. The homomorphic compression scheme is described in Alg. 4 and the decompression in Alg. 5.

Algorithm 4 Comp

Input: $\mathtt{ct}(\boldsymbol{d})$ ▷ ciphertexts of $\boldsymbol{d} \in \mathbb{Z}_p^N$

$\mathtt{ct}(\boldsymbol{v}) \leftarrow \mathtt{Eval}(\mathsf{Power}_{p-1}, \mathtt{ct}(\boldsymbol{d}))$[5] ▷ Fermat's little theorem

$\mathtt{ct}(\boldsymbol{w}) \leftarrow \mathsf{CompIdx}(\mathtt{ct}(\boldsymbol{v}))$ ▷ see Alg. 2

$\mathtt{ct}(\boldsymbol{e}) \leftarrow \mathtt{Eval}(\boldsymbol{C}, \mathtt{ct}(\boldsymbol{d}))$

return $\mathtt{ct}(\boldsymbol{e}), \mathtt{ct}(\boldsymbol{w})$

Algorithm 5 Decomp

Input: $\boldsymbol{e}, \boldsymbol{w} \in \mathbb{Z}_p^k$ ▷ decryption of $\mathtt{ct}(\boldsymbol{e}), \mathtt{ct}(\boldsymbol{w})$

$I_{\boldsymbol{v}} \leftarrow \mathsf{DecompIdx}(\boldsymbol{w})$ ▷ see Alg. 3

$\hat{\boldsymbol{C}} \leftarrow [\boldsymbol{C}_{i_1} | \cdots | \boldsymbol{C}_{i_\ell}]$ ▷ $k \times \ell$ matrix whose j-th column is i_j-th column of $\boldsymbol{C}$, where $I_{\boldsymbol{v}} = \{i_1, \cdots, i_\ell\}$

$\hat{\boldsymbol{d}} = (\hat{d}_1, \cdots, \hat{d}_\ell) \leftarrow$ Solve $\hat{\boldsymbol{C}}\boldsymbol{x} = \boldsymbol{e}$ for $\boldsymbol{x}$

return $\hat{\boldsymbol{d}}$

2.5 OMR and OMD

In this section, we review the formal definitions of Oblivious Message Retrieval (OMR) and Oblivious Message Detection (OMD) [43]. See also Sec. 1.2 and Fig. 1.

2.5.1 Oblivious Message Retrieval (OMR)

Definition 2 (Oblivious Message Retrieval)**.** *An* Oblivious Message Retrieval (OMR) *scheme (w.r.t. payload space $\mathcal{P}$) is a tuple of probabilistic algorithms* $(\mathsf{ParamGen}, \mathsf{KeyGen}, \mathsf{Signal}, \mathsf{Digest}, \mathsf{Decode})$ *described below:*

- $\mathtt{pp} \leftarrow \mathsf{ParamGen}(1^\lambda)$*: The parameter generation algorithm takes a security parameter λ and returns a public parameter* $\mathtt{pp}$*. The following algorithms implicitly take* $\mathtt{pp}$ *as an input.*
- $(\mathtt{sk}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}}) \leftarrow \mathsf{KeyGen}(\mathtt{pp})$*: The key generation algorithm outputs a secret key* $\mathtt{sk}$*, a signal key* $\mathtt{pk}_{\mathtt{sig}}$*, and a detection key* $\mathtt{pk}_{\mathtt{det}}$*.*
- $\mathtt{sig} \leftarrow \mathsf{Signal}(\mathtt{pk}_{\mathtt{sig}})$*: The signaling algorithm takes a signal key* $\mathtt{pk}_{\mathtt{sig}}$ *and outputs a signal* $\mathtt{sig}$*.*

A bulletin board $\mathsf{BB} = ((x_1, \mathtt{sig}_1), \cdots, (x_N, \mathtt{sig}_N))$ *is formed by several parties uploading a payload $x_i \in \mathcal{P}$ tagged with a signal $\mathtt{sig}_i$ with respect to the intended receiver.*

- $\mathsf{D} \leftarrow \mathsf{Digest}(\mathsf{BB}, \mathtt{pk}_{\mathtt{det}}, k)$*: The digesting algorithm takes a bulletin board* $\mathsf{BB} = ((x_1, \mathtt{sig}_1), \cdots, (x_N, \mathtt{sig}_N))$ *for some size N, a detection key* $\mathtt{pk}_{\mathtt{det}}$*, and an upper bound k on the number of pertinent payloads addressed to the receiver. Then, it returns a digest* D*.*
- $\mathsf{M} \leftarrow \mathsf{Decode}(\mathsf{D}, \mathtt{sk})$*: The decoding algorithm takes a digest* D *and a secret key* $\mathtt{sk}$*. Then, it returns a list of decoded payloads $\mathsf{M} \in \mathcal{P}^*$.*

Definition 3 (Security of OMR)**.** *Let* $\mathtt{pp} \leftarrow \mathsf{ParamGen}(1^\lambda)$.

- ***Correctness:*** *Let* $(\mathtt{sk}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}}) \leftarrow \mathsf{KeyGen}(\mathtt{pp})$*. Let* BB *be a bulletin board of length N and* $\tilde{\mathsf{M}}$ *be the list of pertinent payloads in* BB *addressed to* $\mathtt{pk}_{\mathtt{sig}}$*. For $k \geq |\tilde{\mathsf{M}}|$, the following holds.*

$$\Pr\left[\mathsf{M} \neq \tilde{\mathsf{M}} \;\middle|\; \begin{array}{l} \mathsf{D} \leftarrow \mathsf{Digest}(\mathsf{BB}, \mathtt{pk}_{\mathtt{det}}, k) \\ \mathsf{M} \leftarrow \mathsf{Decode}(\mathsf{D}, \mathtt{sk}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

- ***Receiver-Privacy:*** *Signals generated from different signal keys should be computationally indistinguishable to anyone who does not have access to the corresponding secret keys. To be precise, let* $(\mathtt{sk}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}})$ *and* $(\mathtt{sk}', \mathtt{pk}_{\mathtt{sig}}', \mathtt{pk}_{\mathtt{det}}')$ *be two set of keys generated from* $\mathsf{KeyGen}(\mathtt{pp})$*. Let* $\mathtt{sig} \leftarrow \mathsf{Signal}(\mathtt{pk}_{\mathtt{sig}})$ *and* $\mathtt{sig}' \leftarrow \mathsf{Signal}(\mathtt{pk}_{\mathtt{sig}}')$*. For any probabilistic polynomial time adversary $\mathcal{A}$, the following holds for* $\mathtt{aux} = (\mathtt{pp}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}}, \mathtt{pk}_{\mathtt{sig}}', \mathtt{pk}_{\mathtt{det}}')$*.*

$$\left|\Pr[\mathcal{A}(\mathtt{aux}, \mathtt{sig}) = 1] - \Pr[\mathcal{A}(\mathtt{aux}, \mathtt{sig}') = 1]\right| \leq \mathsf{negl}(\lambda)$$

2.5.2 Oblivious Message Detection (OMD)

Oblivious Message Detection (OMD) is a variant of OMR where the receiver only retrieves pertinent *indices*, rather than *payloads*. Note that by combining an OMD scheme with a private information retrieval (PIR) scheme [23] one can obtain an OMR-like functionality at the cost of an additional round of communication: first, retrieve pertinent indices using OMD, and then use PIR to obtain the corresponding payloads. Below are formal definitions for OMD.

[5] Power_ℓ denotes an arithmetic circuit over $\mathbb{Z}_p$ such that $\mathsf{Power}_\ell(\boldsymbol{d}) = (d_1^\ell, \cdots, d_N^\ell)$, where $\boldsymbol{d} = (d_1, \cdots, d_N) \in \mathbb{Z}_p^N$.

Definition 4 (Oblivious Message Detection)**.** *An* Oblivious Message Detection (OMD) *scheme is a tuple of probabilistic algorithms* $(\mathsf{ParamGen}, \mathsf{KeyGen}, \mathsf{Signal}, \mathsf{Digest}, \mathsf{Decode})$. *Descriptions for* ParamGen*,* KeyGen*,* Signal*, and* Digest *are identical to those of OMR (Def. 2).*

- $\mathsf{M} \leftarrow \mathsf{Decode}(\mathsf{D}, \mathtt{sk})$*: The decoding algorithm takes a digest* D *and a secret key* $\mathtt{sk}$*. Then, it returns a list of pertinent indices* $\mathsf{M} \in \mathbb{N}^*$.

Definition 5 (Security of OMD)**.** *Let* $\mathtt{pp} \leftarrow \mathsf{ParamGen}(1^\lambda)$. *The definition of* receiver-privacy *is identical to that of OMR (Def. 3).*

- ***Correctness:*** *Let* $(\mathtt{sk}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}}) \leftarrow \mathsf{KeyGen}(\mathtt{pp})$. *Let* BB *be a bulletin board of length* N *and* $\tilde{\mathsf{M}}$ *be the list of pertinent indices addressed to* $\mathtt{pk}_{\mathtt{sig}}$ *in* BB. *For* $k \leq |\tilde{\mathsf{M}}|$, *the following holds.*

$$\Pr\left[\mathsf{M} \neq \tilde{\mathsf{M}} \;\middle|\; \begin{array}{l} \mathsf{D} \leftarrow \mathsf{Digest}(\mathsf{BB}, \mathtt{pk}_{\mathtt{det}}, k) \\ \mathsf{M} \leftarrow \mathsf{Decode}(\mathsf{D}, \mathtt{sk}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

3 Review: PerfOMR [46]

In this section, we provide a concise overview of the PerfOMR scheme [46]. First, in Sec. 3.1, we recall a building block for OMR, which we call *private signaling*. Then, we review each component of PerfOMR: ParamGen, KeyGen, Signal (Sec. 3.2), and Digest (Sec. 3.3 and 3.4). Specifically, the Digest algorithm is presented in a modular manner, split into a detection phase (Sec. 3.3) and a compression phase (Sec. 3.4).

3.1 RLWE-based Private Signaling Scheme

A Private Signaling (PS) scheme is a core building block of OMR, enabling a sender to generate a *signal* that only the intended receiver can detect. We begin by presenting a formal definition of private signaling (Sec. 3.1.1), followed by a construction based on the Ring Learning with Errors (RLWE) assumption [49, 56] (Sec. 3.1.2).

3.1.1 Private Signaling

Definition 6 (Private Signaling[6])**.** *A* Private Signaling (PS) *scheme is a tuple of probabilistic polynomial time algorithms* $(\mathtt{ParamGen}, \mathtt{KeyGen}, \mathtt{Signal}, \mathtt{Detect})$ *described below:*

- $\mathtt{pp} \leftarrow \mathtt{ParamGen}(1^\lambda)$*: The parameter generation algorithm takes a security parameter* λ *and returns a public parameter* $\mathtt{pp}$*. The following algorithms implicitly take* $\mathtt{pp}$ *as an input.*
- $(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathtt{KeyGen}(\mathtt{pp})$*: The key generation algorithm returns a secret key* $\mathtt{sk}$ *and a public key* $\mathtt{pk}$.
- $\mathtt{sig} \leftarrow \mathtt{Signal}(\mathtt{pk})$*: The signaling algorithm takes a public key* $\mathtt{pk}$ *and returns a signal* $\mathtt{sig}$.
- $0/1 \leftarrow \mathtt{Detect}_{\mathtt{sk}}(\mathtt{sig})$*: The detection algorithm takes a secret key* $\mathtt{sk}$ *and a signal* $\mathtt{sig}$*. Then, it returns* 1 *if* $\mathtt{sig}$ *was generated from a public key corresponding to* $\mathtt{sk}$*; otherwise, it returns* 0.

Definition 7 (Security of Private Signaling)**.** *Let* $\mathtt{pp} \leftarrow \mathtt{ParamGen}(1^\lambda)$.

- ***Completeness:*** *The following holds.*

$$\Pr\left[\begin{array}{l} (\mathtt{sk}, \mathtt{pk}) \leftarrow \mathtt{KeyGen}(\mathtt{pp}); \\ \mathtt{sig} \leftarrow \mathtt{Signal}(\mathtt{pk}) : \\ \mathtt{Detect}_{\mathtt{sk}}(\mathtt{sig}) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

- ***Soundness:*** *The following holds.*

$$\Pr\left[\begin{array}{l} (\mathtt{sk}, \mathtt{pk}) \leftarrow \mathtt{KeyGen}(\mathtt{pp}); \\ (\mathtt{sk}', \mathtt{pk}') \leftarrow \mathtt{KeyGen}(\mathtt{pp}); \\ \mathtt{sig} \leftarrow \mathtt{Signal}(\mathtt{pk}) : \\ \mathtt{Detect}_{\mathtt{sk}'}(\mathtt{sig}) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda)$$

- ***Receiver-Privacy:*** *Signals generated from different public keys should be computationally indistinguishable to anyone who does not have access to the corresponding secret keys. To be precise, let* $(\mathtt{sk}, \mathtt{pk})$ *and* $(\mathtt{sk}', \mathtt{pk}')$ *be two set of keys generated from* $\mathtt{KeyGen}(\mathtt{pp})$. *Let* $\mathtt{sig} \leftarrow \mathtt{Signal}(\mathtt{pk})$ *and* $\mathtt{sig}' \leftarrow \mathtt{Signal}(\mathtt{pk}')$. *For any probabilistic polynomial time adversary* $\mathcal{A}$*, the following holds for* $\mathtt{aux} = (\mathtt{pp}, \mathtt{pk}, \mathtt{pk}')$.

$$\left|\Pr[\mathcal{A}(\mathtt{aux}, \mathtt{sig}) = 1] - \Pr[\mathcal{A}(\mathtt{aux}, \mathtt{sig}') = 1]\right| \leq \mathsf{negl}(\lambda)$$

3.1.2 Construction

We now present an overview of the private signaling scheme from PerfOMR [46] whose security is based on the hardness of the *Ring Learning with Errors (RLWE)* problem [49, 56]. For a more detailed discussion of the scheme, refer to [46].

Remark 2 (On Choice of LWE-based Scheme)**.** *Although we have a generic conversion from any PKE scheme with reasonable properties, private signaling schemes in this line of works [43, 46], are based on (Ring) LWE assumption. The primary reason is that the decryption circuit of LWE-based PKE is better suited to the computational model of HE than other cryptographic assumptions (e.g., group-based assumptions). Specifically, a large part of the decryption process is just a simple linear algebra over a relatively small modulus.*

We first introduce a notation to ease the presentation. We denote and define the i-th *negacyclic* vector of $\boldsymbol{a} = (a_0, \cdots, a_{n'-1}) \in \mathbb{Z}_{q'}^{n'}$ as follows, for $0 \leq i < n'$.

$$\mathtt{Neg}^i(\boldsymbol{a}) = (a_i, \cdots, a_0, -a_{n'-1}, \cdots, -a_{i+1})$$

[6] Our definition differs from that of [51] and aims to capture the framework they refer to as *naive approach* based on *key-private* PKE.

Let $a(X) = \sum_{i=0}^{n-1} a_i X^i$ for $\boldsymbol{a}$ and similarly for $b(X)$ and $\boldsymbol{b} = (b_0, \cdots, b_{n'-1})$. Notice that $\langle \mathtt{Neg}^i(\boldsymbol{a}), \boldsymbol{b} \rangle$ equals to the i-th coefficient of $a(X) \cdot b(X)$ over $\mathcal{R}'_{q'} = \mathbb{Z}_{q'}[X]/(X^{n'}+1)$. With this in mind, the following construction can be seen, at a high level, as an RLWE-based PKE scheme specifically designed for encrypting zeroes. It can be viewed as a *ring-optimized* variant of the PVW scheme [55], or as a *truncated* version of the LPR scheme [50].

Construction 1. *The private signaling scheme* $\mathtt{PS} = (\mathtt{PS.ParamGen}, \mathtt{PS.KeyGen}, \mathtt{PS.Signal}, \mathtt{PS.Detect})$ *is defined as follows.*

- $\mathtt{pp} \leftarrow \mathtt{PS.ParamGen}(1^\lambda)$*: The parameter generation algorithm returns* $\mathtt{pp} = (n', q', \ell, r, \chi_s, \chi_e)$*, consisting of polynomial ring dimension* n'*, modulus* q'*, repetition parameter* ℓ*, range parameter* r*, secret key distribution* χ_s *over* $\mathbb{Z}_{q'}^{n'}$*, and error distribution* χ_e *over* $\mathbb{Z}_{q'}$*.*
- $(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathtt{PS.KeyGen}(\mathtt{pp})$*: The key generation algorithm first samples a secret key* $\mathtt{sk} \leftarrow \chi_s$*. Then, it samples* $\boldsymbol{\alpha} \leftarrow \mathbb{Z}_{q'}^{n'}$ *and compute* $\boldsymbol{\beta} = \left(\left\langle \mathtt{Neg}^i(\boldsymbol{\alpha}), \mathtt{sk} \right\rangle\right)_{i=0}^{n'-1} + \boldsymbol{e} \in \mathbb{Z}_{q'}^{n'}$ *with noise* $\boldsymbol{e} \leftarrow \chi_e^{n'}$*. Finally, the algorithm returns* $(\mathtt{sk}, \mathtt{pk})$*, where* $\mathtt{pk} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathbb{Z}_{q'}^{n'} \times \mathbb{Z}_{q'}^{n'}$
- $\mathtt{sig} \leftarrow \mathtt{PS.Signal}(\mathtt{pk})$*: The signaling algorithm takes a public key* $\mathtt{pk} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathbb{Z}_{q'}^{n'} \times \mathbb{Z}_{q'}^{n'}$ *and samples noises* $\boldsymbol{e}_0 \leftarrow \chi_s$*,* $\boldsymbol{e}_1 \leftarrow \chi_e^{n'}$*, and* $\boldsymbol{e}_2 \leftarrow \chi_e^{\ell}$*. Then, it computes* $\boldsymbol{a} = \left(\langle \mathtt{Neg}^i(\boldsymbol{\alpha}), \boldsymbol{e}_0 \rangle\right)_{i=0}^{n'-1} + \boldsymbol{e}_1$ *and* $\boldsymbol{b} = \left(\langle \mathtt{Neg}^i(\boldsymbol{\beta}), \boldsymbol{e}_0 \rangle\right)_{i=0}^{\ell-1} + \boldsymbol{e}_2$*. Finally, the algorithm returns* $\mathtt{sig} = (\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{Z}_{q'}^{n'} \times \mathbb{Z}_{q'}^{\ell}$*.*
- $0/1 \leftarrow \mathtt{PS.Detect}_{\mathtt{sk}}(\mathtt{sig})$*: The detection algorithm takes* $\mathtt{sig} = (\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{Z}_{q'}^{n'} \times \mathbb{Z}_{q'}^{\ell}$ *and computes* $\boldsymbol{d} = \boldsymbol{b} - \left(\langle \mathtt{Neg}^i(\boldsymbol{a}), \mathtt{sk} \rangle\right)_{i=0}^{\ell-1} \in \mathbb{Z}_{q'}^{\ell}$*. Then, it computes* c_i*'s by checking* $\boldsymbol{d} = (d_1, \cdots, d_\ell)$ *with the range parameter* r*, as follows.*

$$c_i = \begin{cases} 0, & \text{if } d_i \in [-r, r] \\ 1, & \text{otherwise} \end{cases}$$

Finally, the algorithm returns $\prod_{i=1}^{\ell}(1-c_i) \in \{0,1\}$*.*

Remark 3 (Security of Construction 1)**.** *We sketch the security of Construction 1 (Def. 7). For details, refer to [46].*

- ***Completeness:*** *The range parameter* r *can be set to guarantee completeness, following standard analysis in LWE-based encryptions to guarantee correctness.*
- ***Soundness:*** *The repetition parameter* ℓ *can be set to guarantee soundness as follows: By the hardness of RLWE over* $\mathcal{R}'_{q'}$*,* $\mathtt{sig} \leftarrow \mathtt{PS.Signal}(\mathtt{pk})$ *is indistinguishable from a uniform random sample, in the absence of access to the secret key* $\mathtt{sk}$*. Therefore, the following holds when* $\mathtt{sk}'$ *is sampled independently of* $\mathtt{sk}$*.*

$$\Pr[\mathtt{PS.Detect}_{\mathtt{sk}'}(\mathtt{sig}) = 1] \leq \left(\frac{2r+1}{q'}\right)^{\ell} + \mathsf{negl}(\lambda)$$

- ***Receiver-Privacy:*** *The receiver-privacy directly follows from the hardness of RLWE over* $\mathcal{R}'_{q'}$*.*

3.2 Setup

Given homomorphic encryption scheme BFV (Sec. 2.2.1) and private signaling scheme PS (Sec. 3.1.2), (ParamGen, KeyGen, Signal) of PerfOMR can be described as follows.

- $\mathtt{pp} \leftarrow \mathsf{ParamGen}(1^\lambda)$: Run $\mathtt{pp}_{\mathtt{PS}} \leftarrow \mathtt{PS.ParamGen}(1^\lambda)$ and return $\mathtt{pp} = (\mathtt{pp}_{\mathtt{PS}}, 1^\lambda)$.
- $(\mathtt{sk}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}}) \leftarrow \mathsf{KeyGen}(\mathtt{pp})$: First, run $(\mathtt{sk}_{\mathtt{PS}}, \mathtt{pk}_{\mathtt{PS}}) \leftarrow \mathtt{PS.KeyGen}(\mathtt{pp}_{\mathtt{PS}})$ and $(\mathtt{sk}_{\mathtt{BFV}}, \mathtt{pk}_{\mathtt{BFV}}) \leftarrow \mathtt{BFV.KeyGen}(1^\lambda)$. Then, compute $\mathtt{ct}_{\mathtt{sk}} \leftarrow \mathtt{BFV.Enc}(\mathtt{sk}_{\mathtt{PS}})$. Return $\mathtt{sk} = (\mathtt{sk}_{\mathtt{PS}}, \mathtt{sk}_{\mathtt{BFV}})$, $\mathtt{pk}_{\mathtt{sig}} = \mathtt{pk}_{\mathtt{PS}}$, and $\mathtt{pk}_{\mathtt{det}} = (\mathtt{pk}_{\mathtt{BFV}}, \mathtt{ct}_{\mathtt{sk}})$.
- $\mathtt{sig} \leftarrow \mathsf{Signal}(\mathtt{pk}_{\mathtt{sig}})$: Return $\mathtt{sig} \leftarrow \mathtt{PS.Signal}(\mathtt{pk}_{\mathtt{PS}})$.

3.3 Detection

For clarity, we describe the Digest algorithm of PerfOMR in a modular manner, divided into (1) a *detection* phase and (2) a *compression* phase. In this section, we begin by illustrating Detect (Alg. 6) for the detection phase, followed by an explanation of the compression phase in Sec. 3.4.

The goal of Detect is to homomorphically compute the pertinent indices from the bulletin board BB using the detection key $\mathtt{pk}_{\mathtt{det}}$. The output of Detect is ciphertexts encrypting a sparse *binary* vector, where the ones indicate the pertinent indices, referred to as the *Pertinency Vector*, PV.[7] At a high level, Detect can be viewed as a *batched homomorphic detection* of signals, consisting of two main steps: (i) *Affine Transform* and (ii) *Range Check*.

Affine Transform. The goal of Affine Transform (Step 1 of Alg. 6) is to compute *linear* parts of multiple $\mathtt{PS.Detect}_{\mathtt{sk}_{\mathtt{PS}}}(\mathtt{sig}_i)$ in parallel. More precisely, we want to compute $\boldsymbol{d} = \boldsymbol{b}_i - \left(\langle \mathtt{Neg}^j(\boldsymbol{a}_i), \mathtt{sk} \rangle\right)_{j=0}^{\ell-1} \in \mathbb{Z}_{q'}^{\ell}$ where $\mathtt{sig}_i = (\boldsymbol{a}_i, \boldsymbol{b}_i) \in \mathbb{Z}_{q'}^{n'} \times \mathbb{Z}_{q'}^{\ell}$ for $1 \leq i \leq N$. Notice that we can stack up the inner products $\langle \mathtt{Neg}^j(\boldsymbol{a}_i), \mathtt{sk} \rangle$ into a matrix-vector multiplication. In this way, we can leverage homomorphic MatMul algorithms (Sec. 2.3) with respect to $\mathtt{ct}_{\mathtt{sk}}$. We will explore this in greater detail in Sec. 4.4.

Range Check. The goal of Range Check (Step 2 of Alg. 6) is to homomorphically compute the remaining *non-linear* parts of multiple $\mathtt{PS.Detect}_{\mathtt{sk}_{\mathtt{PS}}}(\mathtt{sig}_i)$. More precisely, we want to compute $\prod_{i=1}^{\ell}(1-c_i) \in \{0,1\}$ where r is the range parameter and

$$c_i = \begin{cases} 0, & \text{if } d_i \in [-r, r] \\ 1, & \text{otherwise,} \end{cases}$$

[7] Prior works [43, 46] refer to this procedure as *ClueToPackedPV*.

Algorithm 6 Detect

Input: $\mathtt{pk}_{\mathtt{det}}$, $\mathsf{BB} = (x_i, \mathtt{sig}_i)_{i=1}^{N}$ ▷ $N = s \cdot n$

Step 0: Signal Preprocessing ▷ $\mathtt{sig}_i = (\boldsymbol{a}_i, \boldsymbol{b}_i)$

for $j = 0, \cdots, \ell - 1$ **do**

$\boldsymbol{A}_j \leftarrow [\mathtt{Neg}^j(\boldsymbol{a}_1) | \cdots | \mathtt{Neg}^j(\boldsymbol{a}_N)]^T$ ▷ $\boldsymbol{A}_j \in \mathbb{Z}_{q'}^{N \times n'}$

$\boldsymbol{\beta}_j \leftarrow (b_{1,j}, \cdots, b_{N,j})$ ▷ $\boldsymbol{b}_i = (b_{i,j})_{j=0}^{\ell-1}$

Step 1: Affine Transform

for $j = 0, \cdots, \ell - 1$ **do**

$(\mathtt{ct}_{i,j})_{i=0}^{s-1} \leftarrow \boldsymbol{\beta}_j - \mathsf{MatMul}(\boldsymbol{A}_j, \mathtt{ct}_{\mathtt{sk}})$ ▷ Sec. 4.4

Step 2: Range Check ▷ $g(X) = X \cdot \prod_{i=1}^{r}(X^2 - i^2)$, $h(X) = X^{p-1}$

for $i = 0, \cdots, s - 1$ **do**

for $j = 0, \cdots, \ell - 1$ **do**

$\mathtt{ct}_{i,j} \leftarrow \mathtt{BFV.Eval}_{\mathtt{pk}_{\mathtt{BFV}}}(g(X), \mathtt{ct}_{i,j})$

$\mathtt{ct}_{i,j} \leftarrow \mathtt{BFV.Eval}_{\mathtt{pk}_{\mathtt{BFV}}}(h(X), \mathtt{ct}_{i,j})$

$\mathtt{ct}_i \leftarrow \prod_{j=0}^{\ell-1}(\mathbf{1} - \mathtt{ct}_{i,j})$

return $(\mathtt{ct}_i)_{i=0}^{s-1}$

for $\boldsymbol{d} = (d_1, \cdots d_\ell)$ which is the output from Affine Transform. The computation of c_i's can be done homomorphically by evaluating the following function on d_i's.

$$f(X) = \begin{cases} 0, & \text{if } -r \le X \le r \\ 1, & \text{otherwise} \end{cases} \tag{3}$$

In PerfOMR [46], $f(X)$ is homomorphically evaluated by first computing $g(X) = \prod_{i=0}^{r}(X^2 - i^2)$ and then $h(X) = X^{q'-1}$, rather than directly evaluating the polynomial interpolation of $f(X)$ as done in [43]. Notice that $g(X)$ maps the input to 0 if and only if it lies within the range $[-r, r]$, and $h(X)$ maps non-zero inputs to 1 and zero to 0 by Fermat's little theorem. By decomposing the evaluation of f into g and h, PerfOMR significantly reduced the number of homomorphic multiplications required, while only slightly increasing the circuit depth, resulting in an overall improvement in computational cost.

3.4 Compression

In the compression phase, pertinent payloads are compressed into a digest using the encrypted pertinency vector PV. Roughly speaking, PerfOMR compresses both PV and pertinent payloads; during decoding, PV is decoded first and then used as a *hint* to decode the compressed payloads. Below is a brief overview of PerfOMR's compression phase. For a more detailed discussion, refer to [43, 46].

3.4.1 Unpacking the Pertinency Vector

In PerfOMR, before compressing the pertinency vector PV, each slot of PV is *unpacked* into separate ciphertexts. This process, referred to as PVUnpack in previous works [43, 46], is necessary for them because they compress PV by *hashing* its components into *buckets*. (For more details, see Sec. 3.4.2.) PVUnpack incorporates so-called SlotToCoeff [30, 47] and *oblivious expansion* [3] as subroutines, both of which are quite costly. As a result, PVUnpack becomes a bottleneck in the entire digesting process.

3.4.2 Hashing-based Index Compression

PerfOMR compresses the unpacked pertinency vector $\mathsf{PVUnpack}(\mathsf{PV}) = \{\mathtt{ct}_i\}_{i=1}^{N}$ via *hashing*. Each index $1 \le i \le N$ is randomly assigned to a *bucket* Y_j for $1 \le j \le m$. The pertinency vector is compressed by computing $\mathsf{Ctr}_j \leftarrow \sum_{i \in Y_j} \mathtt{ct}_i$ and $\mathsf{Acc}_j \leftarrow \sum_{i \in Y_j} i \cdot \mathtt{ct}_i$. Notice that Ctr_j indicates the number of pertinent indices that were hashed into Y_j, revealing whether a collision occurred. If Ctr_j encrypts the value 1, the corresponding value in Acc_j is the pertinent index; otherwise, a collision has occurred, and decoding will fail. To prevent such failures, PerfOMR sets the number of buckets m to be sufficiently larger than k, the upper bound on the number of pertinent indices, and repeats the process multiple times.

3.4.3 Payload Compression with Random Linear Codes

PerfOMR then compresses pertinent payloads using random linear codes. The core idea is to compress the vector of pertinent payloads (i.e., the payloads masked by the pertinency vector) by multiplying it with a random $K \times N$ matrix. By setting K sufficiently larger than k—the upper bound on the number of pertinent indices—it becomes possible to solve the system of equations for the pertinent payloads via Gaussian elimination with high probability, given the pertinent indices.

3.4.4 Bundling: Runtime/Digest-Size Trade-off

To reduce the computational overhead of PVUnpack, PerfOMR processes multiple indices simultaneously via *bundling*. Before applying PVUnpack, PerfOMR combines v indices of the pertinency vector by homomorphically adding them into a single ciphertext. While this reduces the runtime by a factor of v, it introduces a trade-off: now the receiver only knows that a bundle contains a pertinent index, requiring retrieval of all v indices and their corresponding payloads. This approach balances the reduction in PVUnpack runtime against an increase in the final digest size.

3.5 Optimizations

Modulus-Switching. *Modulus-Switching* [14] is a technique of switching the ciphertext modulus of a given HE ciphertext into a smaller one. While modulus-switching is not strictly necessary in the BFV scheme [13, 25] (unlike the BGV scheme [14]), it can be highly beneficial for performance optimizations. In BFV, choosing a large enough

ciphertext modulus is crucial to support the desired computation, as the modulus determines the *computational budget* of a ciphertext. However, a larger modulus increases the ciphertext size and degrades the overall performance of homomorphic operations. Modulus-switching addresses this issue by reducing the modulus as the budget is consumed, preserving the remaining budget. PerfOMR leverages modulus-switching repeatedly to keep the ciphertext modulus as small as possible without compromising the budget, enhancing performance.

Rotation Key Management. The BFV scheme can natively support all types of rotation. However, each requires its own rotation key and storing all these keys can be quite demanding in practice. Therefore, trade-offs between storage and computational time are often advantageous. It is also important to note that the size of rotation keys scales with the size of the ciphertext modulus. However, if the rotation keys will be used only after modulus-switching, generating them in a smaller modulus suffices. For this reason, PerfOMR reduces the number of rotation keys more aggressively for the ones used at higher ciphertext modulus levels. The approach taken by PerfOMR will be further detailed in Sec. 4.4.

4 Our Scheme: SophOMR

In this section, we present our new OMR scheme, SophOMR. While we largely follow the approach of PerfOMR [46], the key distinction lies in the compression phase (Sec. 3.4), where we extend and apply the SIMD-aware homomorphic compression scheme of [21] (Sec. 2.4). A detailed explanation of our homomorphic compression mechanism is provided in Sec. 4.1. Additionally, we propose several optimizations: ring-switching for digest-size reduction (Sec. 4.2), hybrid key-switching optimization for key-size reduction (Sec. 4.3), and BSGS-style Affine Transform step (Sec. 3.3) for run-time reduction (Sec. 4.4).

4.1 SIMD-Aware Digest Compression

As discussed in Sec. 3.4, PerfOMR [46] cannot fully exploit the homomorphic SIMD structure (Sec. 2.2.1) during the compression step due to its hashing-based approach (Sec. 3.4.2). As a result, it requires the costly PVUnpack process (Sec. 3.4.1), which unpacks each slot of the packed PV ciphertext into separate ciphertexts, creating a bottleneck for the entire Digest.

In this regard, we adopt a *SIMD-aware* homomorphic compression scheme from [21] (Sec. 2.4), which removes the need for the heavy PVUnpack step, significantly improving the runtime of the compression step.[8] However, [21] only considered payloads of the size of a single HE slot, which is unrealistic for many scenarios. For example, in prior works on OMR [43, 46], each payload is assumed to be 612 bytes, the typical size of Zcash transactions [35], meaning that the payload occupies multiple HE slots. Thus, we need to extend the approach of [21] to accommodate these typical OMR settings.

Naive Approach. To be precise, let each payload occupy t number of HE slots (e.g., $t \approx 300$ for typical HE parameters in the example of Zcash transactions). Let i-th slot payload vector PL_i denote the length-N vector consisting of the i-th slot of each payload. Directly applying the homomorphic compression of [21] (Sec. 2.4) to the compression step of OMR (Sec. 3.4), one would first compress the encrypted pertinency vector PV obtained from the detection step (Sec. 3.3). This involves homomorphically computing $\boldsymbol{C} \cdot \mathsf{PV}$ via MatMul (Alg. 1), where $\boldsymbol{C}$ is defined in Eq. 2. Then, one would compress *masked* payloads $\mathsf{PL}_i \odot \mathsf{PV}$ by computing $\boldsymbol{C} \cdot (\mathsf{PL}_i \odot \mathsf{PV})$ again with MatMul, for $1 \le i \le t$. This naive approach would require a total $(t+1)$ number of calls to MatMul with respect to the $k \times N$ matrix $\boldsymbol{C}$. While this method is already fairly efficient, it can be further optimized as described below.

Stacking Matrices. First, we note a simple yet powerful fact that enables our optimization. Let $\mathrm{diag}(\mathsf{PL}_i)$ denote the $N \times N$ diagonal matrix whose diagonal is PL_i and let $\boldsymbol{D}_i$ denote the $k \times N$ matrix $\boldsymbol{C} \cdot \mathrm{diag}(\mathsf{PL}_i)$. Then, the following holds.

$$\boldsymbol{C} \cdot (\mathsf{PL}_i \odot \mathsf{PV}) = \boldsymbol{D}_i \cdot \mathsf{PV}$$

This allows the detector to precompute $\boldsymbol{D}_i = \boldsymbol{C} \cdot \mathrm{diag}(\mathsf{PL}_i)$ and then compute $\boldsymbol{D}_i \cdot \mathsf{PV}$ homomorphically. Note that this preprocessing is almost free, as both $\boldsymbol{C}$ and $\mathrm{diag}(\mathsf{PL}_i)$ are *plaintexts* with regard to the underlying HE scheme. Furthermore, this approach consumes less *computational budget* (Sec. 3.5) as the homomorphic computation of $\mathsf{PL}_i \odot \mathsf{PV}$ is no longer performed before MatMul.

But much more importantly, this precomputation allows us to *stack* multiple MatMul into a single MatMul as follows.

$$\left[\boldsymbol{C}^T | \boldsymbol{D}_1^T | \cdots | \boldsymbol{D}_t^T\right]^T \cdot \mathsf{PV}$$

To see why this is useful at all, recall that the number of homomorphic rotations required in $k \times N$ MatMul is $O(\sqrt{k})$. (See Tab. 2.) The stacking method significantly reduces the total number of rotations required for compression: the naive approach, which involves $t+1$ calls to $k \times N$ MatMul, requires $O((t+1) \cdot \sqrt{k})$ rotations. In contrast, the stacking approach, which involves a single call to $((t+1) \cdot k) \times N$ MatMul, only requires $O(\sqrt{(t+1) \cdot k})$ rotations. This results in roughly a $\sqrt{t+1}$ reduction in the total number of rotations.

Implementing MatMul. We use MatMul described in Alg. 1 for digest compression. In particular, we employ the Horner-style [33] MatMul, which significantly reduces the

[8] Another advantage of our compression scheme is that, unlike PerfOMR's, it is *deterministic* and does not introduce any additional failure probability.

number of required rotation keys. Although this method is incompatible with the so-called *hoisting* technique [33], which optimizes runtime, we prioritize Horner-style MatMul. We believe that in scenarios where a detector handles multiple clients, minimizing the key size per client is much more important than the runtime improvements offered by hoisting.

Remark 4 (Cost Comparison for Digest Compression)**.** *A detailed comparison of the computational cost enabled by the simpler digest compression in SophOMR over the prior PerfOMR can be found in Tab.1 and Section 1.3.*

Remark 5. *Recall that our homomorphic compression scheme requires $p > N$ to ensure the correctness of the index compression step (Sec. 2.4.1). Although this condition may appear restrictive, the same assumption is implicit in prior OMR schemes [43, 46], which also encode indices up to N into the plaintext modulus p. Earlier works address this via p-ary index decomposition. While SophOMR is compatible with this approach, we do not need it in our parameters.*

4.2 Further Compression with Ring-Switching

Similar to modulus-switching (Sec. 3.5), there is a technique called *ring-switching* [14, 28, 29], which switches the ring degree n of a given HE ciphertext into a smaller one. This can be done while preserving the same security level after some modulus-switching.[9] In our case, ring-switching is useful for further reducing the digest size. Specifically, the ciphertext size scales linearly with the ring dimension n, which corresponds to the number of slots in the BFV scheme (Sec. 2.2.1). This implies that if the final result occupies fewer slots than n, the unused slots are being wasted. For instance, after homomorphic compression (Sec. 4.1), only $(t+1)\cdot k$ slots are occupied, which is typically smaller than n under standard OMR parameters. Ring-switching addresses this inefficiency by reducing the ring dimension and making the digest more compact.[10] In SophOMR, ring-switching is employed as the final step of Digest to further compress the digest. We note that ring-switching is a relatively inexpensive procedure, whose cost is dominated by a single key-switching operation. For further details, please refer to [29].

4.3 Key Size Reduction through Hybrid Key-Switching Optimization

The *key-switching* technique is a fundamental element of homomorphic multiplications and rotations. There are two main approaches to key-switching. The first, known as the *bit-decomposition* approach [8, 15] optimizes the *computational budget* (Sec. 3.5). In contrast, the *special-modulus* approach [19, 31] prioritize key-size optimization. In practice, a hybrid approach that combines these two methods is often most effective [31, 34, 39]. PerfOMR suffers from a large key size due to relying on one extreme of this hybrid design. In contrast, SophOMR balances the two extremes, significantly reducing the key size.

Table 3: Cost Analysis for Affine Transform

	#(Ptxt-Mul)	#(Rot)	#(Rot. Key)
PerfOMR	$s\cdot k$	k	1
Our Approach	$s\cdot k$	$\tilde{b}+s\cdot\tilde{g}$	2

4.4 Affine Transform with BSGS-MatMul

We optimize the runtime of Affine Transform step in the Detect procedure (Sec. 3.3) by adopting the BSGS-style MatMul algorithm, which uses two rotation keys. Thanks to hybrid key-switching (Sec. 4.3), this optimization is achieved while maintaining a smaller key-size compared to PerfOMR's approach of using a single rotation key. (See Sec. 5.)

Recall that the Affine Transform step involves homomorphic matrix multiplications between *tall* plaintext matrices $\boldsymbol{A}_j$'s and the ciphertext vector $\mathtt{ct}_{\mathtt{sk}}$ (Sec. 3.3, Alg. 6). For clarity, in this section, we consider the goal of homomorphically multiplying a tall $N\times k$ plaintext matrix $\boldsymbol{M}$ (representing $\boldsymbol{A}_j$) with a ciphertext $\mathtt{ct}(\boldsymbol{v})$ (representing $\mathtt{ct}_{\mathtt{sk}}$) that encrypts a vector $\boldsymbol{v}\in\mathbb{Z}_p^k$ via sparse packing (Sec. 2.2.2).

4.4.1 PerfOMR's Approach: MatMul with Single Key

Note that the Affine Transform (Sec. 3.3) is the very first step of digesting and occurs before any modulus-switching (Sec. 3.5). As a result, the rotation keys used in this step must be generated at the largest modulus, making them relatively large. In this regard, PerfOMR [46] aggressively minimizes the number of rotation keys required during the Affine Transform, ultimately using only one. The approach skips BSGS-style optimization (Sec. 2.3) and applies Eq. 1 directly. The cost of PerfOMR is summarized in Tab. 3.

4.4.2 Our Approach: MatMul with Two Keys

In SophOMR, thanks to hybrid key-switching (Sec. 4.3), the size of the rotation key generated at the largest modulus is substantially smaller than in PerfOMR. As a result, even with the BSGS-style optimization and the use of two keys, SophOMR's total key-size remains smaller than PerfOMR's. The adoption of BSGS reduces the number of homomorphic rotations from $O(k)$ to $O(\sqrt{k})$, enhancing runtime performance. The BSGS-MatMul algorithm used for Affine trans-

[9] For the same level of security, a larger ring dimension is required when using a larger ciphertext modulus.

[10] Ring-switching is especially effective in the context of OMD (Sec. 2.5.2), where the digest contains only the pertinent indices, excluding payloads, requiring significantly fewer slots. See Sec. 5.

form in SophOMR is simply the tall matrix version of Alg. 1. The costs for our approach are summarized in Tab. 3. Note that the number of rotations is minimized when $\tilde{b} \approx \sqrt{s \cdot k}$.

4.5 Security of SophOMR

We sketch the security of SophOMR with respect to Def. 3.

- **Correctness:** The correctness directly follows directly from the correctness of the underlying BFV scheme (Sec. 2.2.1, Def. 1) and the completeness/soundness of the underlying Private Signaling scheme (Construction 1, Def. 7). It is worth noting that our digest compression (Sec. 4.1) is deterministic and guarantees correct output.
- **Receiver-Privacy:** The receiver-privacy directly follows from the semantic security of the underlying BFV scheme (Sec. 2.2.1, Def. 1) and the receiver-privacy of the underlying Private Signaling scheme (Construction 1, Def. 7). More precisely, let $\mathtt{pp} \leftarrow \mathsf{ParamGen}(1^\lambda)$ and sample two set of keys $(\mathtt{sk}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}})$ and $(\mathtt{sk}', \mathtt{pk}_{\mathtt{sig}}', \mathtt{pk}_{\mathtt{det}}')$ from $\mathsf{KeyGen}(\mathtt{pp})$ (see Sec. 3.2). Two signals $\mathtt{sig} \leftarrow \mathtt{Signal}(\mathtt{pk}_{\mathtt{sig}})$ and $\mathtt{sig}' \leftarrow \mathtt{Signal}(\mathtt{pk}_{\mathtt{sig}}')$ are indistinguishable, even given the auxiliary information $\mathtt{aux} = (\mathtt{pp}, \mathtt{pk}_{\mathtt{sig}}, \mathtt{pk}_{\mathtt{det}}, \mathtt{pk}_{\mathtt{sig}}', \mathtt{pk}_{\mathtt{det}}')$, due to the receiver-privacy of the Private Signaling scheme. This holds because $\mathtt{pk}_{\mathtt{det}} = (\mathtt{pk}_{\mathtt{BFV}}, \mathtt{BFV.Enc}(\mathtt{sk}_{\mathtt{PS}}))$, which is the only component dependent on the secret key $\mathtt{sk}_{\mathtt{PS}}$ of the Private Signaling scheme, reveals no information about $\mathtt{sk}_{\mathtt{PS}}$ by the semantic security of the BFV scheme.

4.6 OMD Variant

Our SophOMR scheme can be readily adapted into an Oblivious Message Detection (OMD) scheme (Sec. 2.5.2). Since OMD does not require payload compression, it is sufficient to compress only the encrypted pertinency vector PV. This is done by homomorphically computing $\boldsymbol{C} \cdot \mathsf{PV}$ via MatMul (Alg. 1), where $\boldsymbol{C}$ is defined in Eq. 2, without stacking the matrices as described in Sec. 4.1. This leads to a significant speedup in the compression phase (Sec. 4.1). See Sec. 5 for concrete performance results.

5 Evaluation

Methodology. To evaluate the concrete performance of our SophOMR scheme, we implement it in a C++ library[11] using the OpenFHE library [7], with the NTL library [60] to implement Decode. We then compare its performance against the implementation [44] of PerfOMR [46], which uses PALISADE library [54] for the private signaling scheme (Sec. 3.1), the SEAL library [58] for the BFV scheme (Sec. 2.2.1). Both implementations leverage the HEXL library [12] to accelerate HE libraries. All experiments for SophOMR and PerfOMR were conducted single-threaded on a Google Compute Cloud `n2-standard-8` instance (Intel Ice Lake). Each experiment was repeated five times, and we report the average.

Parameters. We choose parameters that uphold the security claims of PerfOMR's parameter sets. All bit security is measured using the latest version of the Lattice Estimator[12] [2] available at the time of writing (Commit ID: `5c25455`). Detailed parameters are provided in Appendix A. Note that SophOMR uses a larger BFV ring dimension of $n = 2^{16}$, compared to $n = 2^{15}$ of PerfOMR, to meet 128-bit security (Tab. 5). We also employ a larger parameter for the underlying private signaling scheme (Sec. 3.1) that offers 128-bit security and 30-bit completeness to uphold the security claim of PerfOMR's parameters. The parameters provided in PerfOMR only achieve 94-bit security and 18-bit completeness, leading to easily observable errors (Tab. 7).

Experiments. Experiments are conducted with a total number of payloads $N = 2^{16}$ and 2^{19}. Following PerfOMR, the number of pertinent payloads is set to $k = 50$, and each payload is assumed to be 612 bytes, which corresponds to the typical size of Zcash transactions [35]. For $N = 2^{16}$, we present the results in Tab. 4. Additionally, Fig. 2 illustrate the breakdown of each component contributing to the Digest runtime: Preprocessing (Rmk. 8), Affine Transform (Sec. 4.4), Range Check (Sec. 3.3), and Compress (Sec. 4.1). Refer to Appendix B, for experimental results for $N = 2^{19}$.

Remark 6 (On Choice of N)**.** *While PerfOMR conducted experiments with $N = 2^{19}, 2^{21}, 2^{23}$, we focus on $N = 2^{16}, 2^{19}$, considering $N = 2^{16}$ as the primary setting. Our reasoning is that dividing N into smaller sets can be advantageous for* latency*, provided it maintains similar throughput and keeps communication costs reasonable. For example, PerfOMR uses $N = 2^{19}$ to model Bitcoin-scale applications,[13] but a client may find it more useful to receive digests for 2^{16} payloads every 3 hours than a single digest for 2^{19} every 24 hours. This issue is even more relevant in private messaging scenarios.*

Remark 7 (Scaling with N)**.** Digest *runtime for both PerfOMR and SophOMR scales approximately linearly with the number of payloads N, as shown in experiments with $N = 2^{16}$ (Fig. 2, Tab. 4) and $N = 2^{19}$ (Fig. 4, Tab. 8). By contrast, the* Decode *runtime and the resulting digest size are independent of N, depending only on the number of pertinent messages k.*

Remark 8 (Preprocessing)**.** *In Fig. 2 and 4, the blue mesh represents the time that PerfOMR deliberately excluded from*

[11] https://github.com/keewoolee/SophOMR

[12] https://github.com/malb/lattice-estimator

[13] There are roughly 2^{19} transactions per day on Bitcoin: https://ycharts.com/indicators/bitcoin_transactions_per_day

Table 4: Performance of OMR/OMD Schemes ($N = 2^{16}$, $k = 50$, single-thread)

	Digest (s)	Decode (ms)	Detection Key (MB)[14]	Digest (KB)[15]
PerfOMR ($v = 2$)	541 (std: 2.9)	14.6 (std: 0.5)	171	567
SophOMR	157 (std: 0.4)	14.0 (std: 0.1)	114	263
PerfOMD ($v = 2$)	493 (std: 1.4)	6.8 (std: 0.4)	171	284
SophOMD	107 (std: 0.3)	5.5 (std: 0.1)	142	132

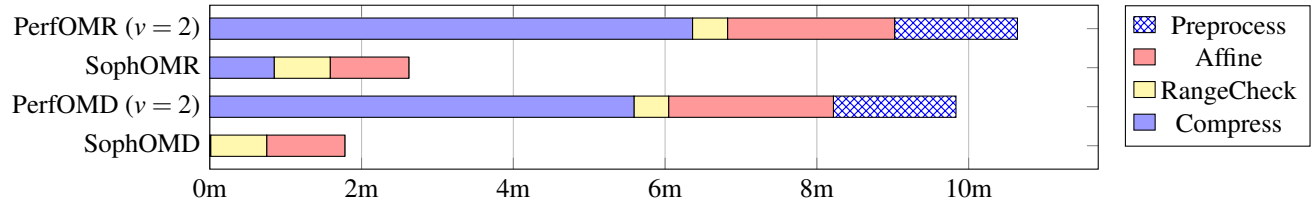


Figure 2: Runtime Breakdown for OMR/OMD Schemes ($N = 2^{16}$, $k = 50$)

their end-to-end timing results to account for the preprocessing potential of a significant portion of SlotToCoeff *(Sec. 3.4.1). However, this preprocessing requires a substantial amount of RAM. If such an amount of RAM is available, SophOMR can also be accelerated by preprocessing MatMul in Compress (Sec. 4.1). For a comparison in favor of PerfOMR, preprocessing time is excluded from Tab. 4 and 8.*

Experimental Results for $N = 2^{16}$. As shown in Tab. 4, SophOMR provides multiple improvements over PerfOMR. SophOMR achieves a 3.4× speedup in Digest runtime while maintaining a reasonably small Decode runtime. The detection key size is reduced by 1.5× through hybrid key-switching optimization (Sec. 4.3). Furthermore, the digest size is reduced by 2.2×, thanks to our compact homomorphic compression scheme (Sec. 4.1) and ring-switching (Sec. 4.2) using Param II in Tab. 6; see also Rmk. 10. These improvements are achieved despite SophOMR using larger parameters than PerfOMR to meet security levels that PerfOMR does not satisfy (See Appendix A).

As illustrated in Fig. 2, the improvement in Digest runtime primarily stems from our SIMD-aware homomorphic compression scheme (Sec. 4.1). Specifically, our compression achieves 7.5× speedup compared to PerfOMR, even when excluding the runtime for preprocessing (Rmk. 8). The Affine Transform step also achieves considerable speedup thanks to BSGS-style MatMul (Sec. 4.4). On the other hand, the Range Check step (Sec. 3.3) shows a slight increase in runtime, attributed to our use of larger parameters to meet security levels that PerfOMR does not satisfy (See Appendix A).

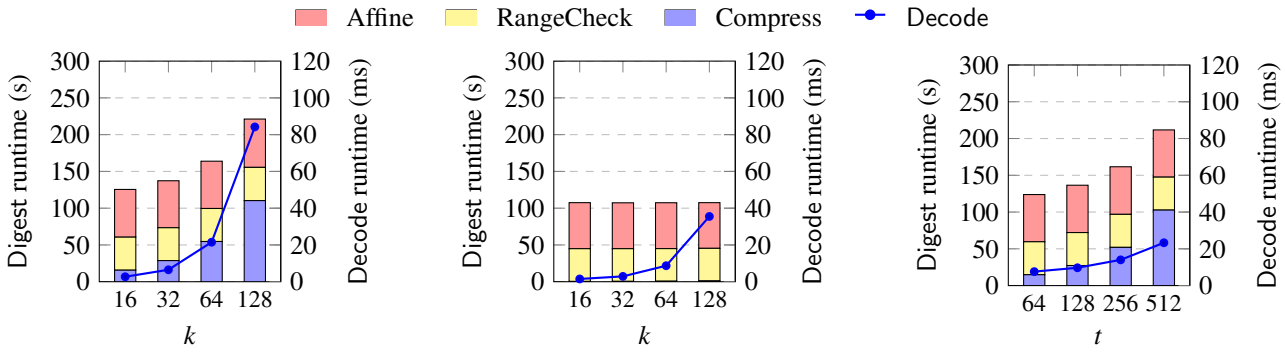
Remark 9 (On Choice of v)**.** *The bundling parameter v in PerfOMR represents how many payloads are concatenated and processed simultaneously (See Sec. 3.4.4). We compare only with PerfOMR using $v = 2$ for $N = 2^{16}$ because (i) setting $v = 2$ incurs nearly no additional runtime or digest size costs compared to $v = 1$ due to PerfOMR's specific utilization BFV slots, and (ii) setting $v > 2$ does not improve runtime further due to the fixed number of BFV slots ($n = 2^{15}$).*

Remark 10 (Digest Size)**.** *PerfOMR optimizes the digest size by setting the BFV ciphertext modulus at the lowest level to be as small as possible in the SEAL library (Sec. 3.5). However, this optimization is not applied in our SophOMR implementation due to limitations in the OpenFHE library: (i) it does not officially allow the base-level ciphertext modulus to be set independently of the RNS limbs, and (ii) its ciphertext serialization does not further reduce when the ciphertext modulus is below* 64*-bit. If we apply the same optimization as in PerfOMR, SophOMR's digest size can be reduced by approximately an additional factor of* 2×*.*

Remark 11 (Multiplicative Depth)**.** *Recall that* Digest *consists of three phases: Affine Transform, Range Check, and Compress (Sec. 3.3 and 3.4). PerfOMR consumes multiplicative depths of 1, 22, and 2 for these phases, respectively, while SophOMR uses 1, 27, and 1. Although the Range Check phase consumes substantial multiplicative depth, its circuit requires only a few key-switching operations per level compared to the Affine Transform and Compress phases. As a result, it is not the bottleneck in either scheme.*

OMD Variants. We also present the performance of SophOMD, the OMD variant of SophOMR (Sec. 2.5.2, Sec. 4.6). As there is no existing OMD implementation corresponding to PerfOMR, we modified the PerfOMR implementation to run only the relevant components and refer to this version as PerfOMD. The OMD experiments use the same

[14] We account for the compression of randomness into the hash seed.
[15] Our digest size can be further reduced, as explained in Rmk. 10.



(a) SophOMR with varying k ($t = 250$) (b) SophOMD with varying k (c) SophOMR with varying t ($k = 50$)

Figure 3: Runtime of SophOMR/SophOMD under varying parameters ($N = 2^{16}$, single-thread)

BFV parameters as those in the OMR experiments, except that the smaller ring-switching parameter set Param I from Tab. 6 is used.

As shown in Tab. 4, SophOMD also provides multiple improvements over PerfOMD, similar to the OMR setting. Specifically, SophOMD achieves a 4.6× speedup in Digest runtime. As illustrated in Fig. 2, this improvement primarily stems from our SIMD-aware homomorphic compression scheme (Sec. 4.1), allowing compression in SophOMD to complete in only one second – a 479× speedup compared to PerfOMD. Notice that the performance gain is even greater than in the OMR setting, as we only compress the indices and not the payloads. On the other hand, the detector key size has slightly increased (though still smaller than PerfOMR) due to the use of additional rotation keys from the reduced number of rows in MatMul (Tab. 2). The bundling technique (Sec. 3.4.4) can still be applied to PerfOMD; however, achieving OMR-like functionality (Sec. 2.5.2) requires performing PIR (Private Information Retrieval) on the *bundles*, causing PIR performance to depend on the bundling parameter v.

Additional Experiments with Varying k and t. To evaluate their impact on performance, we conduct additional experiments that vary the number of pertinent payloads k (Fig. 3a and 3b) and the number of slots per payload t (Fig. 3c). We use different ring-switching parameters, as presented in Tab. 6 in Appendix A, depending on the required number of slots for the digest. Specifically, SophOMR uses Param III for $(k,t) = (128,250)$ and $(50,512)$, Param II for $(k,t) = (64,250)$ and $(50,256)$, while SophOMD and all remaining SophOMR experiments use Param I.

Regarding Digest, we observe that the runtime of Compress scales linearly with both k and t. This indicates that the performance bottleneck of SophOMR's Digest Compression lies at plaintext-ciphertext multiplications rather than homomorphic multiplications (See Tab. 1). We note that the runtimes for RangeCheck and Affine are independent of k and t. In contrast, the Decode runtime is affected by multiple factors. Specifically, Decode consists of a decryption step that is (quasi-)linear in the digest size, primarily determined by the ring-switching dimension, followed by a decompression step that scales linearly with t and super-linearly with k.

Acknowledgments

Yongdong Yeo was financially supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2025-02217656). We would like to thank Zeyu Liu for the discussion on PerfOMR and its implementation.

Ethical Considerations

This research focuses on improving the efficiency and scalability of Oblivious Message Retrieval (OMR), with the goal of enhancing the user experience in anonymous messaging services. As such, OMR inherently involves ethical considerations related to anonymity and its potential societal impact.

Anonymity can protect users—particularly in surveillance-heavy or oppressive environments—but it may also enable harmful or illicit communication. From a stakeholder perspective, potential risks arise for platform operators, who may face challenges in moderating malicious content, as well as for individuals or communities that may be harmed by those who exploit anonymity. At the same time, enhanced privacy and anonymity can benefit other stakeholders, including platforms seeking to serve end users without collecting or retaining unnecessary personal information. In a broader context, prior work [1, 40, 41] highlights the need to evaluate privacy-preserving technologies, such as anonymous messaging systems and privacy-oriented blockchains, within frameworks of regulatory compliance and responsible governance, emphasizing the importance of balancing user anonymity with legitimate oversight.

Our efficiency improvements lower the barrier to deploying

OMR in practice, which amplifies the importance of these ethical considerations. To support responsible adoption, we provide a transparent and reproducible system design and evaluation methodology. All experiments were conducted in controlled environments and did not involve any live systems.

Open Science

In line with open science principles, we are committed to making our research fully transparent and accessible. To support reproducibility and independent verification, we include comprehensive details of our experimental setup and results in the paper, along with the accompanying code available at: https://doi.org/10.5281/zenodo.17958466.

References

[1] Oluwafemi Akanfe, Diane Lawong, and H. Raghav Rao. Blockchain technology and privacy regulation: Reviewing frictions and synthesizing opportunities. *International Journal of Information Management*, 76:102753, 2024.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

[4] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In Kimberly Keeton and Timothy Roscoe, editors, *USENIX OSDI 2016*. USENIX Association, November 2016.

[5] Aztec. https://aztec.network/.

[6] Adam Back, Ulf Möller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In Ira S. Moskowitz, editor, *Information Hiding, 4th International Workshop, IHW 2001, Proceedings*, volume 2137 of *Lecture Notes in Computer Science*, pages 245–257. Springer, April 2001.

[7] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[8] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 423–442. Springer, Cham, August 2016.

[9] Gabrielle Beck, Julia Len, Ian Miers, and Matthew Green. Fuzzy message detection. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1507–1528. ACM Press, November 2021.

[10] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[11] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Batch PIR and labeled PSI with oblivious ciphertext compression. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[12] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). https://github.com/intel/hexl, 2021.

[13] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Berlin, Heidelberg, August 2012.

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[15] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Berlin, Heidelberg, August 2011.

[16] Vitalik Buterin. Exploring fully homomorphic encryption. https://vitalik.eth.limo/general/2020/07/20/homomorphic.html, 2020.

[17] Vitalik Buterin. Open problem: improving stealth addresses. https://ethresear.ch/t/open-problem-improving-stealth-addresses/7438, 2020.

[18] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 395–412. ACM Press, November 2019.

[19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J.: Jacobson, Jr., editors, *SAC 2018*, volume 11349 of *LNCS*, pages 347–368. Springer, Cham, August 2019.

[20] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.

[21] Jung Hee Cheon, Keewoo Lee, Jai Hyun Park, and Yongdong Yeo. SIMD-aware homomorphic compression and application to private database query, 2024. https://arxiv.org/abs/2408.17063.

[22] Seung Geol Choi, Dana Dachman-Soled, S. Dov Gordon, Linsheng Liu, and Arkady Yerukhimovich. Compressed oblivious encoding for homomorphically encrypted search. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2277–2291. ACM Press, November 2021.

[23] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995.

[24] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE Computer Society Press, May 2015.

[25] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[26] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. How to compress encrypted data. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 551–577. Springer, Cham, April 2023.

[27] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[28] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in BGV-style homomorphic encryption. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 19–37. Springer, Berlin, Heidelberg, September 2012.

[29] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Field switching in bgv-style homomorphic encryption. *J. Comput. Secur.*, 21(5):663–684, 2013.

[30] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 1–16. Springer, Berlin, Heidelberg, May 2012.

[31] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Berlin, Heidelberg, August 2012.

[32] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 554–571. Springer, Berlin, Heidelberg, August 2014.

[33] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 93–120. Springer, Cham, August 2018.

[34] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In Stanislaw Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 364–390. Springer, Cham, February 2020.

[35] Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification. https://zips.z.cash/protocol/protocol.pdf.

[36] Sashidhar Jakkamsetti, Zeyu Liu, and Varun Madathil. Scalable private signaling. In *38th IEEE Computer Security Foundations Symposium, CSF 2025*, pages 143–158. IEEE, June 2025.

[37] Yanxue Jia, Varun Madathil, and Aniket Kate. HomeRun: High-efficiency oblivious message retrieval, unrestricted. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 2012–2026. ACM Press, October 2024.

[38] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1651–1669. USENIX Association, August 2018.

[39] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 608–639. Springer, Cham, December 2021.

[40] Rob Kling, Ya-Ching Lee, Al Teich, and Mark S. Frankel. Assessing anonymous communication on the internet: Policy deliberations. *The Information Society*, 15(2):79–90, 1999.

[41] Yannan Li, Willy Susilo, Guomin Yang, Yong Yu, Xiaojiang Du, Dongxi Liu, and Nadra Guizani. Toward privacy and regulation in blockchain-based cryptocurrencies. *IEEE Network*, 33(5):111–117, 2019.

[42] Zeyu Liu, Katerina Sotiraki, Eran Tromer, and Yunhao Wang. Snake-eye resistant PKE from LWE for oblivious message retrieval and robust encryption. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part III*, volume 15603 of *LNCS*, pages 126–156. Springer, Cham, May 2025.

[43] Zeyu Liu and Eran Tromer. Oblivious message retrieval. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 753–783. Springer, Cham, August 2022.

[44] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: proof of concept C++ implementation for OMR (Oblivious Message Retrieval) with Reduced Communication and Computation. https://github.com/ObliviousMessageRetrieval/ObliviousMessageRetrieval/tree/perfomr.

[45] Zeyu Liu, Eran Tromer, and Yunhao Wang. Group oblivious message retrieval. In *2024 IEEE Symposium on Security and Privacy*, pages 4367–4385. IEEE Computer Society Press, May 2024.

[46] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: Oblivious message retrieval with reduced communication and computation. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[47] Zeyu Liu and Yunhao Wang. Amortized functional bootstrapping in less than 7 ms, with $\tilde{O}(1)$ polynomial multiplications. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 101–132. Springer, Singapore, December 2023.

[48] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy*, pages 1057–1073. IEEE Computer Society Press, May 2021.

[49] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Berlin, Heidelberg, May / June 2010.

[50] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, Berlin, Heidelberg, May 2013.

[51] Varun Madathil, Alessandra Scafuro, István András Seres, Omer Shlomovits, and Denis Varlakov. Private signaling. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3309–3326. USENIX Association, August 2022.

[52] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In David M. Martin Jr. and Andrei Serjantov, editors, *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Revised Selected Papers*, volume 3424 of *Lecture Notes in Computer Science*, pages 17–34. Springer, May 2004.

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

[54] PALISADE Lattice Cryptography Library (release 1.11.3). https://palisade-crypto.org/, May 2021.

[55] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, Berlin, Heidelberg, August 2008.

[56] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[57] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

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

[59] István András Seres, Balázs Pejó, and Péter Burcsi. The effect of false positives: Why fuzzy message detection leads to fuzzy privacy guarantees? In Ittay Eyal and Juan A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 123–148. Springer, Cham, May 2022.

Table 5: Parameters for BFV

	n	p	$\log Q$	$\log PQ$	λ
PerfOMR	2^{15}	65537	857	917	120.9
SophOMR	2^{16}	786433	1140	1740	128.6

Table 6: BFV Parameters for Ring-Switching

	n	$\log Q$	$\log PQ$	λ
Param I	2^{13}	120	180	154.4
Param II	2^{14}	120	240	244.6
Param III	2^{15}	120	240	536.1

[60] Victor Shoup. NTL: A Library for doing Number Theory. https://libntl.org/.

[61] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015*, pages 137–152. ACM, October 2015.

[62] Toni Wahrstätter, Matt Solomon, Ben DiFrancesco, and Vitalik Buterin. ERC-5564: Stealth Addresses. Ethereum Improvement Proposals, no. 5564, August 2022. Available: https://eips.ethereum.org/EIPS/eip-5564.

[63] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In Chandu Thekkath and Amin Vahdat, editors, *USENIX OSDI 2012*. USENIX Association, October 2012.

A Parameters

The BFV parameters used in our experiments are presented in Tab. 5. For SophOMR and SophOMD, we also present the BFV parameters for ring-switching (Sec. 4.2) in Tab. 6. The private signaling parameters are presented in Tab. 7.

BFV Parameters. In Tab. 5 and Tab. 6, $\log PQ$ denotes the largest modulus used in the scheme, including *special modulus* used in the hybrid key-switching (Sec. 4.3), which impacts the security level. For all BFV parameter sets, the error distribution is set to a discrete Gaussian with standard deviation $\sigma = 3.19$. In Tab. 5, notice that SophOMR uses a larger BFV ring dimension of $n = 2^{16}$, compared to $n = 2^{15}$ of PerfOMR, to meet 128-bit security. In this setup, a larger plaintext modulus p is required to satisfy $p = 1 \pmod{2n}$ to fully leverage the homomorphic SIMD structure.

Private Signaling Parameters. In Tab. 7, ρ and ν denote the false-positive and false-negative security parameters, which are associated with soundness and completeness, respectively (Def. 7). Note that our key and signal sizes are slightly larger than PerfOMR due to our larger BFV parameter, which is to meet 128-bit security.

B Experimental Results for $N = 2^{19}$

In this section, we present our experimental results for $N = 2^{19}$. We compare the performance of SophOMR/SophOMD with PerfOMR/PerfOMD for $v = 2$ and $v = 16$, representing the two extremes in the trade-off between digest size and runtime. (See Sec. 3.4.4 and Rmk. 9.) As shown in Tab. 8 and Fig. 4, SophOMR/SophOMD provides multiple improvements over PerfOMR/PerfOMD for $N = 2^{19}$.

OMR Schemes. Compared to PerfOMR with $v = 2$, SophOMR achieves a 3.4× speedup in Digest runtime and a 2.2× reduction in digest size. In contrast, compared to PerfOMR with $v = 16$, SophOMR achieves 1.2× speedup in Digest runtime and a 10.8× reduction in digest size. Additionally, note that the Decode runtime for PerfOMR increases when $v = 16$.

OMD Schemes. For the OMD variants, SophOMD achieves a 4.5× and 1.4× speedup in Digest runtime compared to PerfOMD with $v = 2$ and $v = 16$, respectively. The other trends are observed similarly as in the $N = 2^{16}$ setting. Specifically, our SIMD-aware homomorphic compression scheme (Sec. 4.1) enables SophOMD to complete compression for $N = 2^{19}$ in just 2.6 seconds—yielding a 1033× speedup over PerfOMD with $v = 2$ and a 145× speedup with $v = 16$.

[16] We note that the range parameter r is chosen to be as small as possible while satisfying the false-negative bound ν. More precisely, r is set to satisfy $1 - \mathrm{erf}(r/(\sigma\sqrt{2 \cdot (2h+1)})) \leq 2^{-\nu}/\ell$. For details, we refer to Sec. 5.1 of [46].

[17] We account for the compression of randomness into the hash seed.

Table 7: Parameters for PS

	n'	q'	σ	h	ℓ	r[16]	λ	ρ	ν	PS Key (KB)[17]	Signal (KB)
PerfOMR	1024	65537	0.5	32	2	19	94.0	21.4	17.6	2.2	2.2
SophOMR	1024	786433	0.5	80	2	40	128.4	26.5	30.7	2.6	2.6

Table 8: Performance of OMR/OMD Schemes ($N = 2^{19}$, $k = 50$)

	Digest (s)	Decode (ms)	Detection Key (MB)	Digest (KB)
PerfOMR ($v = 2$)	3704 (std: 40.0)	15.0 (std: 0.0)	171	568
PerfOMR ($v = 16$)	1368 (std: 3.5)	72.0 (std: 0.7)	171	2840
SophOMR	1098 (std: 1.7)	12.0 (std: 4.7)	114	263
PerfOMD ($v = 2$)	3328 (std: 7.4)	7.0 (std: 0.0)	171	284
PerfOMD ($v = 16$)	1028 (std: 7.1)	10.0 (std: 0.0)	171	568
SophOMD	745 (std: 6.8)	5.5 (std: 0.2)	142	132

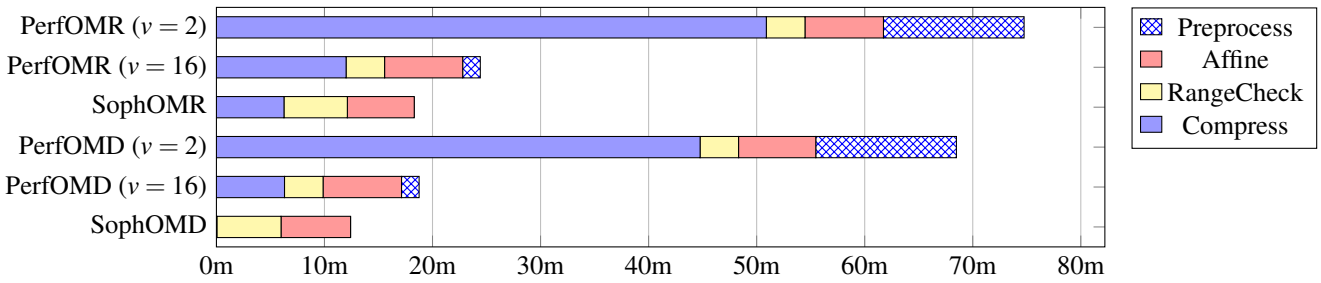


Figure 4: Runtime Breakdown for OMR/OMD Schemes ($N = 2^{19}$, $k = 50$, single-thread)



USENIX Security '26 Artifact Appendix: SophOMR: Improved Oblivious Message Retrieval from SIMD-Aware Homomorphic Compression

Keewoo Lee
Ethereum Foundation

Yongdong Yeo
Seoul National University

A Artifact Appendix

A.1 Abstract

We provide a C++ reference implementation of the Oblivious Message Retrieval (OMR) and Oblivious Message Detection (OMD) schemes, referred to as SophOMR and SophOMD, respectively, in the accompanying paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any security, privacy, or physical risks to evaluators, nor does it raise ethical concerns. The payloads used to simulate data posted on a bulletin board are randomly generated.

A.2.2 How to access

The artifact is available at: https://doi.org/10.5281/zenodo.17958466.

A.2.3 Hardware dependencies

The artifact has no specific hardware dependencies and can be run on standard platforms. However, the HEXL backend is optimized for processors with AVX512_IFMA support (e.g., Intel Ice Lake). The benchmarks reported in our paper were run single-threaded on a Google Compute Cloud `n2-standard-8` instance (Intel Ice Lake, 2.6 GHz CPU with 32 GB RAM[1]).

A.2.4 Software dependencies

The artifact is based on the following dependencies:

- C++ build environment
- CMake build infrastructure
- NTL library
- OpenFHE library (Tested with v1.4.0[2])
- HEXL library (Optional; Skip on ARM)

The benchmarks reported in our paper were run on Ubuntu 20.04 LTS with OpenFHE v1.2.0.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The commands below are based on Ubuntu 24.04 LTS with x86-64 and assume the use of the HEXL backend.

1. Install CMake, GMP, and NTL (if needed).

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install cmake
sudo apt-get install libgmp3-dev
sudo apt-get install libntl-dev
```

2. Install OpenFHE + HEXL. Follow the instructions at https://github.com/openfheorg/openfhe-hexl, or run:

```
git clone https://github.com/openfheorg/
    openfhe-configurator.git
cd openfhe-configurator
# To reproduce the version used in the paper, run:
    git reset --hard 9ac9090
scripts/configure.sh

# Would you like to stage an openfhe
    -development build?      [y/n] : n  (y on ARM)
# Would you like to stage an openfhe-hexl
    build?                   [y/n] : y

sudo scripts/build-openfhe-development.sh
```

[1] The installation of OpenFHE and HEXL may require more than 32 GB of RAM (we were able to install using 64 GB RAM). After installation, however, SophOMR runs comfortably with 32 GB of RAM.

[2] To implement ring-switching, we use OpenFHE in a manner not officially supported by its APIs, which may be incompatible with OpenFHE versions beyond 1.4.0.

3. Build the Library.

```
cd ..  # if still in openfhe-configurator
    directory
git clone https://github.com/keewoolee/
    SophOMR.git  # clone the repository
cd SophOMR
mkdir build
cd build
cmake .. -DCMAKE_PREFIX_PATH=~/openfhe
    -configurator/openfhe-staging/install
    # adjust the path to the location of the
    openfhe libraries
make
```

A.3.2 Basic Test

The following command runs the fastest predefined configuration (1-2 compute-minutes). On success, the final line of the output will be: `Result is Correct!`

```
./test OMD 65536 16
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *SophOMR achieves, compared to the prior state of the art PerfOMR, reductions of* 3.4× *in runtime,* 2.2× *in digest size, and* 1.5× *in key size in a setting with 65,536 payloads (each 612 bytes), of which up to 50 are pertinent. In particular, the runtime of the homomorphic compression step is reduced by* 7.5×*. These results are reported in Tab. 4 and Fig. 2, together with the corresponding OMD variants (SophOMD and PerfOMD). Similar trends are observed for 524,288 payloads (Tab. 8 and Fig. 4). The SophOMR and SophOMD runtimes presented in the tables and figures are reproducible using Experiment E1.*[3]

(C2): *SophOMR and SophOMD exhibit trends consistent with Fig. 3 of our paper for varying values of k and t. The SophOMR and SophOMD runtimes presented in Fig. 3 are reproducible using Experiment E1.*

A.4.2 Experiments

(E1): *Full Benchmarks (6-7 compute-hours): This experiment runs all benchmarks presented in the paper and supports Claims C1 and C2. Each benchmark is repeated five times, and we report the average runtime along with the standard deviation.*
Execution: *In the build directory, run the following command:*

```
python3 -u ../benchmark.py > benchmark.txt 2>&1
```

Results: *The file* `benchmark.txt` *contains the timing results and corresponding standard deviations for all benchmarks presented in the paper. The results are organized by the figures and tables in which they appear. When run in the same environment described in Section A.2.3 and A.2.4, the output should reproduce the reported results; in other environments, it is expected to follow similar performance trends.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.

References

[1] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: proof of concept C++ implementation for OMR (Oblivious Message Retrieval) with Reduced Communication and Computation. https://github.com/ObliviousMessageRetrieval/ObliviousMessageRetrieval/tree/perfomr.

[3] PerfOMR is outside the scope of this artifact. The runtime results reported in our paper can be obtained using the original PerfOMR implementation [1]. To ensure reproducibility, we provide a fork of the original repository with minor modifications necessary for our evaluation, which reproduces the results by following the provided instructions. The fork is available at: https://github.com/keewoolee/PerfOMR.