PerfOMR: Oblivious Message Retrieval with Reduced Communication and Computation

Zeyu Liu
Yale University

Eran Tromer
Boston University

Yunhao Wang
Yale University

Abstract

Anonymous message delivery, as in privacy-preserving blockchain and private messaging applications, needs to protect recipient metadata: eavesdroppers should not be able to link messages to their recipients. This raises the question: how can untrusted servers assist in delivering the pertinent messages to each recipient, without learning which messages are addressed to whom?

Recent work constructed Oblivious Message Retrieval (OMR) protocols that outsource the message detection and retrieval in a privacy-preserving way, using homomorphic encryption. Their construction exhibits significant costs in computation per message scanned ($\sim$0.1 second), as well as in the size of the associated messages ($\sim$1kB overhead) and public keys ($\sim$132kB).

This work constructs more efficient OMR schemes, by replacing the LWE-based clue encryption of prior works with a Ring-LWE variant, and utilizing the resulting flexibility to improve several components of the scheme. We thus devise, analyze, and benchmark two protocols:

The first protocol focuses on improving the detector runtime, using a new retrieval circuit that can be homomorphically evaluated 15x faster than the prior work.

The second protocol focuses on reducing the communication costs, by designing a different homomorphic decryption circuit that allows the parameter of the Ring-LWE encryption to be set such that the public key size is about 235x smaller than the prior work, and the message size is roughly 1.6x smaller. The runtime of this second construction is $\sim$40.0ms per message, still more than 2.5x faster than prior works.

1 Introduction

Protecting message contents in messaging applications has been extensively studied, with the wide usage of end-to-end encryption today. However, metadata (who sent and received messages, and when) can by itself disclose sensitive information. Therefore, protecting metadata is essential to anonymous message delivery systems like private messaging [2, 5, 10, 25, 40]. The importance is further amplified in privacy-preserving cryptocurrencies [3, 6, 18, 29], where the ledger containing all the messages is permissionless and widely replicated, making it easily accessible to everyone.

Among the various critical pieces of metadata, *recipient* privacy is a particularly challenging problem to efficiently solve. From a recipient's perspective, a transaction *pertinent* to them could be located anywhere on the ledger in the blockchain applications (or a queue in private messaging systems). Consequently, to find the messages pertinent to them, one simple way is for every recipient to scan the entire ledger. However, the imposed communication and computation costs may be too much of a burden for recipients with limited resources (e.g., wallet apps running on mobile devices). It is desirable to outsource this to a server in a privacy-preserving way.

Fuzzy Message Detection (FMD) [2, 36] is the first primitive proposed to address this issue. It adopts a decoy-based approach: the server detects and forwards a set of messages, including both the pertinent messages and random false positives. This is a weak, non-robust security guarantee [38].

Two primitives emerged after FMD enhanced the security guarantee to entirely hide the set of pertinent messages. Private Signaling (PS) [20,27] focuses on achieving this functionality by leveraging a trusted execution environment (TEE) or two communicating-but-non-colluding servers, while Oblivious Message Retrieval (OMR) [21, 22] realizes it via cryptographic assumptions on a single server. The state-of-the-art PS work [20] provides a very scalable solution; nonetheless, this line of work has a much stronger environmental assumption than OMR. Thus, in this work, we focus on OMR.

System Model. OMR works in the following model: in the systems, there are *senders* who send messages to the *recipients* without revealing who the recipients are. Each *message* contains a *payload* and a *clue* generated by the sender using the recipient's *clue key*. All the messages are placed on a *bulletin board*. When the recipients want to retrieve their messages, they send a *detection key* to an untrusted server, denoted the *detector*. The detector uses the board and the

detection key to generate a *digest* and sends it back to the recipients. The recipients decode the digest to obtain all the payloads pertinent to them.

Threat model. We consider an adversary that wishes to learn metadata about which messages are addressed to which user, and the identity of users that perform retrieval queries.

The adversary can read all public information (including all board messages and all public keys in the system), and all communication between detectors and the recipients. The adversary may control, or collude with, all the parties in the systems, except for the sender and recipient(s) of the message(s) whose privacy is to be protected. The adversary and its colluding parties may behave maliciously and send malformed messages and keys; but they are computationally bounded (i.e., cannot break the underlying computational assumptions).

Prior OMR schemes. Under these models, OMR [21] (and its extension to group setting [22]) offer a solution based on PVW encrpytion [34] and BFV homomorphic encryption [7, 13], However, there are several major practical concerns about the existing constructions:[1] (1) the detector is slow, taking ∼109 ms/msg (1-thread, and ∼51.7 ms/msg 4-thread) per processed message, i.e., approximately $17.6 per month for Bitcoin-scale application; (2) the clue key size is ∼132kB, which is awkward to senders as part of the recipient's public address/key; (3) the clue itself has a size of ∼956 bytes, which roughly doubles the total message size, e.g., for typical Zcash transactions.

In this paper, we address these practical issues by presenting two protocols that greatly outperform the prior works, and offer different tradeoffs between computation, clue sizes and key sizes.

1.1 Our Contribution

New constructions. We show two new constructions that provide different trade-offs. Both of the constructions are based on standard lattice hardness assumption (Ring-LWE) and are thus plausibly post-quantum secure.

- For our first construction, we tailor Ring-LWE encryption into a variant that suits our application. Combining this tailored scheme and a newly designed retrieval circuit (including a new decryption circuit and a new way to encode the digest), we obtain a construction that is both asymptotically and concretely faster in terms of the detector runtime.
- In the second construction, we show an alternative way to parametrize our Ring-LWE variant, together with a new decryption circuit. This alternative construction achieves a much smaller clue and clue key size, and a detector runtime that is still faster than prior work [21].

Implementation and evaluation. We implement our constructions in a (to-be-open-sourced) C++ library and measure the concrete improvement. Salient observations include:

- With the first construction, the detector runtime is about 15x faster, and the clue key size is about 60x smaller. The detector runtime is only ∼7.3 thread-ms/msg scanned (∼3.7 ms/msg 2-thread), thus only costs ∼$0.12 per million message scanned ($1.12/month for Bitcoin-scale applications).
- With the second construction, the detector runtime is about 2.7x faster than prior work (∼40.0ms/msg 1-thread, ∼20.2ms/msg 2-thread), while the clue key size is about 235x smaller. Furthermore, the clue size is about 1.6x smaller. These advantages allow the applications to have much less clue distribution and message size burdens.

We also discuss implications of these improvements on integration with a blockchain-based privacy-preserving cryptocurrency (exemplified by Zcash).

1.2 Related Works

Oblivious Message Retrieval. [21] proposes a message retrieval primitive with full recipient privacy and [22] extends it to the group setting. We recap how the construction of [21] works in § 4, and compare our schemes with it asymptotically in§ 2 and concretely in § 7.

Fuzzy Message Detection. FMD [2, 36] mainly focuses on decoy based security. While the construction is highly efficient, the security notion is relatively weak as analyzed in [38].

Private Signaling. Like OMR, PS [20,27] provides full security. However, prior works on private signaling have constructions using a Trusted Execution Environment (TEE). TEE is a strong environment assumption since a lot of work shows that the existing TEEs have side-channels that can leak secrets easily [39]. [27] also provides a solution assuming two communicating but non-colluding servers, which is also a very strong environment assumption. Moreover, this construction is concretely slower than [21] and thus our constructions.

PIR. Other related problems are *(batch) Private Information Retrieval (PIR)* [9] and its variant *(batch) Keyword PIR* [8]. Our setting differs in that recipients do not know the indices or labels of messages pertinent to them; rather, the clues are randomized and require nontrivial computation (rather than simple comparison) to detect.

Private Stream Search. In Private Stream Search (PSS) [4, 12, 14, 32], a client can prvately search a keyword over a database of documents and retrieve the ones with that keyword. As for Keyword PIR, this does not directly yield OMR.

See [21, 22] for further discussion.

2 Technical Overview

We follow the OMR framework introduced in [21] to build our improved constructions, and the requisite background is systematically recalled in § 3 and 4; then the improvements

[1] Benchmarked using the parameters of [22, Sec 9] and Google Cloud prices (instance type `e2-standard-4`), amortized.

	ClueToPackedPV		PVUnpack		ExpandedPVToDigest		Clue Size	Clue Key Size	Detection Key Size	Digest Size
	# of hom. operations	Depth	# of hom. operations	Depth	# of hom. operations	Depth				
OMRp2 [21,22]	$O(N\ell t)$	$O(\log(\ell t))$	$O(N\log D)$ or $O(N)$	1 or $\log(D)$	$\tilde{O}(P\cdot N)$	1	$O(n\log(t))$	$\omega(n\ell\log(n)\log(t))$	poly in homomorphic circuit depth	$\tilde{O}(P(\bar{k}+N\varepsilon_{\mathrm{p}}))$
PerfOMR1§ 5	$O(N\ell(\log(t)+h))$	$O(\log(\ell t h))$	$O(N/\mathsf{v})$	1			$O(n\log(t))$	$O(n\log(t))$		$\tilde{O}(P(\bar{k}+N\varepsilon_{\mathrm{p}})\mathsf{v})$
PerfOMR2§ 6	$O(N\ell(q\cdot h))$	$O(\log(\ell q h))$					$O(n\log(q))$	$O(n\log(q))$		

Table 1: Asymptotic comparison with prior construction. N is the total number of messages. $\bar{k}$ is the upper bound of the number of pertinent messages provided by the recipient. ε_{p} is the false positive rate. n,ℓ,q,h are all PVW encryption or sRLWE scheme parameters (see § 3.1 and § 5.1). t is the BFV plaintext space. Practically $t \geq qh \gg \log(t)+h$, and D is the BFV ring dimension. P is the size of a payload (which can also be viewed as a constant). v is a tune-able parameter in our construction, which essentially means "gluing" v messages together and treating them as a single message in the later detection phases (see §§ 4.2.2 and 4.2.3).

are presented in detail in § 5 and § 6. For those readers already familiar with the approach of [21] and the encryption schemes it employs (PVW [34] and BFV [7, 13]), the following succinctly summarizes our approach to improvements.

Setup with tailored RLWE Encryption. Whereas [21] had clues which are PVW encryptions (based on LWE hardness), we instead use an encryption scheme based on Ring-LWE hardness. Our encryption scheme, sRLWE, is a variant of RLWE [26] but with sparse key, smaller decryption range, and smaller plaintext space. The encryption public key sRLWE.pk (included in the clue key) is much smaller than with PVW.

The sender generates $\mathsf{sRLWE.Enc}(\mathsf{sRLWE.pk}, 0) \in \mathbb{Z}_t^{n+1}$ as the clue (for some security parameter n, ciphertext modulus q)[2]. To perform a retrieval, the recipient uses the homomorphic encryption scheme BFV to compute $\mathsf{ct_{sk}} \leftarrow \mathsf{BFV.Enc}(\mathsf{BFV.pk}, \mathsf{sRLWE.sk})$ and send $(\mathsf{BFV.pk}, \mathsf{ct_{sk}})$ as the detection key to the detector.

A more efficient homomorphic decryption circuit. Given a detection key, the first step performed by the detector is to homomorphically decrypt each sRLWE ciphertext over $\mathbb{Z}_t$. Prior work relies on a degree-$(t-1)$ polynomial as in [21], which requires $t-1$ homomorphic operations. By exploiting the fact that sRLWE relies on sparse secret-key RLWE (using secrets with fixed hamming weight h), which implies that the sRLWE ciphertexts have noise $O(h)$, we design a more efficient decryption circuit that only takes $O(h+\log(t))$ operations. Since this circuit is evaluated using BFV, D (BFV ring dimension) clues are homomorphically decrypted simultaneously. For N clues, this process results in $d = \lceil N/D \rceil$ BFV ciphertexts, each of which encrypts a binary vector of size D (in its D slots) representing whether D corresponding messages are pertinent. We call this step ClueToPackedPV.

A new way to expand the BFV ciphertexts. The next step for the detector is to homomorphically expand these ciphertexts. Instead of one ciphertext encrypting D bits, the detector needs D ciphertexts each encrypting a single bit repeated D times (for more efficient digest encoding). To accomplish this goal, we first homomorphically decode the ciphertext via the SlotToCoeff procedure in [24]. Then, we perform OExpand introduced in [1] on the decoded ciphertexts to obtain the targeted result. This new expansion way requires only $O(D)$ operations for each ciphertext, compared to $O(D\log(D))$ operations in prior work [21]. We call this step PVUnpack.

Bundling v messages. With this new way of expansion, it still takes $O(N)$ homomorphic operations for N messages. A natural improvement is to bundle v messages to a single message. This can be done by adding up v ciphertexts obtained from ClueToPackedPV before expanding the ciphertexts.

A new encoding scheme. Despite the improved efficiency of the bundling technique, it introduces extra complexity. The major issue is that the encoding scheme in [21] does not work anymore, given that the ciphertexts output from PVUnpack now encrypts non-binary values (since we add v binary values together). To resolve this, we design a new encoding scheme for index encoding: we first expand each bit of the indices into $\log(\mathsf{v}+1)$ bits; then we use these expanded indices to encode and allow the recipient to decode all the pertinent indices. We call this last encoding step ExpandedPVToDigest

Putting all these three steps ClueToPackedPV,PVUnpack, ExpandedPVToDigest together, we obtain our first construction PerfOMR1, which is both asymptotically and concretely faster than the prior work OMRp2 in [21].

An alternative way to use sRLWE. Another way to use sRLWE is that instead of having its ciphertext modulus be t (the same as the BFV plaintext modulus, which relatively large for practicality), we can set sRLWE modulus to $q \ll t$. As our sRLWE relies on sparse keys (keys with hamming weight h), we set $qh < t/2$. This guarantees that decrypting the sRLWE ciphertext over $\mathbb{Z}_t$ is the same as over $\mathbb{Z}$ (no wrap-arounds). Therefore, we can instead design a polynomial with $O(qh)$ degree to perform the homomorphic decryption. While this makes the runtime worse, the clue key and clue of size $O(n\log(q))$ can be greatly reduced as q now is smaller.

Asymptotics. In Table 1, we compare the asymptotic behavior of our constructions, in terms of the cost metrics, with the prior construction in [21, 22]. As mentioned in above, our work mainly focuses on the improvement of the detector construction, which is composed of three main steps:

[2] In the actual construction, we encrypt 0^{ℓ} for some $\ell \geq 1$, to reduce false positive rate. We set $\ell = 1$ here for simplicity.

ClueToPackedPV, PVUnpack, ExpandedPVToDigest.

For our first construction, PerfOMR1, the detector runtime is strictly faster than the prior works by having much fewer homomorphic operations: in the step ClueToPackedPV, h is the hamming weight of the secret key which is normally viewed as $O(1)$ [15], and we thus have $\log(t)+h = o(t)$; in the step PVUnpack, we have $\mathsf{v} \geq 1$, thus reducing the number of homomorphic operations by a factor of v. Besides, the clue key size is smaller by reducing $\omega(n\log(n))$ to $O(n)$.

For our second construction PerfOMR2, we set $q \cdot h \leq t$. Therefore, the runtime is comparable with the prior work with slightly fewer homomorphic operations in the step ClueToPackedPV. The gain is that the clue size and the clue key size are both smaller since they now depend on $q < t$.

Note that the digest size of both of our constructions is parametrized by v. Concretely, the digest size is exactly the same as the prior work when $\mathsf{v} = 1$. Since v only affects the runtime of PVUnpack step, when PVUnpack is the runtime bottleneck, we set $\mathsf{v} > 1$ (e.g., for PerfOMR1); otherwise, we set $\mathsf{v} = 1$ (e.g., for PerfOMR2).

See § 7 for evaluation of concrete performance.

3 Preliminaries

Notation. Let $[n]$ denote the set $\{1,\ldots,n\}$. For a vector $\mathbf{v}$, $\mathbf{v}[i]$ indicates the i-th element of this vector. For a matrix A, $A[i][j]$ indicates the cell at the i-th row and j-th column. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ denote the $2N$-th cyclotomic ring where N is a power-of-two, and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$ for some $Q \in \mathbb{Z}$. For matrices $A, B \in \mathbb{Z}_t^{n\times m}$, let $\circ$ denote the Hadamard product $C \leftarrow A \circ B$ satisfying $C[i][j] = A[i][j]\cdot B[i][j], \forall i \in [n], j \in [m]$. Drawing x uniformly at random from a set S is denoted $x \xleftarrow{\$} S$.

Definition 3.1 (Decisional ring learning with error problem)**.** [26, 35] Let $N, Q, \mathcal{D}, \chi$ be parameters dependent on λ and N being a power of two. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$. The ring learning with error (RLWE) problem $\mathsf{RLWE}_{N,Q,\mathcal{D},\chi}$ is the following: distinguish $(a, a\cdot s+e)$ and (a,b) (with noticeable advantage), where $a \xleftarrow{\$} \mathcal{R}_Q, s \leftarrow \mathcal{D}, e \leftarrow \chi$ and $b \xleftarrow{\$} \mathcal{R}_Q$.

3.1 PVW Encryption

We adapt the PVW encryption from [34] and modify it according to [21].

- $\mathsf{pp} = (n,\ell,w,q,\sigma,r) \leftarrow \mathsf{PVW.GenParam}(1^\lambda,\ell,q,\sigma,\varepsilon_n)$: Choose a secret key dimension n, and $w = \omega(n\log(q))$ by ciphertext modulus q, plaintext size ℓ, and standard deviation σ for Gaussian distribution, as in [34]. Choose the noise bound r s.t. $\Pr[\mathsf{PVW.Dec}(\mathsf{sk}, \mathsf{PVW.Enc}(\mathsf{pk},\vec{m})) = \vec{m}] \geq 1-\varepsilon_n-\mathsf{negl}(\lambda)$.
- $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{PVW.KeyGen}(\mathsf{pp})$: Draw a secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q^{n\times\ell}$. Sample $A \xleftarrow{\$} \mathbb{Z}_q^{n\times w}$ and a noise matrix $X \in \mathbb{Z}_q^{\ell\times w}$ from the Gaussian distribution χ_σ, and compute $\mathsf{pk} = (A, P = \mathsf{sk}^T A + X)$.
- $\mathsf{ct} = (\vec{a},\vec{b}) \leftarrow \mathsf{PVW.Enc}(\mathsf{pp},\mathsf{pk}=(A,P),\vec{m})$: To encrypt a vector $\vec{m} \in \mathbb{Z}_2^\ell$, define the vector $\vec{t} = \frac{q}{2}\cdot\vec{m} \in \mathbb{Z}_q^\ell$, and draw $\vec{e} \xleftarrow{\$} \{0,1\}^w \in \mathbb{Z}_2^w$. The ciphertext is the pair $(\vec{a},\vec{b}) = (A\vec{e}, P\vec{e}+\vec{t}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^\ell$.
- $\vec{m} \leftarrow \mathsf{PVW.Dec}(\mathsf{pp},\mathsf{sk},\mathsf{ct}=(\vec{a},\vec{b}))$: $\vec{d} = \vec{b} - \mathsf{sk}^T\vec{a} \in \mathbb{Z}_q^\ell$, let $\vec{m} \in \mathbb{Z}_2^\ell$, and $\vec{m}[i] = 1$ iff $\vec{d}[i] + r/2 > r$ for all $i \in [\ell]$.

The scheme satisfies CPA security and tailored correctness: correct with probability $1-\varepsilon_n$ for some $0 < \varepsilon_n < 1$.

PVW also has two additional properties: key privacy (i.e., ciphertexts encrypted under different public keys are computationally indistinguishable); and zero-plaintext wrong-key decryption (i.e., given the wrong key, a PVW ciphertext is decrypted into a zero plaintext with small probability.) Since we later define these two properties for the PKE scheme we use in Thm 5.1, we skip the formal definition here for brevity.

3.2 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE), introduced by Rivest et al. [37] and first constructed by Gentry [16], enables evaluation of a circuit on encrypted data.

BFV FHE scheme. We use the Brakerski/Fan-Vercauteran (BFV) homomorphic encryption scheme [7, 13].

BFV scheme consists of the following PPT algorithms: $\mathsf{GenParam}(1^\lambda), \mathsf{KeyGen}(\mathsf{pp_{BFV}}), \mathsf{Enc}(\mathsf{pp_{BFV}},\mathsf{pk},m)$, $\mathsf{Dec}(\mathsf{pp_{BFV}},\mathsf{sk},c)$ as normal PKE schemes. BFV is unconditionally correct and sound. Under the *RLWE* hardness assumption, it also fulfills CPA security.

Given a polynomial from the cyclotomic ring $R_t = \mathbb{Z}_t[X]/(X^D+1)$ (where D is a power-of-two, $t \equiv 1 \mod 2D$), the BFV scheme encrypts it into a ciphertext consisting of two polynomials, each of which in $R_Q = \mathbb{Z}_Q[X]/(X^D+1)$ for some $Q > t$. Here, t, Q, and D are called the plaintext modulus, the ciphertext modulus, and the ring dimension, respectively.

Plaintext encoding. In practice, instead of having a polynomial in $\mathcal{R}_t = \mathbb{Z}_t[X]/(X^D+1)$ directly as input, applications usually hold a vector of messages $\vec{m} = (m_1,\ldots,m_D) \in \mathbb{Z}_t^D$. Thus, to encrypt such input messages, BFV first encodes the messages into another polynomial $y(X) = \sum_{i\in[D]} y_i X^{i-1}$ such that $m_j = y(\zeta_j)$, $\zeta_j := \zeta^{3^j} \mod t$, and ζ is the $2N$-th primitive root of unity of t. Such encoding can be done using Inverse Number Theoretic Transform (INTT), which is a linear transformation represented as matrix multiplication. We say that a BFV ciphertext has D *slots*, each of which is a $\mathbb{Z}_t$ element.

For simplicity, we assume BFV.Enc takes a vector of form $\mathbb{Z}_t^D$ as an input, and BFV.Dec outputs a vector of form $\mathbb{Z}_t^D$, and will handle encode and decode implicitly. We use BFV.PartialDec to represent decryption without decoding. In other words, for a ciphertext ct, the output of $\mathsf{BFV.PartialDec}(\mathsf{sk},\mathsf{ct}) \in \mathcal{R}_t$ is the encoding of

$\mathsf{BFV.Dec}(\mathsf{sk},\mathsf{ct}) \in \mathbb{Z}_t^D$.

Operations. BFV supports the following operations.

- (Additions) For any two BFV ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$, and $\mathsf{ct} \leftarrow \mathsf{ct}_1 + \mathsf{ct}_2$, it holds that $\mathsf{BFV.Dec}(\mathsf{ct}) = \mathsf{BFV.Dec}(\mathsf{ct}_1) + \mathsf{BFV.Dec}(\mathsf{ct}_2)$ (element-wise).
- (Multiplication) For any two BFV ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$, and $\mathsf{ct} \leftarrow \mathsf{ct}_1 \times \mathsf{ct}_2$, it holds that $\mathsf{BFV.Dec}(\mathsf{ct}) = \mathsf{BFV.Dec}(\mathsf{ct}_1) \times \mathsf{BFV.Dec}(\mathsf{ct}_2)$ (element-wise).
- (Rotation) For any BFV ciphertexts ct, and $\mathsf{ct}' \leftarrow \mathsf{BFV.Rotate}(\mathsf{ct}, k)$ for some $k \in [D]$, let $\mathsf{BFV.Dec}(\mathsf{sk},\mathsf{ct})[i] = \mathsf{BFV.Dec}(\mathsf{sk},\mathsf{ct}')[i+k \mod D]$.
- (Substitution) For any BFV ciphertexts ct, and $\mathsf{ct}' \leftarrow \mathsf{BFV.Substitute}(\mathsf{ct}, k)$ for some odd number k, let $y(X) = \mathsf{BFV.PartialDec}(\mathsf{sk},\mathsf{ct})$ and $y'(X) = \mathsf{BFV.PartialDec}(\mathsf{sk},\mathsf{ct}')$, it holds that $y'(X) = y(X^k) \in \mathcal{R}_t$.

3.3 Oblivious Message Retrieval (Definition)

We adapt the definition of OMR from [21] by introducing a new parameter ν to relax the soundness and compactness as follows. At a high level, the scheme is allowed to include impertinent payloads, as long as the final output is still bounded by ν. For example, the scheme can bundle ν payloads together. If a bundle contains a pertinent message, the scheme can return all ν payloads in the bundle to the recipient. We discuss why this relaxation is reasonable in Remark 3.4.

Definition 3.2 (Oblivious Message Retrieval)**.** An Oblivious Message Retrieval scheme has the following PPT algorithms:

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

To define soundness and completeness, we first define the notion of board generation:

Definition 3.3 (Board Generation)**.** Given pp, and the size of bulletin board N: arbitrarily choose the number of recipients $1 \le p \le N$, and a partition of set $[N]$ into p subsets $S_1, \ldots, S_p$ representing the indices of messages addressed to each party. Also arbitrarily choose unique payloads $(x_1, \ldots, x_N)$. For each recipient $i \in [p]$: generate keys $(\mathsf{sk}_i, \mathsf{pk}_i = (\mathsf{pk}_{\mathsf{clue}i}, \mathsf{pk}_{\mathsf{detect}i})) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$, and for each $j \in S_i$, generate $c_j \leftarrow \mathsf{GenClue}(\mathsf{pk}_{\mathsf{clue}i}, x_j)$. Then, output the board $\mathsf{BB} = \{(x_1,c_1), \ldots, (x_N,c_N)\}$, the set S_1, and $(\mathsf{sk}_1, \mathsf{pk}_1 = (\mathsf{pk}_{\mathsf{clue}1}, \mathsf{pk}_{\mathsf{detect}1}))$.[3]

The scheme must satisfy the following properties:

- (Completeness) Let $\mathsf{pp} \leftarrow \mathsf{GenParam}(1^\lambda, \varepsilon_\mathrm{p}, \varepsilon_\mathrm{n}, \nu)$. Set any $N = \mathsf{poly}(\lambda)$, and $0 < \bar{k} \le N$. Let a board BB, a set S of pertinent messages, and a key pair $(\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}}))$ be generated as in Def 3.3 for any choice of p, partition and payloads therein. Let $M \leftarrow \mathsf{Retrieve}(\mathsf{BB}, \mathsf{pk}_{\mathsf{detect}}, \bar{k})$ and $\mathsf{PL} \leftarrow \mathsf{Decode}(M, \mathsf{sk})$. Let $k = |S|$ (the number of pertinent messages in S). Then either $k > \bar{k}$ and $\mathsf{PL} = \mathsf{overflow}$, or: $\Pr[x_j \in \mathsf{PL} \mid j \in S] \ge (1 - \varepsilon_\mathrm{n} - \mathsf{negl}(\lambda)) \quad \forall j \in [N]$.
- (ν-Soundness) For the same quantifiers as in Completeness: $|\mathsf{PL}| = \tilde{O}(\nu \cdot P \cdot (\bar{k} + \varepsilon_\mathrm{p} N))$
- (Computational privacy) For any PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$: let $\mathsf{pp} \leftarrow \mathsf{GenParam}(\varepsilon_\mathrm{p}, \varepsilon_\mathrm{n})$, $(\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}})) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$ and $(\mathsf{sk}', \mathsf{pk}' = (\mathsf{pk}'_{\mathsf{clue}}, \mathsf{pk}'_{\mathsf{detect}})) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$. Let the adversary choose a payload x and remember its state: $(x, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}, \mathsf{pk}, \mathsf{pk}')$. Let $c \leftarrow \mathsf{GenClue}(\mathsf{pk}_{\mathsf{clue}}, x)$ and $c' \leftarrow \mathsf{GenClue}(\mathsf{pk}'_{\mathsf{clue}}, x)$. Then: $|\Pr[\mathcal{A}_2(\mathsf{st}, c) = 1] - \Pr[\mathcal{A}_2(\mathsf{st}, c') = 1]| \le \mathsf{negl}(\lambda)$.

An OMR scheme is ν-*compact* if it satisfies the following:

- (ν-Compactness) An OMR scheme is ν-compact if for $\mathsf{pp} \leftarrow \mathsf{GenParam}(1^\lambda, \varepsilon_\mathrm{p}, \varepsilon_\mathrm{n})$, $(\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}})) \leftarrow \mathsf{OMR.KeyGen}(\mathsf{pp})$, for any board $\mathsf{BB} = \{(x_1,c_1), \ldots, (x_N,c_N)\}$, letting $M \leftarrow \mathsf{Retrieve}(\mathsf{BB}, \mathsf{pk}_{\mathsf{detect}}, \bar{k})$, it always holds that: $|M| = \mathsf{poly}(\lambda, \log N) \cdot \log \varepsilon_\mathrm{p}^{-1} \cdot \tilde{O}(\bar{k} + \varepsilon_\mathrm{p} N) \cdot \nu$.

In the compactness definition, $\tilde{O}(\bar{k} + \varepsilon_\mathrm{p} N)$ (where $\tilde{O}(x) = x \cdot \mathsf{polylog}(x)$) accounts for the number of messages detected as pertinent, including false positives; and the remaining factors account for the cost of representing each such message, taking the payload size as constant.

Remark 3.4. Note that we have relaxed the "soundness" and "compactness" properties in [21, Def 4.3] special case to "ν-soundness" and "ν-compactness", allowing the digest and the decoded payloads to include more than just the pertinent messages, by a factor of ν. The scheme is thus allowed to bundle $O(\nu)$ messages as a single one and process them together (where each bundle may include pertinent messages and impertinent ones simultaneously).

In most applications of OMR, the recipients are able to find the single intended data payload from a bundle (e.g., the payloads are encrypted and using the wrong key to decrypt is detectable as a decryption failure). Therefore, in many cases, it is not an issue. See the full [23] version showing a general way to guarantee full soundness with a small cost.

[3] That is, S_1 is the indices of messages pertinent to the recipient whose keys are $\mathsf{sk}_1, \mathsf{pk}_1$, which wlog is the first recipient.

4 Revisiting the OMRp2 Construction

We first revisit and summarize the construction of OMR, OMRp2 in [21, Alg 8], which is the basis for improvements in later sections. Here, we abstract out each step of OMRp2 and provides modular analysis to each step to make the entire framework easier to understand.

4.1 Setup

OMRp2 mainly relies on the PVW encryption (see § 3.1) for clues and BFV FHE scheme (see § 3.2) for retrieval.

GenParam**.** Public parameter generation is straightforward. It outputs a public parameter pp including the PVW parameters $\mathsf{pp}_{\mathsf{PVW}} = (n, w, q, \ell, \mathcal{D}, \sigma)$, the BFV parameters $\mathsf{pp}_{\mathsf{BFV}} = (D, t, \dots)$,[4], false positive rate ε_{p} and false negative rate ε_{n}.

KeyGen**.** The recipient first generates a PVW key pair $(\mathsf{sk}_{\mathsf{pvw}}, \mathsf{pk}_{\mathsf{pvw}})$ and a BFV key pair $(\mathsf{sk}_{\mathsf{BFV}}, \mathsf{pk}_{\mathsf{BFV}})$. $\mathsf{pk}_{\mathsf{pvw}}$ will be the clue key. The recipient then generates $\mathsf{ct}_{\mathsf{sk}} \leftarrow \mathsf{BFV.Enc}(\mathsf{pp}_{\mathsf{BFV}}, \mathsf{pk}_{\mathsf{BFV}}, \mathsf{sk}_{\mathsf{pvw}})$, the encrypted $\mathsf{sk}_{\mathsf{pvw}}$ under $\mathsf{pk}_{\mathsf{BFV}}$. The tuple $(\mathsf{ct}_{\mathsf{sk}}, \mathsf{pk}_{\mathsf{BFV}})$ serves as the detection key.

GenClue**.** After fetching the recipient's $\mathsf{pk}_{\mathsf{clue}}$, the sender computes $c \leftarrow \mathsf{PVW.Enc}(\mathsf{pp}_{\mathsf{PVW}}, 1^{\ell})$. If a clue is decrypted to 1^{ℓ}, it indicates that the message is pertinent. Otherwise, there is at least one zero among the ℓ bits, and the message is impertinent. Based on the wrong-key decryption property, if a message is impertinent, the decrypted message will not be 1^{ℓ} with high probability.

4.2 Retrieval

To retrieve the pertinent messages for the recipients, the detector invokes OMRp2.Retrieve, which is composed of three main steps. We first define those steps and describe how OMRp2 in [21] realizes them. Looking ahead, our new construction rewrites these three with better efficiency, both asymptotically and concretely.

4.2.1 Step 1: From Bulletin Board to Pertinency Vector

The first step takes the detection key and all the clues on the bulletin board, and outputs a vector of BFV ciphertexts, each slot of which indicates whether a single message is pertinent (we call *pertinency vector*, PV). We visualize it in Fig. 1. The interface is defined as follows:

- $(\mathsf{ct}_1, \dots, \mathsf{ct}_d) \leftarrow \mathsf{ClueToPackedPV}(\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}, \mathsf{BB})$: takes public parameter pp, a detection key $\mathsf{pk}_{\mathsf{detect}}$, and a bulletin board BB of size N; outputs a vector of BFV ciphertexts $(\mathsf{ct}_1, \dots, \mathsf{ct}_d)$ where $d = \lceil N/D \rceil$.

[4]Technically, we should also set Q, the ciphertext modulus, to guarantee that there is enough noise budget to evaluate the entire circuit. However, it is not used in constructions explicitly, we leave it implicit for better readability.

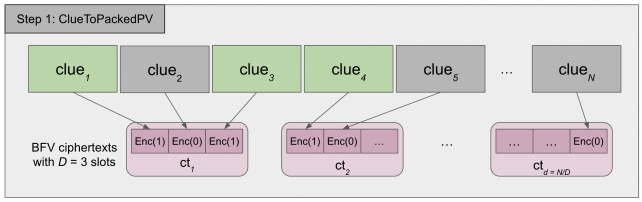


Figure 1: Visualization of Step 1 ClueToPackedPV. The green clues are for the pertinent messages and the gray ones are for the impertinent messages. Each BFV ciphertext in pale pink has $D = 3$ slots and each slot in dark pink encrypts the pertinency of a single message. For N messages, the output has $d = N/D = N/3$ ciphertexts.

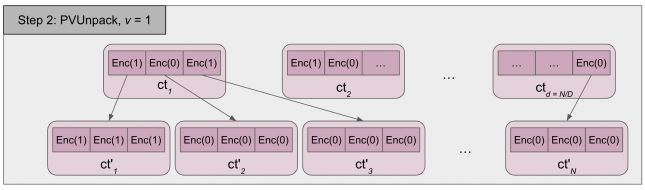
Recall that each BFV ciphertext contains D slots (i.e., encrypting a vector of $\mathbb{Z}_t^D$ for t being the plaintext modulus), where D is the ring dimension. If the i-th message is pertinent, the i-th slots should be 1; and 0 otherwise. Thus, there are in total of $\lceil N/D \rceil$ ciphertexts with $\geq N$ slots to encrypt the pertinency of each message. Correctness is defined as follows:

Definition 4.1 (Correctness of ClueToPackedPV)**.** Let $\mathsf{pp} \leftarrow \mathsf{GenParam}(1^{\lambda}, \varepsilon_{\mathrm{p}}, \varepsilon_{\mathrm{n}})$. For any $N = \mathsf{poly}(\lambda)$, and $0 < \bar{k} \leq N$, let a board BB, a set S of pertinent messages, and a key pair $(\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}}))$ be generated as in Def 3.3 for any choice of p, partition and payloads therein, let $(\mathsf{ct}_1, \dots, \mathsf{ct}_d) \leftarrow \mathsf{ClueToPackedPV}(\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}, \mathsf{BB})$, it holds that: (1) $\Pr[\mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct}_j)[i] = 1 \mid j \cdot D + i \in S] \geq (1 - \varepsilon_{\mathrm{n}} - \mathsf{negl}(\lambda))$ for all $i \in [D], j \in [d]$.
and: (2) $\Pr[\mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct}_j)[i] = 1 \mid j \cdot D + i \notin S] \leq (\varepsilon_{\mathrm{p}} + \mathsf{negl}(\lambda))$ for all $i \in [D], j \in [d]$.

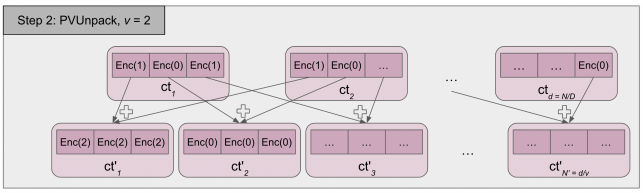
ClueToPackedPV **Implementation.** In the construction of OMRp2, the detector uses $\mathsf{ct}_{\mathsf{sk}}$ to homomorphically decrypt each clue (where $\mathsf{ct}_{\mathsf{sk}}$ is the encryption of the PVW secret key and the clue is a PVW ciphertext). If a message is indeed pertinent, the homomorphic decryption yields 1^{ℓ} except with $\varepsilon_{\mathrm{n}} + \mathsf{negl}(\lambda)$ probability. Otherwise, the result would be 1^{ℓ} with probability $\leq \varepsilon_{\mathrm{p}} + \mathsf{negl}(\lambda)$. Lastly, the detector multiplies all the ℓ bits, and gets 1 if and only if the message is pertinent.

This homomorphic decryption circuit is evaluated under BFV, and D messages are processed simultaneously. As mentioned before, the ciphertext ct after the homomorphic decryption has D slots, where the i-th slot encrypts 1 if and only if the i-th message is pertinent (except with some small bounded probability), for $i \in [D]$. This decryption process is repeated $d = \lceil N/D \rceil$ times to obtain ciphertexts for all the N messages.

Essentially, let $\mathsf{sk}[i] \in \mathbb{Z}_q^n$ denote the i-th column of $\mathsf{sk} \in \mathbb{Z}_q^{n \times \ell}$, the PVW decryption circuit checks whether $|b[i] - \langle \vec{a}, \mathsf{sk}[i] \rangle| \leq r$ for $i \in [\ell]$, for each clue of form $(\vec{a}, \vec{b}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^{\ell}$. The evaluation of $b[i] - \langle \vec{a}, \mathsf{sk}[i] \rangle$ is easy. The hard part is the range check, as BFV operates over a finite field and only



(a) Step 2 with $\mathsf{v} = 1$ (i.e., for the original OMR construction in § 4.2.2)



(b) Step 2 with $\mathsf{v} = 2$ (i.e., for our new construction in § 5.2.2)

Figure 2: Visualization of step 2 PVUnpack. When $\mathsf{v} = 1$, each slot is expanded into a single BFV ciphertext, resulting in N ciphertexts. When $\mathsf{v} = 2$, two slots are added up and expanded into a single BFV ciphertext, thus N/v ciphertexts

supports additions and multiplications. Fortunately, one important observation in [21] is that any function over $\mathbb{Z}_t$ can be represented by a polynomial with degree-$(t-1)$. Therefore, the detector interpolates a degree-$(t-1)$ function to check the range and completes the step ClueToPackedPV.

4.2.2 Step 2: Unpack the Pertinency Vector

After obtaining the pertinency vector, to prepare for the third step, the framework in [21] unpacks those N slots (of the $\lceil N/D \rceil$ ciphertexts from last step) into N ciphertexts. If the i-th slot is 1, the i-th ciphertext after unpacking encrypts 1 in all of its D slots, and 0 in all D slots otherwise.

We further generalize this step to take a parameter called *bundle size* v to output N/v ciphertexts instead of N ciphertexts. For $\mathsf{v} = 1$, this procedure simply unpacks N slots into N separate ciphertexts. Looking ahead, in section § 5.2.2, we bundle $\mathsf{v} > 1$ messages into a single one for better efficiency: the i-th ciphertext after unpacking encrypts the number of slots encrypting 1's among all the v slots corresponding to the bundled messages. For simplicity, we require v to divide d. We visualize step 2 and the difference between $v = 1$ and $v > 1$ in Fig. 2. The interface is as follows:

- $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_{N'}) \leftarrow \mathsf{PVUnpack}(\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_d), \mathsf{v})$: takes public parameter pp, a detect key $\mathsf{pk}_{\mathsf{detect}}$, a vector of BFV ciphertexts of length d, the bundle size v; outputs a vector of BFV ciphertexts of size $N' = d \cdot D/\mathsf{v}$ for D being the underlying BFV ring dimension.

Definition 4.2 (Correctness of PVUnpack)**.** Let pp, $\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}})$ generated as in Def 4.1, for any vector of ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_d)$, let $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_{N'}) \leftarrow \mathsf{PVUnpack}(\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_d), \mathsf{v})$, it holds that:$\forall i \in [D], j \in [0, \frac{d}{\mathsf{v}} - 1]$, $\Pr[\mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct}'_{j \cdot D + i}) = (\sum_{w=0}^{\mathsf{v}-1} \mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct}_{j \cdot \mathsf{v} + w})[i])^D] \geq 1 - \mathsf{negl}$

PVUnpack **Implementation (with** $\mathsf{v} = 1$**).** We explain detailed construction in [21] by giving a simplified example of unpacking a single ciphertext, i.e., given a BFV ciphertext ct encrypting $(b_1, \ldots, b_D)$ where b_i is the i-th encrypted bit, we eventually want D new ciphertexts where the i-th ciphertext encrypts $\vec{b_i} := (b_i, \ldots, b_i)$, which is a vector of D b_i's in all slots.

[21] first multiply ct with $(0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{Z}_t^D$ where the i-th slot is 1 and other slots are 0, obtaining a ciphertext ct' encrypting $(0, \ldots, 0, b_i, 0, \ldots, 0)$, where b_i is the i-th element encrypted in ct. To fill all the D slots with the same value b_i, [21] uses the rotate-and-add method: for $j \in [\log(D)]$, it computes $\mathsf{ct}' \leftarrow \mathsf{ct}' + \mathsf{BFV.Rotate}(\mathsf{ct}', 2^{j-1})$. The final ciphertext ct' thus encrypts $\vec{b_i}$ as desired. This process is simply repeated D times for each input ciphertext.[5] This step requires D multiplications and $D\log(D)$ rotations in total.

4.2.3 Step 3: Use PV to Construct the Digest

Lastly, with all these N' ciphertexts above encrypting either non-zero (if pertinent) or zero (if impertinent), the detector forms a compact digest using BFV: if the i-th ciphertext does not encrypt zero, message i should be included in the digest. In OMRp2, $N' = N$, and each ciphertext either encrypts 1 or 0. However, our optimization requires those ciphertexts to encrypt $\mathsf{v} > 1$ and $N' = N/\mathsf{v}$. We visualize step 3 in Fig. 3. The interface is as follows:

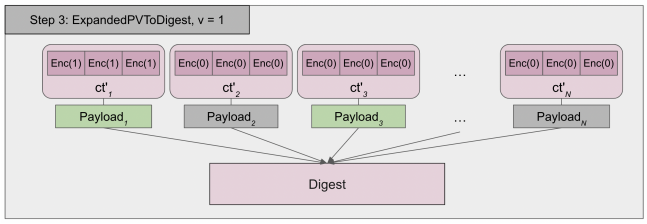
- $M \leftarrow \mathsf{ExpandedPVToDigest}(\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_{N'}, \bar{k}), \mathsf{BB})$: takes public parameter pp, a detector key $\mathsf{pk}_{\mathsf{detect}}$, a vector of BFV ciphertexts of length N', a bulletin board BB of size $N \geq N'$, N divides N', and an upper bound $\bar{k}$ on the number of pertinent messages addressed to that recipient; outputs a digest M.

Definition 4.3 (Correctness of ExpandedPVToDigest)**.** There exists a PPT algorithm DecodeDigest taking a digest M and a secret key sk and outputing payloads PL such that: for the same quantifiers as in Def 4.1, for any $N' \leq N$, and any vector of ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{N'})$ encrypted under $\mathsf{pk}_{\mathsf{detect}} = \mathsf{pk}_{\mathsf{BFV}}$, let $M \leftarrow \mathsf{ExpandedPVToDigest}(\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_{N'}), \bar{k}, \mathsf{BB})$ and $\mathsf{PL} \leftarrow \mathsf{DecodeDigest}(M, \mathsf{sk})$; let $k = |S|$ (the number of pertinent messages in S), it holds that either $k > \bar{k}$ and $\mathsf{PL} = \mathsf{overflow}$, or: $\Pr[x_j \in \mathsf{PL} \mid \mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct}_i) \neq 0^D] \geq (1 - \mathsf{negl}(\lambda))$, for all $j \in [N], i \leftarrow j \bmod N'$

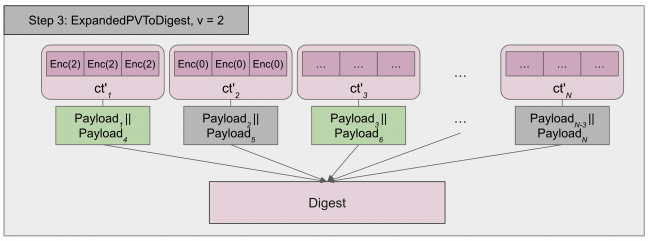
ExpandedPVToDigest **Implementation (with** $N' = N$**).** To realize ExpandedPVToDigest, [21] first encodes all the per-

[4] Here $(\cdot)^D$ means a vector of D of $\cdot$ elements.

[5] In [22], the authors provide an optimization to this step at the cost of a deeper circuit. We omit the details here for simplicity.



(a) Step 3 with $\mathsf{v} = 1$ (i.e., for the original OMR construction in § 4.2.3)



(b) Step 3 with $\mathsf{v} = 2$ (i.e., for our new construction in § 5.2.3)

Figure 3: Visualization of step 3 ExpandedPVToDigest. When $\mathsf{v} = 1$, each ciphertext has one corresponding payload to be included in the digest. When $\mathsf{v} = 2$, since two messages are viewed as a single one, their payloads are concatenated.

tinent indices into the digest, and then the corresponding pertinent payloads.

We refer to the first part as "index encoding": OMRp2 first initialize $m > \bar{k}$ buckets, where $\bar{k}$ is the upper bound of the number of pertinent messages. Then, it randomly assigns all the N messages into m buckets, and let Y_i, represent the set of messages (represented by indices) assigned to bucket $i \in [m]$.

For each bucket $i \in [m]$, compute $\mathsf{Acc}_i \leftarrow \sum_{j \in Y_i} (\mathsf{ct}_j \cdot j)$. If there is no pertinent message assigned to bucket i, Acc_i encrypts 0. If there is only one pertinent message j assigned to bucket i, Acc_i encrypts j, and the recipient can easily decrypt j to be the index of that pertinent message. However, if more than one pertinent message gets assigned in bucket i, there is a collision. To inform recipients of such collisions, OMRp2 computes $\mathsf{ctr}_i \leftarrow \sum_{j \in Y_i} \mathsf{ct}_j$ for bucket i, which is the number of pertinent messages assigned to bucket i. [6] The process is repeated $C \geq 1$ times to allow the recipient to obtain all the pertinent indices except with negligible probability, even with non-negligible probability of collision.

After obtaining all the pertinent indices, obtaining the pertinent payloads is easier. We refer to this second part as "payload encoding". The detector first samples a uniform random matrix[7] $A \in \mathbb{Z}_t^{K \times N}$ for some $K > \bar{k}$, and computes $\mathsf{comb} \leftarrow (A \circ Z) \times (\mathsf{ct}_1, \ldots, \mathsf{ct}_N)$, where $Z = \begin{pmatrix} x_1, x_2, \ldots, x_N \\ \vdots \\ x_1, x_2, \ldots, x_N \end{pmatrix} \in \mathcal{P}^{K \times N}$, [8] for x_i being payload of message i (recall that $\circ$ is the Hadamard prodoct introduced in § 3). With the pertinent indices and A (sent to the recipient as a random seed), the recipient uses Gaussian elimination to solve for all the pertinent payloads except with negligible probability.[9]

The digest thus includes $B_j = ((\mathsf{Acc}_{i,j})_{i \in [m]}, (\mathsf{ctr}_{i,j})), \forall j \in [C]$, together with comb, and the seed s used to generate A.

4.3 Decoding

The last step is for the recipient to run Decode. The recipient first checks all the counters $\mathsf{ctr}_{i,j}$ ($i \in [m], j \in [C]$) and filters the ones that encrypt 1. The corresponding $\mathsf{Acc}_{i,j}$ then contain all the pertinent indices. After obtaining these indices, the recipient uses the seed s to recover A and uses comb to solve for all the pertinent payloads.

5 Improved OMR Construction

This section focuses on our new construction of OMR, containing a new setup phase (a different way to generate the clue keys and the clue) and a more efficient algorithm for each of the three steps forming the detector retrieval.

5.1 Reducing the Clue Key Size using RLWE

We start with the setup algorithms and keys in § 4.1. Recall that the setup for each recipient is essentially generating a PVW key pair, whose public part serves as the clue key. One major issue with the PVW scheme used in OMRp2 is that the public key size is $w\ell \log(q) = \omega(\ell n \log^2(q))$, and concretely, hundreds of kilobytes — which is awkward to distribute (e.g., it is too large for direct inclusion in a cryptocurrency wallet address). We introduce a tailored variant of RLWE encryption of [26] to resolve this issue.

The main contributor to the large PVW public key is the parameter w, which is the number of LWE samples it contains. $w = \omega(n \log(q))$ is required by the leftover-hash lemma to guarantee that $A\vec{e}, P\vec{e}$ are indistinguishable from uniformly random vectors over $\mathbb{Z}_q$.[10] However, instead of making $A\vec{e}, P\vec{e}$ statistically indistinguishable from randomly drawn vectors, the scheme can rely on computational assumptions.

In other words, for $w = n$, although $A\vec{e}, P\vec{e}$ by themselves are not statistically close to random vectors, by adding some noise and get $A\vec{e} + \vec{x}', P\vec{e} + \vec{x}''$, where $\vec{x}', \vec{x}''$ are noise vectors, we again have the resulting public key indistinguishable from

[6] Note that if there are $t+1$ pertinent messages, there is an overflow since the calculation is done over $\mathbb{Z}_t$. However, [21] parametrizes the construction such that it overflows with negligible probability.

[7] Note that in [21, 22], a sparse matrix is used instead. However, that requires additional explanation and is not useful for us, so we ignore that part. See Remark 5.5 for more discussion.

[8] W.o.l.g., assume $\mathcal{P} := \{0,1\}^P$ can be embedded to $\mathbb{Z}_t$ as if not, simply repeat this process for $P/\log(t)$ times.

[9] As in [21], $K = \bar{k} + \delta/\log(t)$ to achieve $O(2^{-\delta})$ failure probability.

[10] Recall that $A\vec{e}, P\vec{e}$ are computed during PVW.Enc for $A, P \in \mathbb{Z}_q^{n \times w} \times \mathbb{Z}_q^{n \times \ell}$ being the public key (§ 3.1).

random vectors based on LWE assumption; and thus greatly reduces the public key size with $w = n$.

To achieve even better efficiency, instead of relying on LWE, we rely on RLWE. In more detail, the key generation algorithm samples $\alpha \xleftarrow{\$} \mathcal{R}_q$, where $\mathcal{R} := \mathbb{Z}(X)/(X^n+1)$ for some security parameter n being a power-of-two. The secret key $s \in \mathcal{R}$ is sampled from some distribution $\mathcal{D}$, and the public key is $(\alpha, \beta = \alpha s + x)$ for some noise x sampled from noise distribution χ_σ. To encrypt, the sender simply samples $e \leftarrow \mathcal{D}$ and computes $a \leftarrow \alpha e + x', b \leftarrow \beta e + x''$ where $x', x'' \leftarrow \chi_\sigma$.

To make it more suitable for our use case, since we need to encrypt just $\ell \ll n$ bits, only the first ℓ coefficients of the ring element b are needed during decryption. In addition, to guarantee correctness with probability $1 - \varepsilon_n$, the scheme simply needs to choose a range parameter r used for decryption to guarantee that the noise of the ciphertexts is $\leq r$ except with ε_n probability. With all noises sampled from χ_σ, and a distribution $\mathcal{D}$ such that the Hamming weight of s, e drawn from $\mathcal{D}$ are both bounded by h and $|s|_\infty, |e|_\infty = 1$, the aggregated noise of $(as + b)$ can be viewed as sampled from distribution $\chi_{\sqrt{2h+1}\cdot\sigma}$ (since there are in total $2h+1$ independently sampled noise being summed up). We can thus set r according to $\chi_{\sqrt{2h+1}\cdot\sigma}$ to guarantee correctness.

Defining sRLWE **encryption.** Putting all these together, we get a tailored variant of the RLWE encryption formally stated as follows:

- $\mathsf{pp} = (n, \ell, q, \sigma, r, \mathcal{D}) \leftarrow \mathsf{sRLWE.GenParam}(1^\lambda, \ell, q, \sigma, \varepsilon_n)$: Choose a secret key dimension n and a distribution $\mathcal{D}$ where the distribution is sampling a random vector of form $\{-1, 0, 1\}^n$ with a fixed Hamming weight h, such that $\mathsf{RLWE}_{n,q,\mathcal{D},\sigma}$ holds. Set ciphertext modulus q, number of bits in plaintext $\ell \leq n$, and standard deviation σ for Gaussian distribution for ciphertext noise generation. Additionally, set minimum integer r such that $\mathsf{erf}(\frac{r}{\sqrt{2}\cdot\sqrt{2h+1}\cdot\sigma}) \leq \varepsilon_n/\ell$.
- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{sRLWE.KeyGen}(\mathsf{pp})$: Draw a secret key $s \leftarrow \mathcal{D}$. Sample $\alpha \xleftarrow{\$} \mathcal{R}_q$ and noise $x \leftarrow \chi_\sigma \in \mathcal{R}$, and compute $\mathsf{pk} = (\alpha, \alpha s + x) \in \mathcal{R}_q \times \mathcal{R}_q$, $\mathsf{sk} \leftarrow s$.
- $\mathsf{ct} = (a, \vec{b}) \leftarrow \mathsf{sRLWE.Enc}(\mathsf{pp}, \mathsf{pk}, \vec{m})$: To encrypt a vector $\vec{m} \in \mathbb{Z}_2^\ell$, define the ring element $t \leftarrow \sum_{i\in[\ell]} \frac{q}{2} \cdot \vec{m}[i] X^{i-1} \in \mathcal{R}_q$. Draw a ring element $e \leftarrow \mathcal{D} \in \mathcal{R}_q$ and noises $x', x'' \leftarrow \chi_\sigma$. Let $b \leftarrow t + x'' = \sum_{i\in[N]} b_i X^{i-1}$. The ciphertext is the pair $(a, \vec{b}) = (\alpha e + x', (b_i)_{i\in[\ell]}) \in \mathcal{R}_q \times \mathbb{Z}_q^\ell$.
- $\vec{m} \leftarrow \mathsf{sRLWE.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{ct} = (a, \vec{b}))$: Let $a' \leftarrow a \cdot \mathsf{sk} := \sum_{i\in[n]} a'_i X^{i-1}$, $\vec{d} = \vec{b} - (a'_i)_{i\in[\ell]}$, $\vec{m} = \lfloor \frac{\vec{d} + q/2}{r} \rfloor \in \mathbb{Z}_2^\ell$.

We formalize the properties of sRLWE, including the (tailored) correctness, CPA security, key privacy (i.e., ciphertexts under different public keys are computationally indistinguishable) and zero-plaintext wrong-key decryption (i.e., given the wrong key, a PVW ciphertext is decrypted into a non-zero plaintext with high probability), as follows. These follow from the analysis above and RLWE.

Theorem 5.1. *Assuming the hardness of RLWE,* sRLWE *satisfies the following properties:*

- *(Correctness) For any* $\lambda > 0, q = \mathsf{poly}(\lambda), \sigma > 0, 1 > \varepsilon_n > 0$ *and* $\ell \leq n$ *for* n *chosen in* $\mathsf{pp}_{\mathsf{sRLWE}}$*, let* $\mathsf{pp}_{\mathsf{sRLWE}} \leftarrow \mathsf{sRLWE.KeyGen}(1^\lambda, \ell, q, \sigma, \varepsilon_n)$*,* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{sRLWE.KeyGen}(\mathsf{pp}_{\mathsf{sRLWE}})$*, for any message* $\vec{m} \in \{0,1\}^\ell$*, it holds that:* $\Pr[\mathsf{sRLWE.Dec}(\mathsf{sk}, \mathsf{sRLWE.Enc}(\mathsf{pk}, \vec{m})) = \vec{m}] \geq 1 - \varepsilon_n - \mathsf{negl}(\lambda)$.
- *(CPA security) For any PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, for the same quantifiers above in correctness, let the adversary choose two messages* $(\vec{m}_1, \vec{m}_2, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}_{\mathsf{sRLWE}}, \mathsf{pk})$*; let* $b \xleftarrow{\$} \{1,2\}, \mathsf{ct} \leftarrow \mathsf{sRLWE.Enc}(\mathsf{pp}_{\mathsf{sRLWE}}, \mathsf{pk}, \vec{m}_b)$*, it holds that:* $|\Pr[\mathcal{A}_2(\mathsf{st}, \mathsf{ct}) = b]| \leq \mathsf{negl}(\lambda)$.
- *(Key privacy) For the same quantifiers above in CPA security, let* $(\mathsf{sk}', \mathsf{pk}') \leftarrow \mathsf{sRLWE.KeyGen}(\mathsf{pp}_{\mathsf{sRLWE}})$*; let the adversary choose a message* $(\vec{m}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}_{\mathsf{sRLWE}}, \mathsf{pk}, \mathsf{pk}')$*; let* $\mathsf{ct} \leftarrow \mathsf{sRLWE.Enc}(\mathsf{pp}_{\mathsf{sRLWE}}, \mathsf{pk}, \vec{m})$*,* $\mathsf{ct}' \leftarrow \mathsf{sRLWE.Enc}(\mathsf{pp}_{\mathsf{sRLWE}}, \mathsf{pk}', \vec{m})$*, it holds that:* $|\Pr[\mathcal{A}_2(\mathsf{st}, \mathsf{ct}) = 1] - \Pr[\mathcal{A}_2(\mathsf{st}, \mathsf{ct}') = 1]| \leq \mathsf{negl}(\lambda)$.
- *(Zero-plaintext wrong-key decryption) For the same quantifiers above in correctness, let* $(\mathsf{sk}', \mathsf{pk}') \leftarrow \mathsf{sRLWE.KeyGen}(\mathsf{pp}_{\mathsf{sRLWE}})$*; it holds that:* $\Pr[\mathsf{sRLWE.Dec}(\mathsf{sk}', \mathsf{sRLWE.Enc}(\mathsf{pk}, 0^\ell)) = 0^\ell] \leq (\frac{r}{q})^\ell + \mathsf{negl}(\lambda)$.

Proof. • (Correctness) The noise of a ciphertext comes from x, x', x'' all sampled from χ_σ. Since $\mathcal{D}$ gives out a binary vector with hamming weight x, there are $2h$ Gaussian noise contributed by x, x', sampled independently. Additionally, x'' is one additional Gaussian noise. In total, there are $2h+1$ independently sampled noises from discrete Gaussian distribution $(0, \sigma)$. The resulting noise x_t is thus from discrete Gaussian distribution $(0, \sqrt{2h+1}\sigma)$. To satisfy $\Pr[r \geq x_t] \geq 1 - p$ for some probability $0 > p > 1$, it is required that $\mathsf{erf}(\frac{r}{\sqrt{2}\cdot\sqrt{2h+1}\cdot\sigma}) \leq p$. As required by sRLWE.GenParam, $\mathsf{erf}(\frac{r}{\sqrt{2}\cdot\sqrt{2h+1}\cdot\sigma}) \leq \varepsilon_n/\ell$. Thus, by union bound, all $\ell \geq 1$ noises together are bounded by r with probability ε_n. Thus, the correctness property follows straightforwardly.

• (CPA security) CPA security is the same as the original RLWE encryption in [26]. Then only change we make is that the ciphertext only contains $\vec{b} \in \mathbb{Z}_q^\ell$ instead of $b \in \mathcal{R}_q$. However, this does not affect the security guarantee (given that the RLWE assumption holds for our parameters).

• (Key privacy) We argue key privacy via a hybrid argument. We introduce the hybrid construction Π_1: the only difference between Π_1 and sRLWE is that instead of computing $a \leftarrow \alpha e + x, b \leftarrow \beta e + x'' + t$ as in Enc (where $t \leftarrow \sum_{i\in[\ell]} \frac{q}{2} \cdot \vec{m}[i] X^{i-1} \in \mathcal{R}_q$ for $\vec{m}$ being the input plaintext of ℓ bits), it first samples $a, b' \xleftarrow{\$} \mathcal{R}_q$ uniformly at random, and then compute $b \leftarrow b' + t$, and outputs (a, b') as the ciphertext. This is simple OTP and thus achieves key privacy trivially. Thus, if there exists an adversary that breaks sRLWE, since it cannot break Π_1, it must break the Ring-LWE assumption.

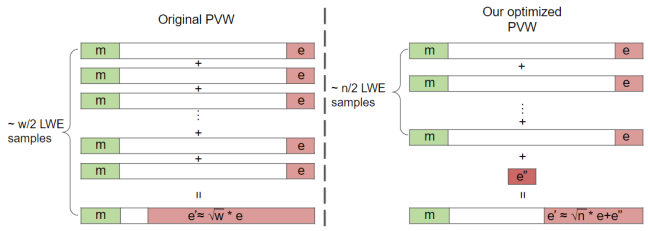


Figure 4: Visualization of the reduced noise of the ciphertexts from our RLWE encryption variant scheme compared to the original PVW scheme

- (Zero-plaintext wrong-key decryption) By RLWE, given a ciphertext $\mathsf{ct} = (a, \vec{b})$ encrypted under sk', $\vec{b}$ is indistinguishable from a random vector sampled from $\mathbb{Z}_q^\ell$ with respect to sk sampled independently from sk'. Therefore, $\vec{b} - a\mathsf{sk}$ results in a random vector sampled from $\mathbb{Z}_q^\ell$, which decrypts to 0^ℓ with probability $(\frac{2r+1}{q})^\ell + \mathsf{negl}(\lambda)$.

□

Efficiency. Since α can be represented as a random seed, and β has $n\log(q)$ bits, the public key size is now $n \cdot \log(q)$ bits, which is much smaller.

Setup for our OMR construction. Our very first step is to replace the underlying PVW scheme in OMRp2 with our RLWE encryption variant sRLWE. This reduces the clue key and sender runtime.

Smaller range r. One additional property of sRLWE compared to PVW is that the value r is much smaller than in § 3.1. sRLWE requires $\mathsf{erf}(\frac{r}{\sqrt{2}\cdot\sqrt{2h+1}\cdot\sigma}) \leq \varepsilon_n/\ell$, while the original PVW requires $\mathsf{erf}(\frac{r}{\sqrt{2}\cdot\sqrt{w}\cdot\sigma}) \leq \varepsilon_n/\ell$ as in [21]. Since $h = O(n)$ and $w = \omega(n\log(n))$, the new noise range is much smaller. We visualize the difference in Fig. 4.

5.2 A New Retrieval Circuit

Another major bottleneck of the prior OMR construction is the detector runtime. Therefore, we shift our focus to Retrieve and design a new and more efficient retrieval circuit.

5.2.1 A New Decryption Circuit for Step 1

Recall that in OMRp2 step 1 (§ 4.2.1), the detector homomorphically decrypts the PVW ciphertexts via a linear transformation followed by a degree-$(t-1)$ polynomial (where t is the plaintext modulus of BFV, $t = q$ for q being the PVW ciphertext modulus). The evaluation of the polynomial requires $O(t)$ homomorphic operations, which is costly for $t > 2D$, where D is the ring dimension of the underlying BFV.

Fortunately, since the new RLWE encryption variant has a much smaller range r, a new more efficient circuit can be designed as follows. The fundamental goal is to compute the following function using a polynomial function: $f(x) = \begin{cases} 0 & \text{if } t-r \leq x \leq r \\ 1 & \text{o.w.} \end{cases}$. While it has t distinct points, implying a degree-$(t-1)$ function, it can be evaluated more efficiently at the cost of having a higher degree, i.e., we represent $f(x) := (\prod_{i=-r}^{r}(x-i))^{t-1}$. This representation chiefly requires two steps of computation: $y = \prod_{i=-r}^{r}(x-i)$, checking whether $x \in [-r, r]$, if so, $y = 0$; otherwise, $y \neq 0$. By Fermat's Little Theorem, y^{t-1} returns 1 iff $y \neq 0$. Therefore, it is equivalent to the defined $f(x)$ above. Furthermore, we optimize the evaluation of y to be $y = \prod_{i=0}^{r}(x^2 - i^2)$, which is equivalent as before, but has $r+1$ multiplications instead of $2r$ multiplications, and thus can be evaluated more efficiently.

Efficiency. In total, to evaluate $f(x)$, only $r + 1 + \log(t-1)$ multiplications are needed.

This decrease in the number of multiplications comes at the cost of increasing the multiplicative degree from $\sim t$ to $\sim r \cdot t$. To evaluate a function of degree k, each multiplication takes $O(D\mathsf{polylog}(k))$ time. Therefore, our new representation takes $O((r + \log(t)) \cdot (\mathsf{polylog}(r) + \mathsf{polylog}(t)))$ time to evaluate, while the original degree-$(t-1)$ polynomial needs $O(t\mathsf{polylog}(t))$ time. As long as $r \ll t$, our method is more efficient. Concretely, with the parameters chosen in § 7, our new representation is ∼20x faster.

We formalize our new construction in Algorithm 1, and the correctness follows from the aforementioned analysis. The more detailed proof is deferred to the full version [23].

Algorithm 1 Our new ClueToPackedPV

procedure PerfOMR1.ClueToPackedPV($\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}} = (\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{\mathsf{sk}}, \mathsf{BB} = (x_i, c_i)_{i\in[N]})$ ▷ $\mathsf{ct}_{\mathsf{sk}} = \mathsf{Enc}(\mathsf{sk}_{\mathsf{BFV}}, \mathsf{sk}_{\mathsf{pvw}})$)
  Parse $c_i = (a_i = \sum_j a_{i,j} X^{j-1}, \vec{b} = (b_{i,l})_{l\in[\ell]})$
  Let $d \leftarrow \lceil N/D \rceil$
  for $k \in [d]$ **do** ▷ D is the ring dimension of the BFV scheme
    for $l \in [\ell]$ **do**
      for $u \in [D]$ **do**
        Let $\vec{a}_u = (\ldots) \in \mathbb{Z}_q^{n\times 1}$
      Let $A \leftarrow (\vec{a}_1 || \ldots || \vec{a}_D) \in \mathbb{Z}_q^{N\times n}$
      Homomorphically compute $\mathsf{ct}_1 \leftarrow \mathsf{sk}_{\mathsf{pvw}} \times A^\intercal$
      Let $\vec{b'} \leftarrow b_{k\cdot d+1,l} || \ldots || b_{k\cdot d+D,l}$
      Homomorphically compute $\mathsf{ct}_{k,l} \leftarrow \vec{b'} - \mathsf{ct}_1$
      $\mathsf{BFV.Eval}(\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{vkl}, h \circ g)$, where $g(x) = \prod_{i=0}^{r}(x^2 - i^2)$ and $h(x) = x^{t-1}$
    Homomorphically compute $\mathsf{ct}_v \leftarrow \prod_{l\in[\ell]} \mathsf{ct}_{k,l}$
  return $(\mathsf{ct}_k)_{k\in[d]}$

Theorem 5.2. ClueToPackedPV *in Algorithm 1 is correct (Def 4.1) given the correctness of the BFV scheme.*

5.2.2 An Efficient PV Unpacking Algorithm for Step 2

After step 1 (constructed above, defined in Def 4.1), we obtain $d = \lceil N/D \rceil$ ciphertexts, each of which encrypts D bits, indicating whether the N messages are pertinent or not. For step 2 (Def 4.2), we need to expand these into $N' = N/\mathsf{v}$ ciphertexts, each of which encrypts a single integer in all of its D slots.

Recall that v is the bundle size (i.e., we bundle v messages into a single one for efficiency). We first focus on $\mathsf{v} = 1$ as in OMRp2 (construction in prior work) for simplicity, and thus $N' = N$. OMRp2 achieves this by performing $O(N\log(D))$ homomorphic operations (or $O(\log(D))$ levels of multiplications but with $O(N)$ operations as in [22])[11]. In this section, we introduce an algorithm with $O(N)$ homomorphic operations and a single multiplication level.

Message Extraction. Let us start with a single ciphertext ct encrypting $(m_1, \ldots, m_N) \in \mathbb{Z}_t^N$. Recall that in BFV, the message vector is first encoded into a polynomial before encryption (see § 3.2). This encoding is to make the multiplications between messages easier over $\mathbb{Z}_t$, while our goal instead, is to unpack these messages into individual BFV ciphertexts. Thus, we first reverse this encoding and extract the message in each slot out to each coefficient of the encoded polynomial.

In other words, we homomorphically revert the encoding process by performing the SlotToCoeff step introduced in [24, Sec 4.4], formally presented in Algorithm 2.

Unpacking. After obtaining a ciphertext ct' encrypting a polynomial $m(X) = \sum_{i\in[D]} m_i X^{i-1}$, we want to obtain D ciphertexts $\mathsf{ct}'_1, \ldots, \mathsf{ct}'_D$, such that each ciphertext ct'_i is encrypts a constant polynomial $p_i(X) = m_i$ (recall that a constant polynomial encodes a vector $(m_i, \ldots, m_i) \in \mathbb{Z}_t^D$, i.e., $p_i(\eta) = m_i$ for all $\eta \in \mathbb{Z}_t$). To do so, the detector performs the oblivious expansion procedure introduced in [1]. This well-established procedure is recalled in Algorithm 2 OExpand.

Allowing $\mathsf{v} > 1$. Despite the great efficiency improvement with the unpacking technique above (both SlotToCoeff and OExpand take only $O(D)$ operations per ciphertext), fundamentally, this requires $O(N)$ homomorphic operations for N messages, which is still quite costly.

One natural idea is to bundle $\mathsf{v} \ll N$ messages as a single one (v to be fixed later), reducing N messages to $N' = N/\mathsf{v}$ messages before performing this PV unpacking process. The number of operations thus reduces to $O(N')$. In more detail, with d input ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_d)$ (assuming v divides d for simplicity), the detector first divides them into v chunks: $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{d/\mathsf{v}}), (\mathsf{ct}_{d/\mathsf{v}+1}, \ldots, \mathsf{ct}_{2(d/\mathsf{v})}), \ldots$. Then, it adds up all the chunks ciphertext-wise, i.e. computing $\tilde{\mathsf{ct}}_i \leftarrow \sum_{j=0}^{\mathsf{v}-1} \mathsf{ct}_{j\cdot(d/\mathsf{v})+i}$ for $i \in [d/\mathsf{v}]$. This gives d/v ciphertexts, each with D slots, where each slot encrypts the summation of v slots of the input ciphertexts. After obtaining these d/v ciphertexts, everything proceeds as the unpacking procedure described above (expanding each slot into a single ciphertext).

[11] [22] gives a flexibility to trade off the number of operations vs. levels.

Putting everything together, we obtain our PVUnpack algorithm as in Algorithm 2. Correctness follows from SlotToCoeff and OExpand, and the proof is deferred to the full version [23].

Algorithm 2 Our new PVUnpack

procedure OExpand(ct) (Adapted from [1])
  ▷ All the keys needed to complete this procedure are assumed to be implicitly taken.
  $\mathsf{res} \leftarrow [\mathsf{ct}]$
  for $i = 0$ to $\log D$ **do**
    for $j = 0$ to $2^i - 1$ **do**
      $\mathsf{tmp}_0 \leftarrow \mathsf{res}[j]$
      $\mathsf{tmp}_1 \leftarrow \mathsf{tmp}_0 \cdot x^{-2^i}$
      $\mathsf{tmp}'_j \leftarrow \mathsf{tmp}_0 + \mathsf{Substitute}(\mathsf{tmp}_0, D/2^i + 1)$
      $\mathsf{tmp}'_{i+2^j} \leftarrow \mathsf{tmp}_1 + \mathsf{Substitute}(\mathsf{tmp}_1, D/2^i + 1)$
    $\mathsf{res} \leftarrow [\mathsf{tmp}'_0, \ldots, \mathsf{tmp}'_{2^{i+1}-1}]$
  for $i = 0$ to D **do**
    $\mathsf{res}[i] \leftarrow \mathsf{BFV.Eval}(\mathsf{res}[i], 1/D, \times)$
  return res
procedure SlotToCoeff(ct) (Adapted from [24])

$$U := \begin{pmatrix} 1 & \zeta_0 & \zeta_0^2 & \cdots & \zeta_0^{N-1} \\ 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{\frac{N}{2}-1} & \zeta_{\frac{N}{2}-1}^2 & \cdots & \zeta_{\frac{N}{2}-1}^{N-1} \\ 1 & \bar{\zeta}_0 & \bar{\zeta}_0^2 & \cdots & \bar{\zeta}_0^{N-1} \\ 1 & \bar{\zeta}_1 & \bar{\zeta}_1^2 & \cdots & \bar{\zeta}_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \bar{\zeta}_{\frac{N}{2}-1} & \bar{\zeta}_{\frac{N}{2}-1}^2 & \cdots & \bar{\zeta}_{\frac{N}{2}-1}^{N-1} \end{pmatrix} \in \mathbb{Z}_t^{N\times N}$$

  Homomorphically compute $\mathsf{res} \leftarrow \mathsf{ct} \cdot U^\intercal$,
  return res
procedure PerfOMR1.PVUnpack($(\mathsf{ct}_1, \ldots, \mathsf{ct}_d), \mathsf{v}$)
  for $i \in [d/\mathsf{v}]$ **do**
    $\mathsf{ct}'_i \leftarrow \sum_{j\in[\mathsf{v}]} \mathsf{ct}_{j\cdot\mathsf{v}+i}$
  for $i = 0$ to d/v **do**
    $\mathsf{tmp}_i \leftarrow \mathsf{SlotToCoeff}(\mathsf{ct}'_i)$
    $[\mathsf{ct}'_{i\cdot D+1}, \ldots, \mathsf{ct}'_{i\cdot D+D}] \leftarrow \mathsf{OExpand}(\mathsf{tmp}_i)$
  return $(\mathsf{ct}'_i)_{i\in[N']}$ (where $N' = d \cdot D/\mathsf{v}$)

Theorem 5.3. PVUnpack *in Algorithm 2 is correct (Def 4.2) given the correctness of the underlying BFV scheme.*

5.2.3 A General Encoding Procedure for Step 3

Lastly, we discuss digest encoding. As aforementioned, v messages are bundled as a single one for efficiency. Therefore, to deliver all the pertinent payloads, the payloads of these v messages bundled together should also be concatenated and viewed as a single large payload. In other words, we consider the concatenated payloads of the form $x'_i \leftarrow x_{i_1} || \ldots || x_{i_\mathsf{v}}$. The corresponding ciphertext ct'_i indicating whether x'_i is pertinent

(if ct'_i encrypts a value > 1, x'_i should be included in the digest and otherwise not)[12]. Hence, the detector performs digest encoding as if there were only N/v messages.

Overall, this speeds up the step 2 of the detector (nearly) v times. However, there are several issues to address:

Spurious payloads. The decoding algorithm will output all v messages in the bundle, some of which may be impertinent. This is allowed by v-soundness as defined in Def 3.2, and often the application using OMR can filter the bundled impertinent messages as discussed in Remark 3.4.

Alternatively, we can do a final filtering (and thus attain the original OMR soundness definition [21, Thm 4.1]) by a small addition to the scheme: the sender will encrypt the payloads to the recipient, using an encryption scheme with the property that decryption using the wrong key is detectable. See [23, Apd A.1] for details.

Intra-bundle collisions. A more serious issue is that the index encoding process would fail if using the realization in § 4.2.2. Recall that during the index encoding part of the original ExpandedPVToDigest step, each message is assigned to a bucket Y_j. Then, for each bucket Y_j, compute $\mathsf{Acc}_j \leftarrow \sum_{i\in Y_j} i \cdot \mathsf{ct}_i$ and $\mathsf{ctr}_j \leftarrow \sum_{i\in Y_j} \mathsf{ct}_i$. The decoding procedure views all the bucket j with ctr_j encrypting a value > 1 as having collisions and discards that bucket.

Now, when there is at most one pertinent message among each bundle of v messages, ct'_i indeed encrypts 0 or 1 and the original decoding procedure (§ 4.2.3) works. However, when there are multiple pertinent messages (either true or false positives) in the same bundle, the counter ctr in the associate bucket (i.e., sum of ct'_i for the bundled message i in the bucket) encrypts a number greater than 1, causing those buckets to be treated as a collision of different (bundled) messages and discarded.

To solve this, we use a different index encoding scheme, supporting pertinency vector entries that are greater than 1. Let $N' \leftarrow \lceil N/\mathsf{v} \rceil$ be the total number of bundled messages. At a high level, we expand each single bit of $i \in [N']$ into $\log(\mathsf{v}+1)$ bits and combine the expanded bits into a new index $i' \in [N' \cdot (\mathsf{v}+1)]$. In this way, when multiplying i' by v, no carrying occurs into the next expanded bit..

More formally, we process as follows. For any index $i \in [N']$, let $i[j]$ denotes the j-th bit of i. We define the following function to expand each bit into $\log(\mathsf{v}+1)$ bits: $i' \leftarrow \mathsf{BitExpand}(\mathsf{v}, i) := \sum_{j=1}^{\lceil \log(N'+1) \rceil} i[j](\mathsf{v}+1)^{j-1}$. In other words, this function represents a binary value using a $(\mathsf{v}+1)$-ary value: $0,1 \in \mathbb{Z}_2$ are encoded by $0,1 \in \mathbb{Z}_{\mathsf{v}+1}$.

Then, the index encoding process uses the new index i' instead of i. In more detail, the detector randomly assigns each bundled message i into a bucket j for $j \in [m]$ (where m is the number of buckets and $m > \bar{k}$ for $\bar{k}$ being the upper bound on the number of pertinent messages). Let Y_j represent the set of indices (of messages) assigned to bucket j. Then, as before, for bucket j, the detector computes $\mathsf{Acc}_j \leftarrow \sum_{i\in Y_i} \mathsf{ct}'_i \cdot i'$; and computes $\mathsf{ctr}_j \leftarrow \sum_{i\in Y_j} \mathsf{ct}'_i$ for bucket j.

The collision happens if and only if one of the following conditions happens: (1) $r_j > \mathsf{v}$ for $r_j \leftarrow \mathsf{Dec}(\mathsf{ctr}_j)$; (2) let $b_j \leftarrow \mathsf{Dec}(\mathsf{Acc}_j)$ and $b_j = \sum_i b_{j,i}(\mathsf{v}+1)^{i-1}$, there exists an i such that $b_{j,i} \neq 0 \wedge b_{j,i} \neq r_j$.

This process, again, is repeated for C times to guarantee that all the indices can be successfully recovered.

Toy example. We provide an example of this encoding method. Assume we have $N' = 32$ bundled messages, where bundle $15, 20, 25$ contain pertinent messages, and $\mathsf{v} = 7$. After buildling the messages accordingly, we obtain $\mathsf{ct}'_{15} = \mathsf{Enc}(1), \mathsf{ct}'_{20} = \mathsf{Enc}(5), \mathsf{ct}'_{25} = \mathsf{Enc}(7), \mathsf{ct}'_{i\notin\{15,20,25\}} = \mathsf{Enc}(0)$: We first get the binary representation of the indices: $15 = 001111, 20 = 010100, 25 = 011001$.

Then we encode all of them using BitExpand: $\mathsf{BitExpand}(15,7) = 000{,}000{,}001{,}001{,}001{,}001$,
$\mathsf{BitExpand}(20,7) = 000{,}001{,}000{,}001{,}000{,}000$, and
$\mathsf{BitExpand}(25,7) = 000{,}001{,}001{,}000{,}000{,}001$.

If there is no collision, i.e., if those three pertinent messages are assigned into different buckets, then these buckets' Acc encode $15 \cdot 1 = 000{,}000{,}001{,}001{,}001{,}001$, $20 \cdot 5 = 000{,}101{,}000{,}101{,}000{,}000$, and $25 \cdot 7 = 000{,}111{,}000{,}111{,}000{,}000$. The corresponding ctr's are $1 = 001, 5 = 101, 7 = 111$. If there is a collision, say 15 and 20 collide, then Acc for that colliding bucket would be decrypted to $000{,}101{,}001{,}110{,}001{,}001$ while the counter is $1+5 = 6 = 110$; mismatch happens: $110 \neq 001 \neq 101$ (where $001{,}101$ appears in the Acc decryption above). The recipient can identify such a collision. On the other hand, if 15 and 25 collide, ctr is $8 > \mathsf{v} = 7$, which is clearly a collision.

Decoding. Recipient decoding is then straightforward: first checks whether the counter encrypts a number $> \mathsf{v}$, and if so, there is a collision. Then, the recipient checks whether the Acc number matches the ctr number.

Again, after such index encoding, the payloads are encoded using the sparse matrix the same way as in § 4.2.3, so we omit the details. We formalize all these in Algorithm 3The correctness holds as the intuition discussed above, and the proof is deferred to the full version [23].

Theorem 5.4. ExpandedPVToDigest *in Algorithm 3 is correct (Def 4.3 with* DecodeDigest *defined in Algorithm 3) given the correctness of the underlying BFV scheme.*

Remark 5.5. As mentioned in § 4.2.3, we do not use sparse matrix A but instead a uniform random matrix (see line 9 in Algorithm 3) as the weights to compute the random linear combinations of the payloads. In [21], a sparse matrix is suggested to boost efficiency (as if a cell is 0, the corresponding multiplication can be skipped). However, in practice, this may not be the case. This is mainly because by the SIMD natural of BFV ciphertext, a single ciphertext can store $D \cdot \log(t)$

[12] Recall that ct'_i encrypts the summation of v slots in the input ciphertexts of PVUnpack corresponding to message $x_{i_1}, \ldots, x_{i_\mathsf{v}}$.

Algorithm 3 Our new ExpandedPVToDigest

procedure PerfOMR1.ExpandedPVToDigest($\mathsf{pp}, \mathsf{pk}_{\mathsf{detect}}$, $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{N'}, \bar{k}), \mathsf{BB} = ((x_i, \cdot))_{i\in[N]}$)
  $\hat{k} \leftarrow \bar{k} + N\log(N)\varepsilon_{\mathsf{p}}$ ▷ Recall that ε_{p} is the false positive rate included in pp.
  Choose C, m s.t:
  (1) $C \cdot m$ is minimized
  (2) the index encoding fails with probability $\mathsf{negl}(\lambda)$ (i.e., decoding using eliminating the collisions fails with negligible probability)
    ▷ Failure probability is $1 - \prod_{i=1}^{\hat{k}-1}(1 - (\frac{i}{m})^C)$ per [21, Sec 6.1.2]
  (3) each bucket is assigned at most $t-1$ messages except with negligible probability
    ▷ Overflow probability is $m \cdot \exp(-\frac{(2m'-1)^2}{2m'+1}\frac{t/2}{m'})$ where $m' \leftarrow m/D$
  Choose K such that a random matrix in $\mathbb{Z}_t^{K\times\bar{k}}$ is full rank with $1 - \mathsf{negl}(\lambda)$ probability
    ▷ $K = O(\bar{k} + \lambda/\log(t))$ as discussed in [21].
  for $i \in [D], j \in [0, \frac{N'}{D} - 1]$ **do**
    $x'_{j\cdot D+i} \leftarrow x_{j\cdot\mathsf{v}+1} || \ldots || x_{j\cdot\mathsf{v}+\mathsf{v}}$
  for $i \in [C]$ **do**
    Initialize $\mathsf{Acc}_{i,u}, \mathsf{ctr}_{i,u}$ for $u \in [m]$
    for $j \in [d/\mathsf{v}]$ **do**
      $u \xleftarrow{\$} [m]$
      $\mathsf{Acc}_{i,u} \leftarrow \mathsf{Acc}_{i,j} + \mathsf{ct}_j \cdot \mathsf{BitExpand}(\mathsf{v}, \mathsf{binary}(j))$
        ▷ Note that for each accumulator Acc, the calculation needs to be split into multiple $\mathbb{Z}_t$ elements. Each $\mathbb{Z}_t$ element contain $t' = \lfloor \log(t)/\log((\mathsf{v}+1)) \rfloor$ bits, and thus totally need $\lceil \log(N/(\mathsf{v}+1))/t' \rceil$ $\mathbb{Z}_t$ elements.
      $\mathsf{ctr}_{i,u} \leftarrow \mathsf{Acc}_{i,j} + \mathsf{ct}_j$
  $A \xleftarrow{\$} \mathbb{Z}_t^{K\times N'}$
  Homomorphically computes $\mathsf{comb} \leftarrow (A \circ Z) \times (\mathsf{ct}'_i)_{i\in[N']}$ where $Z = \begin{pmatrix} x'_1, x'_2, \ldots, x'_{N'} \\ \vdots \\ x'_1, x'_2, \ldots, x'_{N'} \end{pmatrix} \in \mathcal{P}^{K\times N'}$
  return $M = (s, (\mathsf{Acc}_{i,u}, \mathsf{ctr}_{i,u})_{i\in[C],u\in[m]}, \mathsf{comb})$
procedure PerfOMR1.DecodeDigest(M, sk)
  $\hat{k} \leftarrow \bar{k} + N\log(N)\varepsilon_{\mathsf{p}}$
  Parse $M = (s, (\mathsf{Acc}_{i,j}, \mathsf{ctr}_{i,j})_{i\in[C],j\in[m]}, \mathsf{comb})$
  Initialize an empty set$P = \{\}$ to record all pertinent indices
  for $i = 1$ to C **do**
    for $j = 1$ to m **do**
      $a, c \leftarrow \mathsf{BFV.Dec}(\mathsf{sk}, (\mathsf{Acc}_{i,j}, \mathsf{ctr}_{i,j}))$
      If $c > \mathsf{v}$, skip this iteration
      Parse a into $(\mathsf{v}+1)$-ary numbers,
        i.e., $a = \sum_i a_i(\mathsf{v}+1)^i$
      If any $a_i \neq c$, skip this iteration
      Let $a' = \sum_i a'_i$ where $a'_i = 1$ if $a_i \neq 0$,
        and $a'_i = 0$ if $a_i = 0$.
      Add a' to P
  If $|P| > \hat{k}$, $\mathsf{PL} = \mathsf{overflow}$ and skip the next step
  Use P, comb, s and Gaussian elimination to solve for all payloads PL
  If failed, $\mathsf{PL} = \mathsf{overflow}$
  return PL

bits of information, and thus can store $W = D \cdot \log(t)/P$ (for $\mathcal{P} = \{0,1\}^P$) linear combinations. In this case, as long as one of the W corresponding weights is non-zero for a particular payload, the whole ciphertext needs to be multiplied. Therefore, practically, a sparse matrix does not reduce the number of multiplications, unless $\mathcal{P}$ is large enough (e.g., such that $W = 1$). One can easily change this step to use a sparse matrix using the Sparse Random Linear Coding (SRLC) discussed in [21, Section 6]. To avoid extra complicity, we omit the details about SRLC.

5.3 Putting Everything Together

Putting everything above together yields a more efficient OMR construction. The pseudocode is presented in Algorithm 4The correctness, soundness, and compactness follow from the correctness of the three sub-procedures discussed above. Privacy holds by the key privacy of sRLWE. The proof is deferred to the full version [23].

Theorem 5.6. *The scheme* PerfOMR1 *is an* OMR *scheme (with* v*-soundness) for $N < D \cdot t/2$, assuming the hardness of* RLWE*, the correctness of* BFV *leveled HE. Moreover* PerfOMR1 *is also* v*-compact.*

5.4 Optimizations

Next, we introduce several implementation-level optimizations that further improve efficiency. Note that these optimizations are all implementation-level and compute the same algorithm proposed above, so all the properties trivially hold.

One-time rotation of sk**.** Originally in [21], there are ℓ secret key vectors independently sampled (forming an sk matrix). During homomorphic decryption (i.e., step ClueToPackedPV), when computing the vector-matrix multiplication $a \cdot \mathsf{sk}$, the ℓ secret key vectors need to be rotated for a total of $n \cdot \ell$ times (note that only a single encryption of sk and a single rotation key is sent to minimize the size of the detection key). However, with our use of sRLWE, we compute $a \cdot \mathsf{sk}$ over $\mathcal{R}_t$, where sk is a single $\mathbb{Z}_t^n$ vector and a is a (structured) matrix. This means that we only need to rotate sk n times. Furthermore, we rotate the sk once at the beginning of the retrieval and store them all in the memory. This implements $a \cdot \mathsf{sk}$ for all N messages using just n rotations, instead of nN/D rotations for a naive implementation.

Improved linear transformation. Recall that in step PVUnpack, we need to compute a linear transformation as in Algorithm 2 line 16, which takes N rotations and N homomorphic multiplications. We reduce this to just $2\sqrt{N}$ rotations using the Baby-step-giant-step algorithm of [17].

Two-level oblivious expansion. Recall that in OExpand in Algorithm 2, we expand a single ciphertext with D slots to D ciphertexts. However, a naive call of this function directly

Algorithm 4 PerfOMR1: Practical Oblivious Message Retrieval

Let $f_s(x)$ be a PRF. Let BFV and sRLWE be as defined above.

procedure PerfOMR1.GenParam($1^\lambda, \varepsilon_p, \varepsilon_n$)
  Choose $\mathsf{pp}_{\mathsf{BFV}} = (D, t, \dots)$ such that homomorphically evaluate Retrieve with all but negligible probability
  Choose $\ell, q = t, \sigma$ such that with the r generated below, it satisfies that $(\frac{2r+1}{q})^\ell \le \varepsilon_p$ [13]
  $\mathsf{pp}_{\mathsf{PVW}} = (n, \ell, q, \sigma, r, \mathcal{D}) \leftarrow$ sRLWE.GenParam($1^\lambda, \ell, q, \sigma, \varepsilon_n$)
  return $\mathsf{pp} = (\mathsf{pp}_{\mathsf{BFV}}, \mathsf{pp}_{\mathsf{PVW}}, \varepsilon_p, \varepsilon_n)$
procedure PerfOMR1.KeyGen(pp)
  $(\mathsf{sk}_{\mathsf{sRLWE}}, \mathsf{pk}_{\mathsf{sRLWE}}) \leftarrow$ sRLWE.KeyGen($\mathsf{pp}_{\mathsf{BFV}}$)
  $(\mathsf{sk}_{\mathsf{BFV}}, \mathsf{pk}_{\mathsf{BFV}}) \leftarrow$ BFV.KeyGen($\mathsf{pp}_{\mathsf{BFV}}$)
  $\mathsf{ct}_{\mathsf{sk}_{\mathsf{sRLWE}}} \leftarrow$ BFV.Enc($\mathsf{pk}_{\mathsf{BFV}}, \mathsf{sk}_{\mathsf{sRLWE}}$)
  return $(\mathsf{sk} = (\mathsf{sk}_{\mathsf{BFV}}), \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}} = \mathsf{pk}_{\mathsf{sRLWE}}, \mathsf{pk}_{\mathsf{detect}} = (\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{\mathsf{sk}_{\mathsf{sRLWE}}})))$
procedure PerfOMR1.GenClue(pp, $\mathsf{pk}_{\mathsf{clue}}, x$)
  $\vec{b} \leftarrow (0, \dots, 0) \in \mathbb{Z}_2^\ell$
  $c \leftarrow$ sRLWE.Enc($\mathsf{pk}_{\mathsf{clue}}, \vec{b}$)
  return c ▷ $c \in \mathcal{R}_q \times \mathbb{Z}_q^\ell$
procedure PerfOMR1.Retrieve(pp, BB, $\mathsf{pk}_{\mathsf{detect}}, \bar{k}$)
  Select v such that the runtime of ClueToPackedPV and PVUnpack are similar and also v divides N/D
  $(\mathsf{ct}_i)_{i \in [N/D]} \leftarrow$ ClueToPackedPV(pp, $\mathsf{pk}_{\mathsf{detect}}$, BB)
  $(\mathsf{ct}'_i)_{i \in [N']} \leftarrow$ PVUnpack(pp, $\mathsf{pk}_{\mathsf{detect}}, (\mathsf{ct}_i)_{i \in [N/D]}$, v)
  $M \leftarrow$ ExpandedPVToDigest($\mathsf{pp} = (c, C, m)$, $\mathsf{pk}_{\mathsf{detect}}, (\mathsf{ct}'_1, \dots, \mathsf{ct}'_{N'}, \bar{k})$, BB)
  return M
procedure PerfOMR1.Decode(M, sk)
  return PerfOMR1.DecodeDigest(M, sk)

produces D ciphertexts. Concretely this is very costly: each ciphertext has a size ∼100kB, so with $D = 32768$ (cf. § 7), this takes 3.2GB of memory (and it gets worse for multi-thread). To reduce the memory cost, we first expand the input ciphertext to l_1 ciphertexts (to-be-fixed later), and then, for each expanded ciphertext, further expand it to $l_2 = D/l_1$ ciphertexts. In this way, we keep only $l_1 + l_2$ ciphertexts in memory and expand on the fly for each batch of l_2 ciphertext.

6 OMR with Further Reduced Key and Clue Sizes

While PerfOMR1 in § 5 greatly improves the detector runtime, which is indeed a major practical concern of deploying OMR in real-world applications, there are some other practical considerations.

Recall that in the previous construction, the clue size remains to be $\sim(n \cdot \log(q))$. However, since now sRLWE relies on sparse secret keys, practically, n needs to be larger than in [21] (relying on uniform keys) for the same security level. Therefore, the concrete clue size increases. Additionally, since the circuit for homomorphic decryption is deeper (§ 5.2.1), the detection key size (poly-logarithmically in the depth) is also larger. Lastly, while it has already been greatly reduced in PerfOMR1, the clue key size is always better to be smaller.

Taking these three costs into consideration, we propose an alternative construction that makes these reduce the size of the clue, the detection key, and the clue key, at the cost of making the detection time larger than our protocol in § 5, while still slightly smaller than in [21].

6.1 Reducing the Clue Field

In PerfOMR1 above (as in OMRp2 [21]), we use $q = t$ so that decryption of the clues' PVW ciphertexts (defined over $\mathbb{Z}_q$) can be done directly via BFV homomorphic operations (defined over $\mathbb{Z}_t$). This modulus matching provides the mod q reductions for free. However, it forces q to match the large t needed by BFV, and we pay for this in the size of clues and clue keys (both $O(n \cdot \log(q))$). Removing this constraint gives us the flexibility to reduce the clue size, and we indeed do so when sRLWE uses (sparse) short secrets, as follows.

We set $q \ll t$ (q to be even for simplicity) such that evaluating the PVW decryption in $\mathbb{Z}_t$ is the same as working in $\mathbb{Z}$ (i.e., no modular reductions). Then, when the PVW decryption performs its $\mathbb{Z}_q$ range test, we need to account for all the congruent values in $\mathbb{Z}$.

Formally, for clue $(a, \vec{b}) \in \mathcal{R}_q \times \mathbb{Z}_q^\ell$, by computing $\vec{b} - (a'_i)_{[\ell]}$ where $a' = a \cdot s$ over $\mathcal{R}$ (instead of $\mathcal{R}_q$), we obtain $(\mu_i + \kappa_i \cdot q)$ for some $\kappa_i \in \mathbb{Z}$ satisfying $||\mu_i + \kappa_i \cdot q|| \le t/2$ for all $i \in [\ell]$. If the message is pertinent, $\mu_i \in [-r, r]$ for the noise bound r. Otherwise, μ_i is indistinguishable from uniform in $[-q/2, q/2)$. Furthermore, since s has some fixed Hamming weight h, we bound $|\kappa_i|$'s by $B+1$, such that $\Pr[\kappa_i \ge B+1] \le \mathsf{negl}(\lambda)$ using h. The detailed analysis involving h is deferred to the full version [23].

A new homomorphic decryption circuit. The homomorphic decryption in $\mathbb{Z}_t$ can thus be realized as follows: first compute $c_i \leftarrow \vec{b}[i] - a'_i \in \mathbb{Z}_t$ for all $i \in [\ell]$ as before; then homomorphically evaluate the function f' over c_i:

$$f'(x) = \begin{cases} 0 & \text{if } x \in [\kappa \cdot q - r, \kappa \cdot q + r] \text{ for all } \kappa \in [-B, B] \\ 1 & \text{else if } x \in [-(B+1)q, (B+1)q] \end{cases}$$

This function has $2(B+1)q$ distinct points, and therefore degree $2(B+1)q - 1$ (requiring $O(Bq)$ operations). We can evaluate it as a polynomial using Lagrange interpolation.

Efficiency. The clue key and clue are both of size $O(n \cdot \log(q))$, and thus shrink now as we choose a smaller q. Fur-

[13] Such a choice exists since, given a fixed q, we can choose σ such that $r = \mathsf{poly}(\sigma)$ is small enough (e.g., $r = (q-1)/2$ and sufficiently large ℓ to satisfy the condition), then choose n sufficiently large to maintain the security level. Our implementation uses a choice that minimizes the cost of homomorphically evaluating the induced decryption circuit.

thermore, since q is smaller, we can maintain the noise distribution while reducing n as well.

On the other hand, the degree of the function is now $2(B+1)q-1$ instead of $r \cdot t - 1$. Thus, with careful parameter selection, the depth is greatly reduced as well. Therefore, the detection key size is also reduced.

The downside is that, evaluating f' homomorphically requires $O(Bq)$ homomorphic operations, compared to only $O(r+\log(t))$ operations before. Moreover, since the noise distribution remains the same, $\frac{2r+1}{q}$ becomes larger. Therefore, the guarantee the same wrong-key decryption probability (the false positive rate in OMR), ℓ needs to be larger. For the concrete results of this tradeoff, see § 7.

Since the construction is very similar to Algorithm 4, and only the homomorphic decryption circuit is changed, we defer the pseudocode and proof to the full version [23]

Theorem 6.1. *The scheme* PerfOMR2 *is an* OMR *scheme (with* v*-soundness) for $N < D \cdot t/2$, assuming the hardness of* RLWE*, the correctness of* BFV *leveled HE. Moreover,* OMRp2 *is also* v*-compact.*

7 Evaluation

We implemented the above PerfOMR1 and PerfOMR2 schemes in a C++ library (to be released as open source, with the optimizations introduced in § 5.4). Our code extends the OMR library [31] and uses the SEAL [28] and PALISADE [33] libraries. We then compare our constructions to an improved version of OMRp2 introduced in [22]. We run our implementation and the improved OMRp2 using Google Compute Cloud `e2-standard-4` with 16GB RAM for a fair comparison.

Parameters. We chose the number of messages to be $N = 2^{19}$, and let $\bar{k} = k = 50$ (i.e., the upper bound and the real total number of pertinent messages are both 50)[14].

For OMRp2, we reuse the parameters in [22] and guarantee > 120-bit of computational security. Therefore, for the sRLWE used in PerfOMR1 we choose $n = 1024, q = 65537, \sigma = 0.5, h = 32, \ell = 2$. For the BFV used in PerfOMR1, we choose $D = 32768, Q \approx 2^{905}, t = q = 65537$. For the sRLWE used in PerfOMR2, we choose $n = 512, q = 400, \sigma = 0.5, h = 32, \ell = 6$. For the BFV used in PerfOMR2, we choose $D = 32768, Q \approx 2^{808}, t = 65537$. All these parameter settings guarantee > 128-bit of security by [11]. Furthermore, same as in [21], we choose the false negative rate $\varepsilon_n = 2^{-30}$ and false positive rate $\varepsilon_p = 2^{-21}$, range $r = 19$, and the κ_i bound in § 6 as $B = 29$. We choose $\ell_1 = 1024, \ell_2 = 32$ for our two-level oblivious expansion (see § 5.4). We used $m = 400$ buckets and $C = 16$ trials. For the the random matrix A (line 9 in Algorithm 3), we choose $K = 53$.

[14] For simplicity, since $k = \bar{k}$, we set $\hat{k} = \bar{k}$ instead of $\hat{k} = \bar{k} + N\log(N)\varepsilon_p$. One can also alternatively view it as we choose $\bar{k} = 45$ and $\hat{k} = 50$.

Lastly, we choose $v = 8$ for PerfOMR1 and $v = 2$ for PerfOMR2 (discussed later).

7.1 Results

Representative costs. Table 2 summarizes the main cost metrics of all our schemes and the baseline (i.e., OMRp2 in [21] integrated with optimizations in [22]).

Our PerfOMR1 is about 15x faster than OMRp2 in terms of the detector runtime. Furthermore, the clue key size is reduced by roughly 60x. On the other hand, the clue size is about 2.2x worse (since sRLWE uses sparse keys, n for PerfOMR1 is larger than the n used for OMRp2). The detection key is also about 10% worse (due to the larger depth, we needed a larger Q). Lastly, since we choose $v = 8$, the digest size is also increased, but only by about 2.5x, since originally the digest contains 2 ciphertexts, one for index encoding and one for payload encoding. With the current parameter setting, we need 4 ciphertexts for payload encoding (the 8 concatenated payloads together have a size of 4896 bytes and one BFV ciphertext can encrypt at most 65536 bytes), but still only one ciphertext for index encoding. See below for how the runtime and digest size scale with v.

For PerfOMR2, the runtime is slightly better ($\sim$2.7x) than OMRp2 (mainly due to the improved PVUnpack step for $v = 1$ in § 4.2.2). The clue key size is drastically decreased: roughly 235x smaller than the clue key size of OMRp2. The clue is about 1.6x smaller. The detection key and digest size both remain roughly the same. Therefore, PerfOMR2 is strictly better than OMRp2, while having some complicated tradeoffs compared to PerfOMR1.

Improvement of each individual step. Fig. 5 shows the runtime breakdown for the three main steps of our schemes, compared to OMRp2 (all using $v = 1$ for fair comparison).

For PerfOMR1, our ClueToPackedPV is much faster than OMRp2 (about 15x faster); the bottleneck is PVUnpack, which is why we choose $v = 8$ for PerfOMR1 to optimize the runtime in the rest of our benchmarks. Conversely, for PerfOMR2, the runtime of ClueToPackedPV is roughly the same as OMRp2, which is thus the bottleneck. Therefore, we choose $v = 1$ for PerfOMR2. Moreover, our PVUnpack is about 5x faster than the PVUnpack in [21] even for $v = 1$.

Lastly, in both of our constructions, ExpandedPVToDigest remains similar to [21], while our encoding scheme requires slightly more operations.

How costs scale with v. As shown in Fig. 6, the costs of our two constructions changes with v: the runtime decreases as v increases. However, since it only boosts the PVUnpack step, enlarging v only works well when PVUnpack is a bottleneck. For example, for our PerfOMR1, the runtime with $v = 8$ is about 2x faster than the runtime with $v = 1$. Conversely, the runtime for PerfOMR2 is only about 20% faster when changing v from 1 to 8. Additionally, the digest size also grows with

	Detector Runtime (ms/msg)	Clue Key Size (kB)	Clue Size (Bytes)	Detector Key Size (MB)	Digest Size (Bytes/msg)	Recipient Runtime(ms)
OMRp2 [21,22]	1 thread: 109.5 2 thread: 54.7 4 thread: 51.7	132.81	956	139	1.08	20
PerfOMR1 § 5	**1 thread: 7.31** **2 thread: 3.65**	2.13	2181	171	2.57	37
PerfOMR2 § 6	1 thread: 39.64 2 thread: 19.84	**0.56**	**583**	140	1.03	20

Table 2: Comparison of cost metrics. Costs are per recipient. The bulletin contains $N = 2^{19}$ messages, of which $\bar{k} = k = 50$ are pertinent to the recipient. ms/msg and Bytes/msg are all amortized over N messages. Each message has 612 bytes of payload (as in [21,22]).

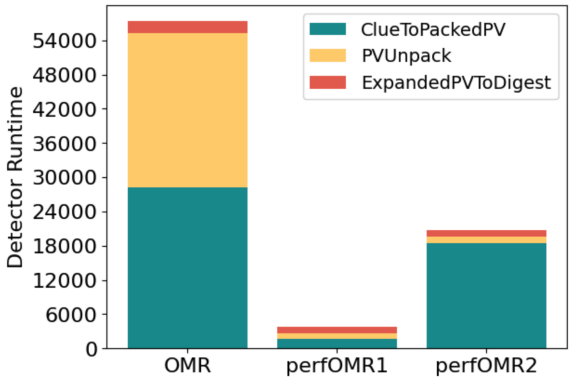


Figure 5: Comparison of runtime of each step for OMRp2 in [21], and our constructions PerfOMR1 and PerfOMR2 with $N = 2^{19}$ and $\bar{k} = k = 50$. We set $\mathsf{v} = 1$ for both of our constructions for fair comparison.

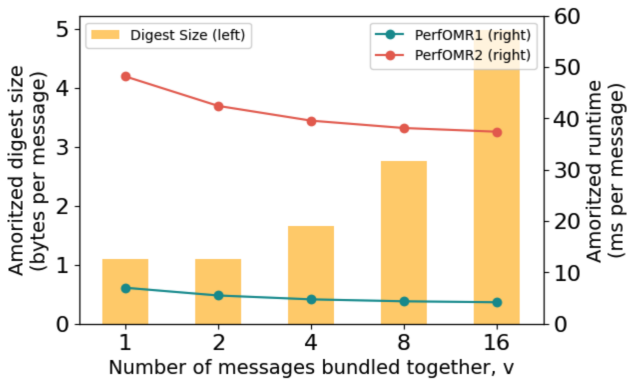


Figure 6: The runtime and digest size of our two schemes with respect to the value of v.

v linearly for both schemes (except for $v = 2$ which comes for free due to the SIMD nature of BFV, which can also improve the OMRp2 construction in [21,22] but not by much).

Larger N and $\bar{k}$. To show scalability, we also test $N = 2^{21}$ and $N = 2^{23}$ for $\bar{k} = k = 50$; and fixing $N = 2^{19}$, we also

		$k = \bar{k} = 50$			
	N	Amortized runtime (ms/msg)	Total runtime (s)	Amortized digest size (Bytes/msg)	Total digest size (MB)
PerfOMR1	2^{19}	7.31	3931.65	2.57	1.35
	2^{21}		15868.37	0.48	
	2^{23}		64701.09	0.12	
PerfOMR2	2^{19}	39.64	20953.45	1.03	0.54
	2^{21}		82826.57	0.26	
	2^{23}		330985.56	0.06	

		$N = 2^{19}$			
	$k = \bar{k}$	Amortized runtime (ms/msg)	Total runtime (s)	Amortized digest size (Bytes/msg)	Total digest size (MB)
PerfOMR1	50	7.31	3931.65	2.57	1.35
	100	9.29	4874.03	4.71	2.47
	150	11.15	5847.55	9.34	4.67
PerfOMR2	50	39.96	20953.45	1.03	0.54
	100	41.58	21797.76	1.63	0.81
	150	42.85	22465.38	2.16	1.08

Table 3: Performance when N and $k = \bar{k}$ varies.

test $\bar{k} = k = 100$ and $\bar{k} = k = 150$. As shown in Table 3, the total runtime scales linearly with N, and grows slightly with larger $\bar{k}$. Digest size is essentially independent of N and scales linearly with $\bar{k}$. The scaling behavior is essentially the same as the prior construction OMRp2 in [21,22] and matches the asymptotic analysis.[15]

Smaller $\varepsilon_n, \varepsilon_p$. To achieve even better $\varepsilon_n, \varepsilon_p$, we need to increase our parameters. For example, for $\varepsilon_n = 2^{-80}$ and $\varepsilon_p = 2^{-38}$ (which we believe is essentially enough for almost all real-world applications) for $\bar{k} = k = 45$ and $N = 2^{19}$, we need to make $r = 42$, $\ell = 4$, $C = 30$ for PerfOMR1 and other parameters remain unchanged.[16] The runtime, by our estimation, is only about 2x slower and other metrics remain roughly the same based on our test. PerfOMR2 and OMRp2 in [21,22] requires a similar parameter change and slow down.

[15] As discussed in Remark 5.5, only when the payload size is large enough, does the runtime benefit from having sparse encoding and depends only on polylog of k. Of course, if k is too large, one can simply concatenate multiple payloads together to form a payload as large as 65536 bytes and take advantage of the sparse coding.

[16] Note that $\bar{k}$ is reduced from 50 to 45 for simplicity. Otherwise, we need to increase K to 56 instead, which introduces another BFV ciphertext. This makes the runtime about 30% slower and the digest size 10% larger based on our estimation.

7.2 Integration Considerations

Lastly, we discuss some system aspect of integrating the improved OMR in real-world applications, exemplified by the Zcash cryptocurrency [18] as analyzed in [21] (for detection cost, we consider Bitcoin-scale applications).

Clue key distribution. To integrate OMR, senders need to obtain the prospective recipient's clue key to generate clues. As in [21], we consider Zcash' Unified Addresses mechanism [19] to include a clue key as an extension of the recipient's public address in a backward-compatible way, and extend the payment URIs similarly [30]. The clue key size of our PerfOMR1 is only 2.13KB, which can easily fit in a standard QR code that stores up to 3KB of data. Furthermore, our PerfOMR2 clue key size is only 0.59KB. Therefore, the clue key distribution is no longer as complicated as [21].

Clue embedding. For PerfOMR1, a clue of size 2181 bytes needs to be attached to every payload. This is larger than the 1.3kB of data on-chain per such payment. On the other hand, for PerfOMR2, the clue is only 583 bytes, much smaller than the one in OMRp2. This allows smaller on-chain data size. Including the clue in a transaction, we can simply use the `OP_RETURN` data that Zcash supports as discussed in [21].

Detection cost. The computational cost of detectors for OMRp2 is roughly $1.95 per million payments scanned (4-thread), using commodity could computing.[17] On the other hand, using our PerfOMR1, the cost is only $0.12 per million payments scanned (2-thread). Using our PerfOMR2, the cost is $0.88 per million payments scanned (2-thread). For Bitcoin-scale applications of roughly 300,000 payments per day,[18] our costs are $1.12/month and $7.88/month respectively, while prior work's cost is $17.56/month.

Acknowledgements

We are grateful to the anonymous reviewers for their insightful comments and suggestions.

This material is based upon work supported by DARPA under Contract No. HR001120C0085. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

[1] Angel, S., Chen, H., Laine, K., Setty, S.T.V.: PIR with compressed queries and amortized query processing. In: 2018 IEEE S&P. IEEE Computer Society Press (2018)

[2] Beck, G., Len, J., Miers, I., Green, M.: Fuzzy message detection. ACM CCS 2021

[3] Ben Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE S&P. pp. 459–474 (2014)

[4] Bethencourt, J., Song, D.X., Waters, B.: New techniques for private stream searching. ACM Trans. Inf. Syst. Secur. **12**, 16:1–16:32 (2009)

[5] Bittau, A., Erlingsson, Ú., Maniatis, P., Mironov, I., Raghunathan, A., Lie, D., Rudominer, M., Kode, U., Tinnes, J., Seefeld, B.: Prochlo: Strong privacy for analytics in the crowd. In: SOSP. pp. 441–459 (2017)

[6] Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: Zexe: Enabling decentralized private computation. In: 2020 IEEE S&P (SP). pp. 947–964 (2020)

[7] Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: CRYPTO 2012. LNCS, Springer (Aug 19–23, 2012)

[8] Chor, B., Gilboa, N., Naor, M.: Private information retrieval by keywords (1998), `http://eprint.iacr.org/1998/003`

[9] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: 36th FOCS. pp. 41–50. IEEE Computer Society Press (Oct 23–25, 1995)

[10] Corrigan-Gibbs, H., Boneh, D., Mazières, D.: Riposte: An anonymous messaging system handling millions of users. In: 2015 IEEE S&P. pp. 321–338 (2015)

[11] Curtis, B., Lefebvre, C., Virdia, F., Göpfert, F., Owen, J., Ducas, L., Schmidt, M., Albrecht, M., Player, R., Scott, S.: Security estimates for the learning with errors problem, `https://bitbucket.org/malb/lwe-estimator/src/master/`

[12] Danezis, G., Diaz, C.: Space-efficient private search with applications to rateless codes. In: FC'07. p. 148–162. Springer (2007)

[13] Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), `https://ia.cr/2012/144`

[14] Finiasz, M., Ramchandran, K.: Private stream search at almost the same communication cost as a regular search. In: Knudsen, L.R., Wu, H. (eds.) Selected Areas in Cryptography. pp. 372–389. Springer, Berlin, Heidelberg (2013)

[15] Geelen, R., Vercauteren, F.: Bootstrapping for bgv and bfv revisited. J. Cryptol. **36**(2) (mar 2023)

[17] Using GCP `e2-standard-4`, 4 vCPUs, billed at $0.136/hour.

[18] `https://ycharts.com/indicators/bitcoin_transactions_per_day`, retrieved 2023-10-17.

[16] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: ACM Symposium on Theory of Computing. p. 169–178. STOC '09, ACM (2009)

[17] Halevi, S., Shoup, V.: Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481 (2020), https://eprint.iacr.org/2020/1481

[18] Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash Protocol Specification Version 2021.2.14. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf

[19] Hopwood, D., Wilcox, N., Hornby, T., Grigg, J., Bowe, S., Nuttycombe, K., Lai, Y.T.: Zcash improvement proposal 316: Unified addresses and unified viewing keys. https://zips.z.cash/zip-0316 (Apr 2021)

[20] Jakkamsetti, S., Liu, Z., Madathil, V.: Scalable private signaling. Cryptology ePrint Archive, Paper 2023/572 (2023), https://eprint.iacr.org/2023/572, https://eprint.iacr.org/2023/572

[21] Liu, Z., Tromer, E.: Oblivious message retrieval. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology – CRYPTO 2022. pp. 753–783. Springer Nature Switzerland, Cham (2022), full version: Cryptology ePrint Archive 2021; internal citations follow the latter's numbering

[22] Liu, Z., Tromer, E., Wang, Y.: Group oblivious message retrieval. S&P 2024. Full version on eprint https://ia.cr/2023/534 (2023)

[23] Liu, Z., Tromer, E., Wang, Y.: Perfomr: Oblivious message retrieval with reduced communication and computation. Cryptology ePrint Archive, Paper 2024/204 (2024), https://eprint.iacr.org/2024/204, full version of this paper. Available on print: https://eprint.iacr.org/2024/204

[24] Liu, Z., Wang, Y.: Amortized functional bootstrapping in less than 7ms, with $\tilde{O}(1)$ polynomial multiplications. Asiacrypt 2023, https://eprint.iacr.org/2023/910, https://eprint.iacr.org/2023/910

[25] Lund, J.: Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/ (Oct 2018)

[26] Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM (2013)

[27] Madathil, V., Scafuro, A., Seres, I.A., Shlomovits, O., Varlakov, D.: Private signaling. USENIX Security 2022 (2022)

[28] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL (Nov 2020), Microsoft Research, Redmond, WA.

[29] Noether, S.: Ring signature confidential transactions for monero. Cryptology ePrint Archive, Paper 2015/1098 (2015), https://eprint.iacr.org/2015/1098, https://eprint.iacr.org/2015/1098

[30] Nuttycombe, K., Hopwood, D.: Zcash improvement proposal 321: Payment request URIs. https://zips.z.cash/zip-0321 (Aug 2020)

[31] Oblivious message retrieval implementation. https://github.com/ZeyuThomasLiu/ObliviousMessageRetrieval (Dec 2021)

[32] Ostrovsky, R., Skeith, W.E.: Private searching on streaming data. In: CRYPTO (2005)

[33] PALISADE lattice cryptography library (release 11.2). https://palisade-crypto.org/ (Jun 2021)

[34] Peikert, C., Vaikuntanathan, V., Waters, B.: A framework for efficient and composable oblivious transfer. In: CRYPTO 2008. pp. 554–571. Springer (2008)

[35] Player, R.: Parameter selection in lattice-based cryptography. Ph.D. thesis, Royal Holloway, University of London (2018)

[36] Pu, S., Thyagarajan, S.A., Döttling, N., Hanzlik, L.: Post quantum fuzzy stealth signatures and applications. CCS 23 (2023)

[37] Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation pp. 169–179 (1978)

[38] Seres, I.A., Pejó, B., Burcsi, P.: The effect of false positives: Why fuzzy message detection leads to fuzzy privacy guarantees? In: Eyal, I., Garay, J. (eds.) Financial Cryptography and Data Security. pp. 123–148. Springer International Publishing, Cham (2022)

[39] Szefer, J.: Survey of microarchitectural side and covert channels, attacks, and defenses. Journal of Hardware and Systems Security **3**, 219–234 (Sep 2019)

[40] Wolinsky, D.I., Corrigan-Gibbs, H., Ford, B., Johnson, A.: Dissent in numbers: Making strong anonymity scale. In: OSDI 12. pp. 179–182. USENIX (Oct 2012)